

DOS EXTRA

Sonderausgabe DOS EXTRA

Nr. 4 1993

ISSN 0934-2842
DMV-Verlag
Eine Publikation der
DOS INTERNATIONAL

Tolle Animationen mit Sound

- ◆ die wichtigsten Routinen für Spielefreaks
- ◆ bewegte Laufschriften
- ◆ skalierbare Sprites mit Geräuscheffekten

KURS: RICHTIG PROGRAMMIEREN

- ◆ die ersten Schritte
- ◆ Programmiergeheimnisse entschlüsselt

Superschnelle Grafikroutinen

Referenz: die wichtigsten Grafikformate

NÜTZLICHE WINDOWS-TOOLS

Track-Menüs ◆ Bildschirmschoner ◆ Mastermind



Keine halben Sachen – hier gibt es EXTRA-starke Themen!

Ausgabe 13



Spiele Pur
Alles zum Thema Flugsimulation: Die besten Programme, Tips & Tricks, Programmierung • Tips & Tricks zu den beliebtesten adventures auf dem PC • Hardware: Soundkarten und Joysticks

Ausgabe 14



Tips, Tricks & Utilities
Backup: Mit Komfort und Geschwindigkeit • Programmierung mit Interrupts • Entschlüsselung von Strichcodes • Programmierung: Quelltextoptimierung • Programmieren mit Interrupts

Ausgabe 15



Programmierung, Grundlagen
Programmierung: Turbo Pascal im Überblick, Grundlagen der Programmierung • Grundlagen: MS-DOS für Einsteiger, Batchdateien für Profis, Tips zur Druckerinstallation

Ausgabe 16



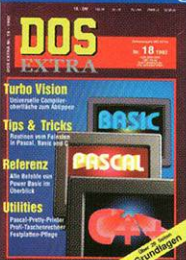
Assembler, Systemprogrammierung, Tips & Tricks für Programmierer
Assembler: Grundlagen Grafikkarten, Befehlsreferenzen • System: Treiberprogrammierung, Speicherverwaltung, GEM, Interruptcontroller

Ausgabe 17



Grundlagen, Utilities
Tips & Tricks
Grundlagen: Turbo Vision-Kurs, Turbo Pascal, Turbo C • Jede Menge Tips & Tricks für Programmierer • Utilities: Format-Befehl der S-Klasse, Diskcopy

Ausgabe 18



Leider schon vergriffen.

Ausgabe 19



Alles über Grafik
Raytracer: Entwerfen Sie eigene Welten • Grafik selbstgemacht: Vektorgrafik, Animation mit Quick Basic, Superschnelle Fraktale • Spriteprogrammierung • EGA- und VGA-Karten: Register auf einen Blick

Ausgabe 20



Programmierung
Grundlagen: Objektorientierte Programmierung, Turbo Pascal • Tools: Soundblaster, Makrocompiler, Grafiktreiber • Programmierung für Profis: Turbo Vision, Turbo Pascal, Interrupts, Windows

Ausgabe 21



Programmierung, Assembler
Grundlagen: Schnelleinstieg in Assembler • Assembler in der Praxis • Viele Tips & Tricks für Assembler • Systemprogrammierung: Alle Prozessorbefehle, BIOS-Interrupts, DOS-Funktionen

Ausgabe 1/93



Spiele-Programmierung
Top-Routinen zur Spieleprogrammierung • Rollenspiel: 3D-Grafik in Echtzeit • Strategiegeheimnisse: Die besten Algorithmen, Listing für Profi-Schach • Alle Spiele können Sie auf Diskette bestellen.

Ausgabe 2/93



Windows
Programmbaukasten: MDI, Werkzeugleiste, Statuszeile, echtes Multitasking in C • Super-Tool: Windows-Nachrichten sichtbar machen • Utilities: Speicherfresser aufgespürt, Kfz-Kosten-Verwaltung, Windows-Packer

Ausgabe 3/93



3 Generationen Soundkarten und alles für Multimedia:
PC im Eigenbau, Karten, Shareware, Programmierung: Video und Sound in vielen Formaten. Und über 15 Seiten Tips und Tricks zu Sound

Vervollständigen Sie jetzt Ihre Sammlung. Wählen Sie sich einfach die Ausgaben der DOS EXTRA, die Ihnen noch fehlen und bestellen Sie gleich mit dem Coupon. Sollten Sie hier eine Ausgabe vermissen, fragen Sie bei uns nach. Vielleicht haben wir die von Ihnen gewünschte Ausgabe noch auf Lager. Übrigens, wenn Ihnen der Coupon nicht reicht, kreuzen

Sie einfach auf dieser Seite die Hefte an, die Sie benötigen und senden oder faxen Sie uns die ganze Seite zu.

Fax **0 89/24 01 32 15**
Tel. 0 89/24 01 32 22

oder schicken Sie den Coupon an:
DMV-Verlag Leser-Service, CSJ,
Postfach 14 02 20, 80452 München

JA, ich möchte meine DOS EXTRA-Sammlung vervollständigen.

Ich bestelle folgende Exemplare und bezahle per Bankeinzug

Konto-Nr. _____

BLZ/Bankverbindung _____

oder lege einen Scheck bei (quitierte Rechnung liegt der Sendung bei).

— Stück für je DM 18,- der Ausgabe _____ : DM _____

— Stück für je DM 18,- der Ausgabe _____ : DM _____

Zuzüglich Porto (innerhalb der BRD) _____ : DM 4,-

Gesamt summe _____ : DM _____

Name, Vorname _____

Straße _____

PLZ, Ort _____

Unterschrift _____

Dieses Angebot gilt nur innerhalb der Bundesrepublik Deutschland.



Bewegte Geräusche

Grafik und Sound sind die beiden Schlagworte in der heutigen Computerwelt. Gerade im kreativen Bereich wird der PC zunehmend noch mehr an Bedeutung gewinnen. Durch immer bessere Grafikprogramme kann bald jeder in kürzester Zeit die tollsten Kunstwerke schaffen, natürlich unterstützt durch zahlreiche Clipart-Sammlungen.

Auch in der Filmbranche hat sich der Einsatz von Rechentechnik bewährt, was ja beispielsweise die Filme Jurassic Park oder Terminator II beweisen. Dabei handelt es sich keinesfalls um einfache bewegte Grafiken, sondern um animierte Objekte, die kaum mehr einen Unterschied zu realen Gegenständen oder Objekten erkennen lassen. Auch in der Werbebranche boomt der PC-Einsatz. Gerade die aufkommende Morphing-Software verdrängt immer mehr die belanglosen Wirbelgrafiken und ersetzt diese durch realistisch gestaltete Animationen. Kurzum, Multimedia ist mit gewaltigen Schritten auf dem Vormarsch. Natürlich kommt diese Entwicklung nicht von weit her. Erst die immer

besser und billiger werdende Hardware macht es möglich.

Moderne Grafikkarten, auf denen hochspezialisierte Grafikprozessoren arbeiten, verwandeln den PC von der reinen Textmaschine in eine leistungsfähige Grafik-Workstation. Und das ist erst der Anfang. Es vergeht derzeit kaum eine Woche, in der nicht neue Grafikboards mit noch höherer Leistung angekündigt werden.

Mindestens genauso dramatisch ist die Entwicklung im Soundbereich. Längst vergessen sind die Zeiten, in denen lediglich ein winziger Lautsprecher aus dem PC-Gehäuse piepste. 16-Bit-Soundkarten, die Tonaufnahme und -wiedergabe in CD-Qualität erlauben, erobern zur Zeit die Erweiterungssteckplätze unserer Rechner.

Die dabei anfallende Datenflut bewältigen die immer größer und leistungsfähiger werdenden Festplatten. Wo das nicht reicht, springen CD-ROM Laufwerke mit mehr als 600 MByte Kapazität in die Bresche.

Für Programmierer ist die Kombination von Grafik und Sound natürlich ebenfalls äußerst interessant. Da es zu diesem Thema leider noch nicht viel Literatur gibt, beschäftigt sich diese DOS EXTRA ausführlich damit. Ein kleines Beispielprogramm wird Sie in die Geheimnisse der Sound- und Grafikprogrammierung einweihen. Besonders interessant ist, daß die im Vordergrund ablaufende Animation durch Musik im Hintergrund und eventuell durch zusätzliche Geräusche unterstützt wird. Das spielt vor allem bei Spielen eine große Rolle.

Gehen Sie nun auf die Reise durch die Basic-, Pascal- oder C-Sprache. Beim Testen und Ausprobieren der vielen Programme und Utilities wünsche ich Ihnen viel Spaß!

Kerrin Uhlmann

DOS EXTRA

EINSTEIGER-PROGRAMMIERUNG

Programmier-Grundlagen

Das Geheimnis der Programmierung

Einsatz von Entwicklungsumgebungen

Der Weg zum ersten Programm

Datenausgabe in Programmen

Basiselement der Programmierung

Datentypen

Jonglieren mit Zahlen und Zeichen

6

10

16

20

GRAFIK-PROGRAMMIERUNG

Teil 1: Punkte setzen und lesen

Geschwindigkeit ist keine Hexerei

Teil 2: Linien ziehen

Schneller als die Polizei erlaubt

Teil 3: Kreise und Ellipsen zeichnen

Schwerarbeit: Kreise und Ellipsen

Teil 4: Flächen füllen

Geh aufs Ganze – Flächen füllen in Rekordzeit

Teil 5: Textausgabe im Grafikmodus

Flexibilität und Geschwindigkeit kontra VGA-BIOS

Grafikformate intern

Aufgeschlüsselte Standards

32

40

46

53

61

65

WINDOWS-PROGRAMMIERUNG

3D-Animationen

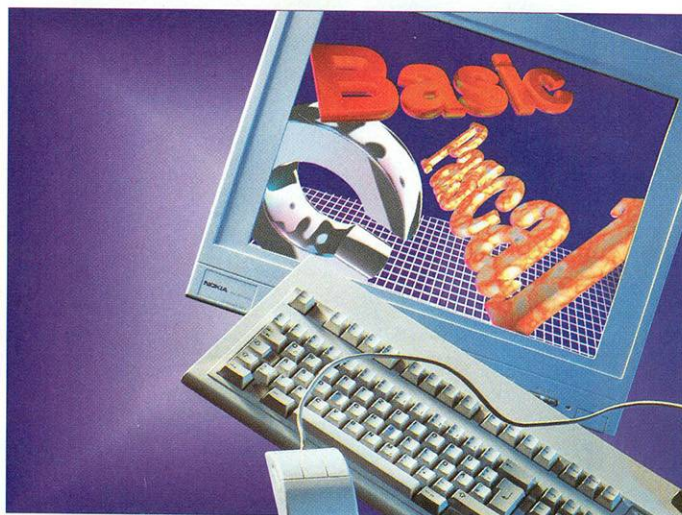
3D-Objekte im Raum

Selbstdefinierte Track-Menüs

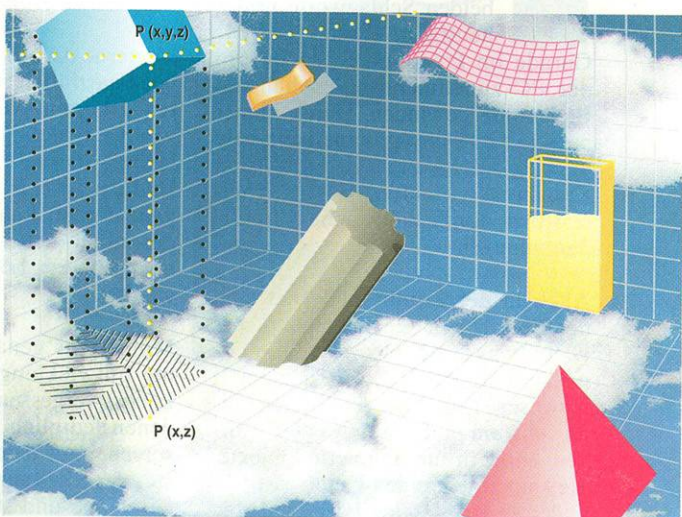
Menüs vom Meister

77

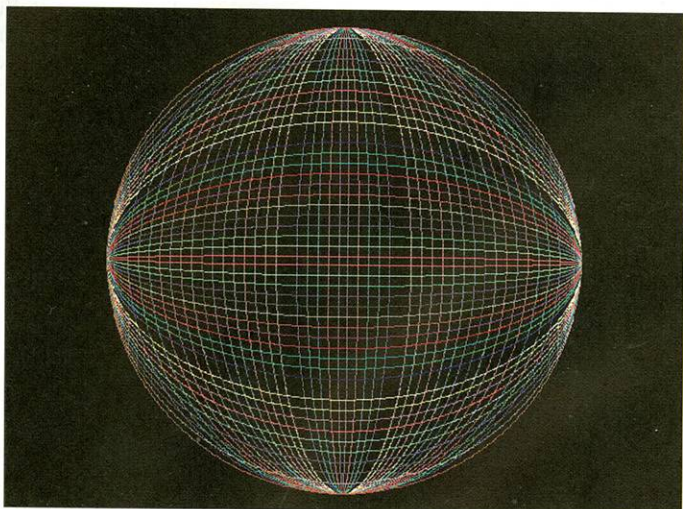
82



Hier geht es nicht speziell um eine Programmiersprache, sondern um den Zweck, die Verwendung und die Elemente der Programmierung. **Seite 6**



Mit Hilfe von Assembler und einigen interessanten Tricks können Sie den Aufbau von Grafiken um zirka 1000 Prozent beschleunigen. **Seite 32**



Diese Grafik bauen Sie um 400 Prozent schneller auf, wenn Sie von konventionellen Programmialgorithmen abweichen. **Seite 46**

INHALT

4/93

Bildschirmschoner

Aktiver Strahlenschutz

Denk- und Strategiespiel für Windows

Color Code: Mastermind für Windows

Schnelle Grafikausgabe

Kesse Kurven

MULTIMEDIA

Grafik und Sound

Der PC spielt auf

Soundblaster-Programmierung

DSP, das Tor zur Tonwelt

Das MOD-Dateiformat entschlüsselt

Das MOD-Dateiformat unter der Lupe

Unit für Soundblaster-Einstellungen

Soundblaster unter Kontrolle

Abspiel-Unit für MOD-Dateien

Sounds aus dem Hintergrund

MOD-Editor

MOD-Dateien à la carte

Timer-Programmierung

Unterbrechung nach Maß

Laufschrift, Sprites und Texturen

Animierte Demos mit Pep

Verwendung externer MIDI-Geräte

MIDI – Schlüssel zur digitalen Musikwelt

RUBRIKEN

Editorial

Bewegte Geräusche

Impressum

85

90

93

96

98

103

105

107

119

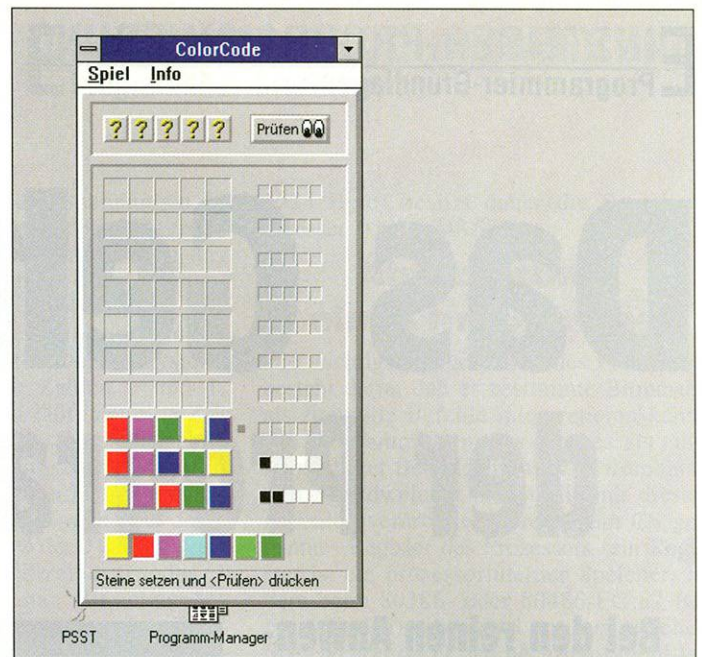
130

132

145

3

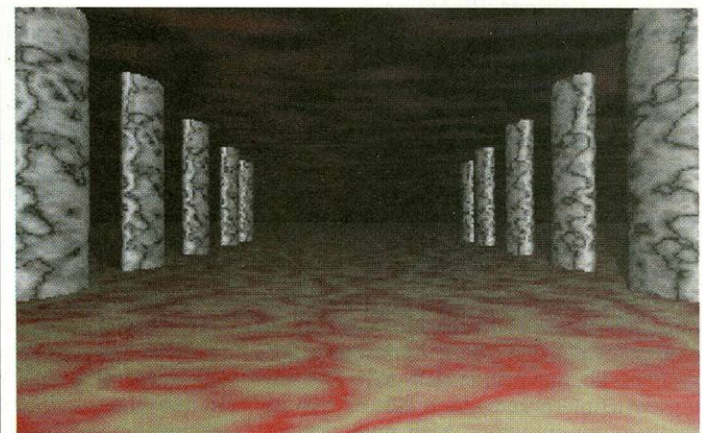
104



Eine Mastermind-Kopie in Borland C++ zeigt Ihnen, wie Sie mit wenigen Programmzeilen ein tolles Windows-Programm schreiben. **Seite 90**



Grafik und Sound gehören zusammen. Deshalb zeigt Ihnen diese Rubrik, wie Sie MOD-Dateien im Hintergrund von animierten Grafiken ablaufen lassen. So werden Sie in die Welt der Sound- und Multimedia-Programmierung eingeführt. **Seite 96**



Dieses Hintergrundbild finden Sie im Demoprogramm »simple_s.pas«. Durch diesen Gang bewegen sich Bälle und machen dazu entsprechende Geräusche. **Seite 132**

Das Geheimnis der Programmierung

Bei den reinen Anwendern gelten sie als Halbgötter in Jeans: die Programmierer, welche – so die allgemeine Vorstellung – 24 Stunden am Tag arbeiten, nur von Kaffee und Zigaretten leben und dabei noch so wunderschöne Produkte wie Windows oder Corel Draw ausbrüten. Dabei ist der Weg vom Anwender zum Programmierer nicht so schwierig, wie oft vermutet.



Ein falscher Weg auf dem Weg zum Programmierer ist ein Crash-Kurs in einer einzigen Programmiersprache. Diese Angebote liefern ausschließlich Kochrezepte nach dem Motto »Man nehme, man tue, und heraus kommt ein Programm«.

Das Problem bei der Programmierung besteht jedoch darin, daß jede Aufgabe individuell bearbeitet werden muß. So kann der aufgezeigte Weg X bei Aufgabe Y völlig fehl am Platz sein. Der richtige Weg besteht daher darin, sich zunächst einmal ein breites Grundlagenwissen anzueignen und sich darüber klar zu werden, was im Computer bei der Abarbeitung eines Programms überhaupt vor sich geht. Erst mit diesem Hintergrundwissen läßt es sich effektiv programmieren. Man muß sich gewissermaßen in den Prozessor hineinversetzen können, um zu beurteilen, ob ihm die eigenen Kreationen wohl gefallen oder

nicht. Schließlich sollte man sich auch darüber Gedanken machen, welche Programmiersprache einzusetzen ist. Dabei dürfen nicht nur die Annehmlichkeiten für den Programmierer (wie komme ich am schnellsten und einfachsten zum Ziel?) eine Rolle spielen, sondern vielmehr auch die Belange der Aufgabe selbst und des Computers (welches Programm läßt sich am schnellsten verarbeiten?).

Elektronik mit Gedächtnis

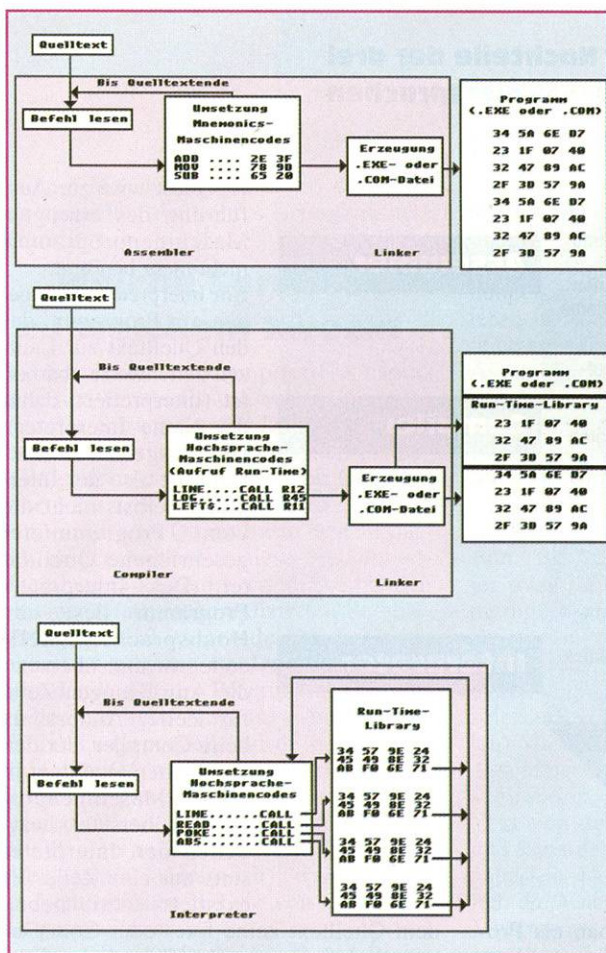
Das Herz eines jeden Computers ist der Mikroprozessor. Hierbei handelt es sich, vereinfacht gesagt, um ein Siliziumplättchen, auf dem sich Millionen von Transistoren (dies sind elektrische Schalter) und mindestens genauso viele elektrische Leitungen befinden. Hierbei sollten Sie allerdings nicht an das Stromkabel des Fernsehgeräts denken. Einige dieser Leitungen

sind über die typischen »Beinchen« eines Chips nach außen geführt. Man kann an sie eine elektrische Spannung anlegen oder auch nicht.

Der Zweck eines Prozessors besteht darin, daß er in Abhängigkeit der Kombination der angelegten Spannungen an bestimmten Leitungen unterschiedliche Ergebnisse (Spannungskombinationen) liefert. Mehr noch, nicht nur die aktuell angelegten Spannungen, sondern auch die zuvor angelegten Spannungen spielen für das Ergebnis eine Rolle. Die Schaltung auf dem Siliziumplättchen hat also ein Gedächtnis. Man nennt sie in der Elektrotechnik »Schaltwerk«, um sie von »Schaltnetzen«, die kein Gedächtnis aufweisen, zu unterscheiden.

Mathematisch kann man das Ergebnis in der Form

$$A(T) = F(I(T), I(T-1), I(T-2), \dots, I(T-n))$$



Schematisches Modell des Ablaufs vom Quellprogramm bis hin zur ausführbaren Anwendung bei den drei Grundtypen der Programmiersprachen: Assembler-Codierung, Compiler- und Interpretersprachen

beschreiben, was bedeutet, die Ausgabe A zum Zeitpunkt T ist eine Funktion der Eingaben I zum Zeitpunkt T und den davorliegenden Zeitpunkten T-1, T-2 und so weiter.

Damit der Prozessor weiß, zu welchen Zeitpunkten die Eingaben relevant sind (was ist T-1 oder T-2?), wird er über eine zusätzliche Leitung getaktet. An ihr liegt das Signal eines Quarzes an, das mehrere Millionen Mal in der Sekunde seinen Zustand (Spannung liegt an = High, Spannung liegt nicht an = Low) wechselt. Nur dann, wenn die Spannung an der Taktleitung anliegt, arbeitet der Prozessor (sogenanntes »synchrones Schaltwerk«). Die Anzahl der Takte pro Sekunde wird auch Taktrate genannt und in Megahertz (Millionen Spannungswechsel pro Sekunde, Abkürzung: MHz) angegeben. Je größer die Taktrate, desto mehr Operationen kann die Prozessorschaltung ausführen. Ein mit 66 MHz getakteter Prozessor beispielsweise ist doppelt so schnell wie ein mit 33 MHz getakteter.

Die Leitungen, über die die Ein- und Ausgaben getätigt werden, heißen Datenleitungen. Der 80386- und der 80486-PC enthalten davon 32, der 80286 nur 16 Datenleitungen.

Über Bits und Bytes

Das Problem für den Programmierer besteht darin, daß er mit elektrischen Spannungen nicht viel anfangen kann. Man hat

daher die Spannungen in mathematische Begriffe umgesetzt und hierzu das Dualzahlensystem benutzt. Dieses beinhaltet, im Gegensatz zum gewöhnlich benutzten Dezimalsystem, nur die Zahlen 0 und 1. Hiermit läßt sich eine Zuordnung der elektrischen Spannungszustände High und Low in Zahlen vornehmen. High wies man der 1 und Low der 0 zu.

Eine einzelne 0 oder 1 nennt man Bit, eine Ansammlung von jeweils 8 Bit heißt Byte. Um also eine bestimmte Kombination der Spannungszustände von 32 Datenleitungen anzugeben, müssen 32 Bit oder 4 Byte zur Verfügung stehen (4 Byte werden Langwort, 2 Byte Wort genannt). Um den dezimalen Wert einer Dualzahl anzugeben, geht man wie beim Dezimalsystem vor, wobei jede höhere Stelle statt einer Multiplikation mit 10 nur einer Multiplikation mit 2 entspricht. Zur Unterscheidung einer Dualzahl

von einer Dezimalzahl stellt man ihr das Zeichen »%« hinten an:

$$1204_{10} = 1 \cdot 10^3 + 2 \cdot 10^2 + 0 \cdot 10^1 + 4 \cdot 10^0$$

$$10101_{10} = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ = 16 + 0 + 4 + 0 + 1 \\ = 21$$

Mit Hilfe einer Dualzahl kann jeder Spannungs-kombination eine eindeutige Zahl zugeordnet werden, wenn jede Stelle der Dualzahl einer bestimmten Leitung entspricht. Aus diesem Grund wurden die Datenleitungen des 80386 oder 80486 mit D0 bis D31 bezeichnet: Leitung D0 korrespondiert zur untersten Stelle der 32-Bit-Dualzahl, Leitung D31 zur obersten.

Speicher: Quelle und Ziel der Bits

Wenn man sich diese Überlegungen ansieht, muß man sich fragen, woher die Bitmuster, die irgendwann an den Datenleitungen anliegen, überhaupt kommen. Die Antwort: aus dem Speicher des PC. Die Datenleitungen sind über mehr oder weniger komplizierte Schaltungen mit den Speicherchips verbunden, so daß über sie der Datenaustausch zwischen Speicher und Prozessor vorgenommen wird. Um die Speicherchips vernünftig zu organisieren, hat man ihnen jeweils eine eindeutige Adresse zugeordnet:

Ein PC mit 1 MByte Arbeitsspeicher (1 MByte = 1024 KByte und 1 KByte =

1024 Byte) besitzt daher die Speicheradressen 0 bis 1 048 575.

Abarbeitung eines Programms

Die wichtigste Eigenschaft des Prozessors besteht darin, daß er bestimmte Bitmuster als sinnvolle Befehle interpretieren kann. Ein Programm ist nichts anderes als eine Folge dieser Befehlsbitmuster in Kombination irgendwelcher Daten, die mit diesen Befehlen verarbeitet werden. Ein CS genanntes Register des Prozessors (ein Register ist ein prozessorinterner Speicher, in dem beim 80386- oder 80486-PC 32 Bit Platz finden) gibt dabei an, aus welcher Speicheradresse er den nächsten Befehl lesen soll, ein Register DS zeigt auf die Speicheradresse, aus der Daten zu lesen oder in die Daten zu schreiben sind. Die Adressierung einer bestimmten Speicherzelle erfolgt über weitere, Adreßleitungen genannte Verbindungen des Prozessors, wovon der 80386 und 80486 jeweils 32 aufweisen. Damit lassen sich theoretisch $2^{32} = 4\,294\,967\,296$ Byte oder 4 GByte (1 GByte = 1024 MByte) eindeutig ansprechen.

Welcher Prozessorbefehl zu welchem Bitmuster gehört, hängt vom Aufbau der Schaltung auf dem Siliziumplättchen ab und ist vom Prozessorhersteller dokumentiert.

So wird bei den Intel-Prozessoren durch die Bitfolge

000001010001000000000000

zum augenblicklichen Inhalt des AX-Registers (hierbei handelt es sich um eines der sogenannten Allzweckregister des Prozessors, mit denen sich zum Beispiel rechnen läßt) der Wert 16 hinzuaddiert. Um nicht mit endlosen Bitfolgen operieren zu müssen, hat man in der Computerwelt ein drittes Zahlensystem, das sogenannte Hexadezimalsystem (Basis 16), eingeführt. Dieses bietet sich an, da sich mit jeweils 4 Bit eine Ziffer im Hexadezimalsystem darstellen läßt. Da eine Hex-Ziffer Werte von 0 bis 15 annehmen kann, führte man zusätzlich zu den Ziffern 0 bis 9 die Buchstaben A bis F ein, wobei A der Dezimalzahl 10 und F der Dezimalzahl 15 entspricht. Die Umrechnung dezimal-hexadezimal ist einfach, wenn man bedenkt, daß jede höhere Stelle im hexadezimalen System einer Multiplikation mit 16 gleichkommt. Eine Hex-Zahl wird um das Zeichen »h« ergänzt, um es von Dual- und Dezimalzahlen zu unterscheiden:

$$A4Fh = 10 \cdot 16^2 + 4 \cdot 16^1 + 15 \cdot 16^0 \\ = 2560 + 64 + 15 \\ = 2639$$

Die Umrechnung dual-hexadezimal ist noch einfacher, da jeweils 4 Bit genau einer Hex-Ziffer entsprechen:

$$0101001101101101_{10} \\ = 0101\ 0011\ 0110\ 1101 \\ = 5\ 3\ 6\ Dh \\ = 536Dh$$

Die obige Befehlssequenz hätten Sie also

Die Vor- und Nachteile der drei Grundtypen von Programmiersprachen

Vorteile

Assembler

- sehr schneller und kompakter Code
- direkter Einfluß des Programmierers auf den erzeugten Code
- Assembler für Programmausführung nicht erforderlich

Compiler

- schneller Code (direkte Ausführung als Exe- oder Com-Datei)
- einfache Programmierung dank Hochsprache
- sehr leicht zu erlernen
- Compiler für Programmausführung nicht erforderlich
- einfache Portierung auf andere Systeme

Interpreter

- wird kostenlos mit dem Betriebssystem ausgeliefert (Quick Basic)

Assembler

- komplizierte Programmierung
- schwierig zu erlernen
- Assembler ist extra zu kaufen
- Portierung auf andere Systeme nicht möglich

Compiler

- große Programmdateien aufgrund Laufzeitbibliothek
- Programmierer hat keinen direkten Einfluß auf erzeugten Code
- relativ teure Anschaffung

Interpreter

- langsame Programmausführung
- keine Portierung möglich (Quick Basic)
- Interpreter zur Programmausführung erforderlich
- geringer Sprachumfang (Quick Basic)

Nachteile

ler selbst wird zur Ausführung des erzeugten Maschinenprogramms nicht mehr benötigt.

Ein Interpreter ist hingegen ein Programm, das den Quelltext zur Laufzeit zeilenweise abarbeitet (interpretiert, daher der Name Interpreter). Die ausgeführte Exe-Datei ist also der Interpreter selbst, nicht die vom Programmierer geschriebene Quelldatei. Das Interpreter-Programm liest den Hochsprachen-Quellcode ein und übersetzt die Anweisungen Zeile für Zeile. Anders als beim Compiler, der den kompletten Quelltext in ein Maschinenprogramm übersetzt, bearbeitet der Interpreter stets nur eine Zeile. Es existiert somit neben

im hexadezimalen System sehr einfach als »051000h« schreiben können.

Wie gesagt, ein Programm ist nichts anderes als eine Aneinanderreihung von Befehlen und Daten. Das Problem für den durchschnittlichen Programmierer besteht darin, daß er einen solchen Code (die sogenannte Maschinensprache) im Gegensatz zum Prozessor kaum lesen kann, wenn er nicht alle Prozessorbefehle und deren Hex-Codes im Kopf hat. Zu den alten Zeiten des Commodore C64 gab es nicht wenige Freaks, die Hex-Listings lesen konnten wie andere Leute die Zeitung. Allerdings verstand der antiquierte 8-Bit-Prozessor 6502 auch viel weniger Befehle als die modernen Intel-Prozessoren 80386 und 80486.

Die Assembler-Sprache

Als wesentliche Vereinfachung benutzt man daher einen sogenannten Assembler. Hierbei handelt es sich um ein Programm, mit dessen Hilfe man statt Hex-Zahlen sogenannte Mnemonics programmiert. Diese wiederum bilden aussagekräftigere Texte, die der Assembler in die zugehörigen Hex-Codes übersetzt. Unser Beispiel würde in der mnemonischen Sprache

```
ADD AX, 10h
```

ergehen. Hieraus läßt sich leicht ersehen, daß zum Inhalt des Registers AX der Wert 16 (dezimal) zu addieren ist. Der Assembler wandelt diesen Text schließlich in die Byte-Folge »051000h« um.

Ein Assembler-Programm ist grundsätzlich also nichts anderes als ein ASCII-Text, in dem jede Zeile einen mnemonischen Befehl enthält. Das große Problem der Assembler-

Sprache besteht darin, daß man als Programmierer wirklich nur auf die eingebauten Befehle des Prozessors zurückgreifen kann und daß diese nicht unbedingt sehr leistungsfähig sind. Dazu zwei Beispiele:

Kein Prozessor der Intel-Familie versteht einen einzigen Grafikbefehl. Möchten Sie nur eine Linie über den Bildschirm zeichnen, sind sehr umfangreiche Anweisungsfolgen in Assembler erforderlich. Oder denken Sie an die Fließkommarechnung. Eine für den Menschen triviale Rechnung, beispielsweise

$$1,5 * 1,5 = 2,25$$

wird für den Assembler-Programmierer zur Qual, da der Prozessor von Haus aus nur mit ganzzahligen Daten rechnet.

Um die Komplikationen der Assembler-Programmierung zu umgehen, findet die Programmentwicklung heutzutage fast ausschließlich in sogenannten Hochsprachen wie Basic, Pascal oder C statt. Eine Hochsprache stellt dem Programmierer wesentlich leistungsfähigere Befehle als die Assembler-Sprache zur Verfügung. Damit sind die Voraussetzungen vorhanden, mit einem einzigen Befehl eine Quadratwurzel zu ziehen, eine Ellipse zu zeichnen oder eine Eingabe von der Tastatur einzulesen. Da der Prozessor diese komplexen Befehle natürlich nicht versteht, wird das Hochsprachenprogramm (der sogenannte Sourcecode oder Quelltext) in die Maschinensprache übersetzt. Hierzu gibt es spezielle Programme, die man Compiler oder Interpreter nennt.

Zwischen beiden Gattungen besteht ein wesentlicher Unterschied. Der Compiler erzeugt aus einem Quelltext ein Maschinenprogramm, das Sie anschließend wie jedes andere Programm auf der Kommandozeile starten (Exe- oder Com-Datei). Der Compi-

ler selbst wird zur Ausführung des erzeugten Maschinenprogramms nicht mehr benötigt.

Sowohl Compiler als auch Interpreter beinhalten eine sogenannte Runtime Library (Laufzeitbibliothek), die die Maschinenbefehle für alle Hochsprachenanweisungen enthält. Wenn der Compiler auf eine Anweisung zum Ziehen einer Quadratwurzel trifft, produziert er einen Befehl, der die entsprechende Routine in der Runtime Library aufruft. Diese Bibliothek ist daher auch Bestandteil jedes erzeugten (kompilierten) Maschinenprogramms. Beim Interpreter enthält die Runtime Library die Anweisungsfolgen, die bei der Übersetzung und Ausführung eines Hochsprachenbefehls angesprungen werden, sie sind also anders als beim Compiler Bestandteil des Interpreters und nicht des erzeugten Exe- oder Com-Programms.

Hochsprachen: Die Assembler-Alternative

Das Bild zeigt die jeweilige Arbeitsweise eines Compilers, Assemblers und Interpreters, die Textbox veranschaulicht die grundsätzlichen Vor- und Nachteile dieser drei Programmiervarianten.

Compiler und Assembler beinhalten ein Programm »Linker«, das die endgültige Exe- oder Com-Datei herstellt, indem es Laufzeitbibliothek und übersetzten Maschinencode zusammenfügt (linkt, bindet). Während die Compiler den Linkprozeß für den Benutzer meist unsichtbar im Hintergrund durchführen, muß der Assembler-Programmierer die vom Assembler erzeugte Objektdatei (eine Art Zwischencode zwischen Quelltext und lauf-

fähigem Maschinenprogramm) durch einen manuellen Aufruf des Linkers weiterverarbeiten.

Welche Sprache zu welchem Zweck?

Die bekanntesten Compiler-Hochsprachen sind C und Pascal, während als Interpreter-Sprache unter DOS eigentlich nur Quick Basic eine Rolle spielt, da dieses Programmiersprache mit dem Betriebssystem zusammen geliefert wird.

Der Vorteil von Quick Basic besteht darin, daß der Anwender diese Sprache gewissermaßen kostenlos erhält, verbunden mit einer relativ unkomplizierten Befehlssyntax, so daß es gerade für Anfänger schnell zu erlernen ist. Der Nachteil besteht darin, daß die Programme Interpreter-bedingt sehr langsam arbeiten (der Übersetzungsprozeß wird beim Programmablauf vorgenommen) und daß Quick Basic ausschließlich auf dem PC existiert. Wenn Sie Ihre Programme später auf eine andere Plattform (zum Beispiel Apple Macintosh) transferieren möchten, müssen Sie sie von Grund auf neu programmieren.

Anders sieht die Sache bei C und Pascal aus: Für beide Sprachen sind Compiler auf den unterschiedlichsten Systemen etabliert, so daß sich die Programme sehr leicht portieren (auf ein anderes System übertragen) lassen. Als Compiler-Sprache erzeugen sie zudem einen schnellen Maschinencode.

Beide Sprachen überfordern jedoch oft gerade den Einsteiger, da sie gegenüber Quick Basic erheblich erweiterte Funkti-

onsmerkmale anbieten, die dieser allerdings kaum nutzen dürfte.

C wird dabei häufig als noch komplizierter angesehen als Pascal, obwohl die Meinungen hier auseinandergehen. Des weiteren hat sich C im professionellen Bereich als »die« Sprache durchgesetzt.

Assembler schließlich ist völlig maschinenabhängig, da man direkt die Prozessorbefehle codiert. Zudem ist diese Sprache extrem schwer zu erlernen, weil komplizierte Befehle, für die Hochsprachen eine einzige Zeile benötigen, durch langwierige Assembler-Routinen zu ersetzen sind. Auf der anderen Seite ist ein Compiler oder Interpreter aufgrund der allgemeingültigen Laufzeitbibliothek (zum Beispiel muß die Funktion *Tastatur einlesen* sowohl Texte als auch Zahlen verarbeiten können) niemals in der Lage, einen solch effektiven (schnellen und kompakten) Maschinencode zu erzeugen, wie der Assembler-Programmierer selbst. Er weiß schließlich, ob er eine Zahl oder einen Text von der Tastatur einlesen möchte, und kann dementsprechend seine Befehle anpassen. Der Assembler übersetzt lediglich die Mnemonics in Hex-Codes, da die Mnemonics aber Prozessorbefehlen entsprechen, findet lediglich eine 1:1-Umsetzung statt, während Compiler und Interpreter Hochsprachenbefehle in eine Reihe von Maschinenbefehlen übersetzen, wobei komplexe Befehle (zum Beispiel mathematische Funktionen oder Grafikbefehle) die hierfür existierenden Routinen der Laufzeitbibliothek aufrufen. Assembler bietet sich daher in erster Linie für zeitkritische Anwendungen und als Ergänzung zur Hochsprachenprogrammierung an.

In der Tat erlauben es moderne Compiler, Assembler-Programme in Hochsprachenprogramme einzubinden.

Die Auswahl einer für Sie geeigneten Programmiersprache kann Ihnen niemand abnehmen. Vieles spricht für C oder als Alternative die Kombination Pascal und Assembler. Quick Basic ist wirklich nur für Hobbyprogrammierer akzeptabel, die kleinere Programme für den Hausgebrauch entwickeln möchten.

Die sieben Elemente der Programmierung

Unabhängig davon, für welchen Compiler oder welche Programmiersprache Sie sich entscheiden, läßt sich die Aufgabe eines Programms wie folgt definieren:

Ein Programm löst gestellte Aufgaben durch Manipulation von Daten.

Um ein allgemeines Problem zu lösen, muß es somit

- irgendwoher Daten erhalten,
- Platz zur Speicherung dieser Daten zur Verfügung stellen,
- Anweisungen zur Bearbeitung der Daten enthalten und
- die Ergebnisse der Bearbeitung an den Benutzer ausgeben.

Die Anweisungen zur Bearbeitung von Daten lassen sich auf drei verschiedene Arten organisieren:

- Die Ausführung kann von einer oder mehreren Bedingungen abhängig gemacht werden.
- Einzelne Anweisungen oder Anweisungsblöcke sind mehrfach wiederholbar.
- Anweisungsblöcke können als separate Einheiten definiert werden, die von mehreren Teilen des Programms benutzbar sind, was bedeutet, sie bearbeiten je nach Zusammenhang unterschiedliche Daten.

Für diese sieben Punkte hat man in der Programmierung folgende Oberbegriffe eingeführt:

- Eingabe
- Datentypen
- Operationen
- Ausgabe
- bedingte Ausführung
- Schleifen
- Unterprogramme

Diese Liste erhebt jedoch keinen Anspruch auf Vollständigkeit. So bieten heute zahlreiche Compiler die Fähigkeit der objektorientierten Programmierung. Im Gegensatz zu C oder Pascal spielt diese Art der Programmierung in Quick Basic oder Assembler aber keine Rolle, ist also nicht allgemein gültig. Die hier aufgeführten sieben Elemente (siehe nebenstehende Textbox) sind daher als kleinster gemeinsamer Nenner aller Programmiersprachen anzusehen.

Alle Programme (außer Trivialprogrammen) müssen daher unabhängig von der Sprache, in der sie entwickelt sind, mindestens diese sieben Elemente enthalten.

(Frank Riemenschneider/hs/bj)

Grundelement Beschreibung

1	Eingabe	Lesen der Daten von Tastatur, Massenspeicher (Diskette, Festplatte, CD-ROM und ähnliches) oder Schnittstelle (COM, LPT)
2	Datentyp	Konstanten, Variablen und Strukturen (Felder), die numerische Werte, Texte (einzelne Zeichen oder Zeichenketten) sowie Adressen (von Code oder Variablen) enthalten
3	Operationen	Zuweisung von Werten an Variablen, Verknüpfung (arithmetische oder logische Verknüpfungen) und Vergleiche von Ausdrücken
4	Ausgabe	Schreiben von Daten auf Bildschirm, Drucker, Schnittstelle (COM oder LPT) sowie Massenspeicher (Diskette, Festplatte, Tape-Streamer und so weiter)
5	bedingte Ausführung	Ausführung von einzelnen Anweisungen oder Anweisungsblöcken auf Basis des Ergebnisses einer Vergleichsoperation. Hat der Vergleich das Ergebnis »wahr« ergeben, löst dies bestimmte Anweisungen aus, beim Ergebnis »falsch« entsprechend nicht beziehungsweise andere.
6	Schleifen	Wiederholung der Ausführung einer Anweisung oder eines Anweisungsblockes, wobei die Anzahl der Wiederholungen fest vorgegeben ist oder aber sich vom Ergebnis einer Vergleichsoperation abhängig machen läßt (Wiederholung so lange, bis das Ergebnis »wahr« beziehungsweise »falsch« ergibt).
7	Unterprogramme	Hierbei handelt es sich um einen Anweisungsblock, der von mehreren Stellen eines Programms aufrufbar ist. Er erhält in den allermeisten Fällen Daten, die dort zu bearbeiten sind. Ein Sonderfall eines Unterprogramms stellt eine Funktion dar, die das Resultat einer Berechnung an das aufrufende Programm zurückgibt.

Der Weg zum ersten Programm

Der erste Schritt ist oft der wichtigste. Um gute Programme zu schreiben, ist es notwendig, seine Programmierwerkzeuge gut zu kennen. Die folgende Einführung in die Entwicklungsumgebung von Turbo Pascal 6.0, Borland C++, Quick Basic und Turbo-Assembler stellt die wichtigsten Funktionen und Leistungsmerkmale dieser vier Systeme vor.

Wenn Sie beabsichtigen, in mehreren Sprachen parallel zu arbeiten, die von verschiedenen Herstellern angeboten werden (zum Beispiel Pascal oder C von Borland, Microsoft oder Watcom), sollten Sie sich unbedingt auf die Produkte eines einzigen Herstellers beschränken, da diese im allgemeinen eine ähnliche, wenn nicht sogar identische Entwicklungsumgebung aufweisen. Dies bedeutet, daß neben Menüs und Hotkeys (wichtige Tastenkombinationen) auch die Optik (Fenster, Dialogboxen) ähnlich ist, so daß man sich als Programmierer nur an ein Konzept gewöhnen muß.

In unserem Fall trifft dies auf Borland C++ und Turbo Pascal 6.0 zu. Quick Basic weist eine völlig andere Umgebung auf. Der Turbo-Assembler schließlich liegt nur in einer Kommandoversion vor. Das erlaubt Ihnen, den Editor zur Quelltexterzeugung zu verwenden, mit dem Sie am besten vertraut sind.

Arbeiten Sie mit Quick Basic, empfiehlt sich der DOS-Editor Edit, da er zum Quick-Basic-Editor identisch ist. Arbeiten Sie jedoch mit Turbo Pascal oder Borland

C++, sollten Sie die Ihnen vertraute Umgebung auch für die Programmierung der Assembler-Quelltexte nutzen.

■ Turbo Pascal 6.0

Nach dem Start des Compilers mit

TURBO [Enter]

wird die integrierte Entwicklungsumgebung geladen (Bild 1). Sie beinhaltet drei wesentliche Komponenten:

- die Menüleiste am oberen Rand
- den Arbeitsbereich in der Mitte
- die Statuszeile am unteren Rand

Die Menüleiste ist grundsätzlich sichtbar, wenn nicht gerade Ihr Programm abläuft und den Bildschirm für sich in Anspruch nimmt. Um die Menüleiste zu aktivieren, können Sie wahlweise Tastatur oder Maus einsetzen, wobei letztere Methode schneller und einfacher ist. Wenn Sie den gewünschten Menütitel anklicken (den Mauszeiger auf dem zugehörigen Text in der Menüleiste positionieren und die linke Maustaste drücken), klappt augenblicklich das zugehörige Pull-down-Menü

herunter. Um einen Punkt anzuwählen, klicken Sie ihn nun einfach an. Manche Auswahlpunkte führen auf eine weitere Auswahl, ein sogenanntes Untermenü, in dem man den endgültigen Menüpunkt startet.

Mit der Tastatur wird die Menüleiste durch die F10-Taste aktiviert und der erste Menütitel hervorgehoben. Die Tasten [Cursor



Bild 1. Die integrierte Entwicklungsumgebung von Turbo Pascal

links] und [Cursor rechts] erlauben es nunmehr, das gewünschte Menü auszuwählen, indem der invers dargestellte Auswahlbalken nach links oder rechts wandert. Ist das Menü erreicht, drücken Sie [Enter], worauf sich das Pull-down-Menü präsentiert.

Alternativ verwenden Sie dazu auch den sogenannten Shortcut. Hierbei handelt es sich um einen bestimmten Buchstaben, der im Menütext eine andere Farbe als die restlichen Buchstaben aufweist (beispielsweise der Buchstabe E beim Edit-Menü). Der Shortcut ist mit dem sogenannten Hotkey prinzipiell identisch. Ein Hotkey zeichnet sich dadurch aus, daß er sich von jeder beliebigen Stelle im Programmablauf aufrufen läßt. Dabei muß, anders als beim Shortcut, die Menüleiste nicht zuvor mit [F10] aktiviert worden sein. Der Unterschied besteht darin, daß der Hotkey zusätzlich die Alt-Taste erfordert. So ist für den Aufruf des Edit-Menüs die Tastenkombination [Alt-E] zu drücken.

Nachdem ein Pull-down-Menü aktiviert wurde, läßt sich entweder wiederum der Shortcut des gewünschten Auswahlpunktes direkt eingeben oder Sie drücken so lange die Tasten [Cursor auf] oder [Cursor ab], bis der gewünschte Auswahlpunkt hervorgehoben erscheint. Anschließend führen Sie ihn mit [Enter] aus (Bild 2).

Um ein Menü zu verlassen, drücken Sie die Taste [Esc]. Damit gelangen Sie jeweils eine Ebene höher (beispielsweise mit dem ersten [Esc] aus einem Untermenü ins übergeordnete Pull-down-Menü, das zweite [Esc] deaktiviert die Menüleiste).

Verwenden Sie anstelle der Tastatur die Maus, so klicken Sie damit in einen Bildschirmbereich, in dem sich weder die Menüleiste noch ein aktives Pull-down-Menü befindet.

Die hier beschriebenen Methoden weisen den Nachteil auf, daß sie zur Auswahl eines Menüpunktes mindestens zwei aufeinanderfolgende Tastendrucke benötigen. Die Entwickler von Turbo Pascal 6.0 haben daher häufig benötigten Menüpunkten eigene Hotkeys zugeordnet, welche über den Auswahlpunkt mit nur einem Tastendruck oder einer Tastenkombination aufzurufen sind (Tabelle 1). Die Schreibweise File/Open stellt beispielsweise den Auswahlpunkt Open des File-Menüs dar,

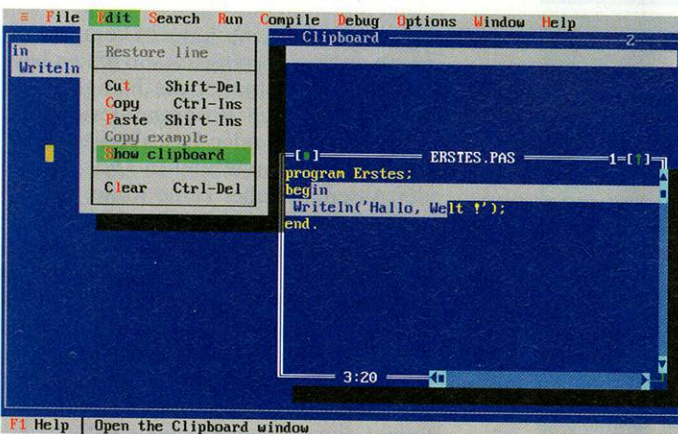


Bild 2. Pull-down-Menüs erleichtern das Zurechtfinden in der Oberfläche.

Menübefehl	Bedeutung	Hotkey
System	öffnet das Pull-down-Menü <i>System</i>	[Alt-Leertaste]
File	öffnet das Pull-down-Menü <i>File</i>	[Alt-F]
File/Save	speichert Datei, die sich im aktiven Fenster befindet	[F2]
File/Open	öffnet ein Dialogfenster zum Laden einer Datei	[F3]
File/Exit	beendet Turbo Pascal 6.0	[Alt-X]
Edit	öffnet das Pull-down-Menü <i>Edit</i>	[Alt-E]
Edit/Cut	verschiebt den markierten Text in die Zwischenablage	[Umschalt-Entf]
Edit/Copy	kopiert den markierten Text in die Zwischenablage	[Strg-Einf]
Edit/Paste	fügt den Text aus der Zwischenablage in das aktive Fenster ein	[Umschalt-Einf]
Edit/Clear	löscht den markierten Text	[Strg-Entf]
Search	öffnet das Pull-down-Menü <i>Search</i>	[Alt-S]
Run	öffnet das Pull-down-Menü <i>Run</i>	[Alt-R]
Run/Run	startet das Programm im aktiven Fenster	[Strg-F9]
Run/Program Reset	setzt das Programm im aktiven Fenster zurück	[Strg-F2]
Run/Go To Cursor	führt das Programm bis zur Cursor-Position aus	[F4]
Run/Trace Into	startet das Programm im Debugger-Modus und führt Unterprogramme im Einzelschrittmodus aus	[F7]
Run/Step Over	startet das Programm und führt Unterprogramme als einzelnen Befehl aus	[F8]
Compile	ruft das Pull-down-Menü <i>Compile</i> auf	[Alt-C]
Compile/Compile	kompiliert den Quelltext im aktiven Fenster	[Alt-F9]
Compile/Make	kompiliert und linkt das Programm im aktiven Fenster	[F9]
Debug	ruft das Pull-down-Menü <i>Debug</i> auf	[Alt-D]
Debug/Evaluate/Modify	wertet einen Ausdruck aus	[Strg-F4]
Debug/Watches/Add Watch	überwacht den Inhalt einer Variablen	[Strg-F7]
Debug/Toggle Breakpoint	setzt oder löscht wechselweise einen Haltepunkt	[Strg-F8]
Options	öffnet das Pull-down-Menü <i>Options</i>	[Alt-O]
Window	öffnet das Pull-down-Menü <i>Window</i>	[Alt-W]
Window/Zahl	aktiviert das Fenster mit der Nummer x	[Alt-x]
Window/Size/Move	ändert Position und/oder Größe des aktiven Fensters	[Strg-F5]
Window/Zoom	wechselweise Verkleinerung/Vergrößerung des aktuellen Fensters	[F5]
Window/Next	aktiviert das nächste Fenster in der Fensterliste	[F6]
Window/Previous	aktiviert das vorhergehende Fenster in der Fensterliste	[Umschalt-F6]
Window/Close	schließt das aktive Fenster	[Alt-F3]
Window/Call Stack	zeigt Aufrufe-Stack an	[Strg-F3]
Window/User Screen	zeigt den Bildschirm des ausgeführten Programms an	[Alt-F5]
Window/List	zeigt die Fensterliste an	[Alt-O]
Help	öffnet das Pull-down-Menü <i>Help</i>	[Alt-H]
Help/Contents	ruft kontextsensitive Hilfefunktion auf	[F1]
Help/Help on Help	ruft Hilfe für Hilfefunktion auf	[F1] [F1]
Help/Index	öffnet den Hilfe-Index	[Umschalt-F1]
Help/Previous Topic	öffnet den zuletzt aktiven Hilfebildschirm	[Alt-F1]
Help/Topic search	aktiviert die Syntaxhilfe im Editor	[Strg-F1]

Tabelle 1. Die Hotkeys von Turbo Pascal 6.0

Run/Step Over den Auswahlpunkt *Step Over* des Run-Menüs.

● Arbeiten im Fenster

Die Ausführung fast aller Funktionen erfolgt in Bildschirmfenstern, so auch die Eingabe des Quelltextes. Ein Fenster kann man verschieben, verkleinern oder vergrößern, schließen, öffnen und durch andere Bildschirmelemente zeitweise verdecken. Sie können theoretisch beliebig viele Fenster öffnen und damit die Quelltexte mehrerer Programme parallel bearbeiten. In der Praxis ist die Anzahl der

Fenster jedoch durch den verfügbaren Arbeitsspeicher begrenzt. Alle Funktionen des Compilers und Eingaben beziehen sich stets auf das zur Zeit aktive Fenster, also immer nur ein einziges. Wenn Sie jedoch dieselbe Quelltextdatei in mehrere Fenster geladen haben, wirken sich die Aktionen des aktiven Fensters auch automatisch auf die nicht aktiven Fenster aus, in denen sich derselbe Quelltext befindet. Ein Fenster enthält sechs typische Elemente (Bild 1):

- einen Fenstertitel
- ein Schließfeld in der linken oberen Ecke

- zwei Bildrolleiten am unteren und rechten Rand, mit denen man den Fensterinhalt horizontal beziehungsweise vertikal verschieben kann (Scrolling)
- ein Feld zur Änderung der Größe in der rechten unteren Ecke
- ein Zoom-Feld in der rechten oberen Ecke, mit dem Sie das Fenster wechselweise vergrößern oder verkleinern
- eine Fensternummer links neben dem Zoom-Feld, welche die Position des Fensters in der Liste aller geöffneten Fenster angibt (in Bild 1 handelt es sich also um das erste Fenster).

Ausschließlich ein Edit-Fenster enthält zusätzlich in der linken unteren Ecke die aktuelle Position des Cursors in Spalten und Zeilen von der linken oberen Ecke gemessen (so bedeutet die Angabe 4:12, daß sich der Cursor augenblicklich in Zeile 4 und Spalte 12 befindet) und links neben diesen Zahlen einen Stern (*), falls der Quelltext seit dem Laden verändert wurde.

Das Schließfeld (Quadrat zwischen zwei eckigen Klammern) befindet sich in der linken oberen Ecke. Durch Anklicken mit der Maus oder über den Menüpunkt *Window/Close* beziehungsweise [Alt-F3] wird das Fenster geschlossen. Hilfe-Fenster schließen Sie mit der Taste [Esc]. Der Fenstertitel am oberen Rand enthält den Namen und rechts außen die Fensternummer, durch die der Anwender es eindeutig in der Fensterliste identifiziert. Wenn Sie das Fenster zu irgendeinem Zeitpunkt verkleinert haben, so daß es nicht mehr den gesamten Bildschirm einnimmt, können Sie es durch einen sogenannten Doppelklick auf den Titelfeldwechselweise auf maximale Größe (Bildschirmgröße) ausdehnen oder wieder auf die ursprüngliche Größe zurücksetzen.

Als Doppelklick bezeichnet man das zweimalige Drücken der linken Maustaste kurz hintereinander. Damit das Programm einen Doppelklick von zwei aufeinanderfolgenden Einzelklicks unterscheiden kann, muß der Zeitabstand der Klicks hinreichend gering sein. Wenn Sie den Titelfeld einfach anklicken und die linke Maustaste gedrückt halten, verändert sich die Farbe des Fensterrahmens (in der Regel von Weiß zu Grün). Sie können das Fenster nun beliebig verschieben, wobei es am Mauszeiger »zu kleben« scheint. Haben Sie die gewünschte Bildschirmposition erreicht, lassen Sie die Maustaste los, wodurch das Fenster an der entsprechenden Stelle fixiert wird.

Tastaturbenutzer können den Menüpunkt *Size/Move* des Window-Menüs oder den Hotkey [Strg-F5] verwenden. Hierbei verschieben Cursor-Tasten das Fenster. Drücken Sie [Cursor auf], bewegt sich das Fenster um eine Spalte nach oben. Ist die gewünschte Position erreicht, beenden Sie den Vorgang mit [Esc].

Jedem Turbo-Pascal-Fenster wird eine fortlaufende Fensternummer zugeordnet, die am oberen rechten Rand ersichtlich ist.

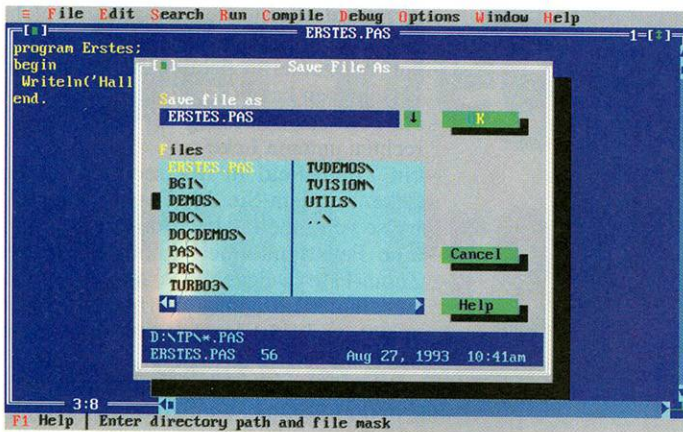


Bild 3. Dialogfenster unterstützen die Auswahl. Zusätzlich erfolgt ein Hinweis auf die erwartete Eingabe.

Mit [Alt-0] lassen Sie sich die Fensterliste anzeigen, in welcher alle aktuell geöffneten Fenster mit den dazugehörigen Nummern erscheinen.

Eine Schnellwahl gestattet [Alt-n], wobei Sie direkt das Fenster mit der Nummer n aktivieren und es damit automatisch in den Vordergrund bringen. Dieses Vorgehen ist immer dann sinnvoll, wenn ein Fenster verdeckt ist und sich somit nicht durch einen Mausklick in das Fensterinnere aktivieren lässt.

Um die Größe eines Fensters stufenlos zu verändern, klicken Sie das Feld in der rechten unteren Fensterecke an (dort, wo die Umrahmung nur einfach gezeichnet ist). Bei gedrückt gehaltener Maustaste ist das Fenster nun vergrößerbar. Nach dem Loslassen der Maustaste bleibt die eingestellte Größe fixiert. Falls das Fenster kleiner als der Gesamtbildschirm ist, kann man es über den Titelbalken oder das Zoom-Feld zwischen maximaler Ausdehnung (Bildschirmgröße) und der eingestellten Größe hin und her wechseln lassen.

Statt des Größe-Feldes können Sie auch den Menüpunkt *Size/Move* des Window-Menüs oder den Hotkey [Strg-F5] verwenden. Drücken Sie beispielsweise [Umschalt-Cursor rechts], verschiebt sich der rechte Fensterrand um eine Spalte nach rechts, das Fenster wird somit um eine Spalte breiter. Haben Sie die gewünschte Größe erreicht, schließen Sie den Vorgang mit [Esc] ab.

Das Zoom-Feld befindet sich in der oberen rechten Fensterecke. Falls das Fenster kleiner als der Gesamtbildschirm ist, sehen Sie hier einen Pfeil nach oben. Wenn Sie diesen anklicken, öffnet sich das Fenster auf seine maximale Größe, und der einfache Pfeil wechselt in einen Doppelpfeil mit zwei Spitzen nach oben und unten. Klicken Sie das Symbol erneut an, erhält das Fenster wieder seine ursprüngliche Größe. Das Zoom-Feld besitzt dieselbe Funktion wie der Doppelklick auf den Titelbalken. Über die Tastatur können Sie den Menüpunkt *Window/Zoom* oder den Hotkey [F5] verwenden.

wesentlich größer als der Bildschirm sein, so daß immer nur ein Ausschnitt des Fensters sichtbar ist.

Mit Hilfe der Bildrolleiten läßt sich der angezeigte Fensterausschnitt verschieben. Steht das horizontale Schiebefeld am linken Anschlag, bedeutet dies, daß der am weitesten links liegende Fensteranteil dargestellt wird. Durch Anklicken eines der zwei Pfeile einer Bildrolleiste verschiebt sich der Fensterausschnitt um eine Spalte oder Zeile in Pfeilrichtung. Falls Ihnen diese Methode zu langsam arbeitet, klicken Sie besser das Schiebefeld direkt an und bewegen es, bei gedrückt gehaltener Maustaste, an die gewünschte Position. Nach dem Loslassen der Maustaste sehen Sie augenblicklich den neue Fensterausschnitt. Wollen Sie genau um eine Seite (Ausschnittbreite oder -höhe) rollen, klicken Sie einfach einen schattierten Bereich neben beziehungsweise ober- oder unterhalb des Schiebefeldes an.

● Die Statuszeile

Die Statuszeile finden Sie am unteren Bildschirmrand. Sie hat je nach Kontext unterschiedliche Aufgaben:

- Wenn Sie gerade einen Quelltext im Edit-Fenster bearbeiten, zeigt sie die Hotkeys der wichtigsten Menüpunkte (Bild 1).
- Bei einem heruntergeklappten Pull-down-Menü erscheint eine Erläuterung zu dem augenblicklich hervorgehobenen Menüpunkt (Bild 2).
- Führt der Compiler gerade eine Operation aus (zum Beispiel Speichern eines

Der untere und rechte Fensterrand zeigen zwei sogenannte Bildrolleiten, die jeweils durch zwei Pfeile begrenzt sind. Irrendwo zwischen den Pfeilen steht ein Schiebefeld (das Zeichen mit dem Quadrat in der Mitte). Die Position des Schiebefeldes gibt dabei die Position des angezeigten Fensterinhaltes relativ zum Gesamtfenster an. In der Tat können Fenster

Quelltextes), wird eine Vollzugsmeldung angezeigt (zum Beispiel »saving erstes.pas«).

● Dialogfenster

Zahlreiche Menüpunkte weisen hinter dem Menütex drei Punkte auf (zum Beispiel *Open...* im File-Menü). Dies ist ein Hinweis darauf, daß nach Anwahl des Menüpunktes ein sogenanntes Dialogfenster öffnet, in dem der Benutzer Einstellungen und/oder Eingaben aller Art vornehmen kann (Bild 3). Prinzipiell stehen fünf unterschiedliche Eingabearten zur Auswahl:

- Radioknöpfe
- Markierungsfelder
- Knöpfe
- Texteingabefelder
- Auswahllisten

Das Dialogfenster in Bild 3 dient zur Auswahl eines Quelltextes, um ihn zur Bearbeitung zu laden. Es enthält fünf Eingabevarianten: drei Knöpfe (Ok, Cancel und Help) sowie ein Texteingabefeld (Save File As) und eine Auswahlliste (Files). Zwischen den einzelnen Eingabefeldern schalten Sie mit den Tasten [Tab] oder [Umschalt-Tab] hin und her. Das aktuell aktive Eingabeelement wird farblich hervorgehoben und läßt sich durch [Enter] anwählen.

Viel einfacher ist natürlich das direkte Anklicken mit der Maus. Auch hier lassen sich wieder Shortcuts verwenden.

Jedes Turbo-Pascal-Dialogfenster enthält mindestens die drei Knöpfe Ok, Cancel und Help.

Mit Ok wird die in den übrigen Eingabefeldern (Auswahllisten, Radioknöpfe, Markierungsfelder) getroffene Auswahl umgesetzt.

Cancel verwirft die bislang vorgenommenen Eingaben und schließt das Dialogfenster. Die Esc-Taste führt zum gleichen Ziel.

Durch Help wird ein Hilfe-Fenster zum aktuellen Dialogfenster geöffnet.

● Texteingabefelder und Auswahllisten

Ein Texteingabefeld dient dazu, direkt vom Benutzer eingegebene Texte aufzu-

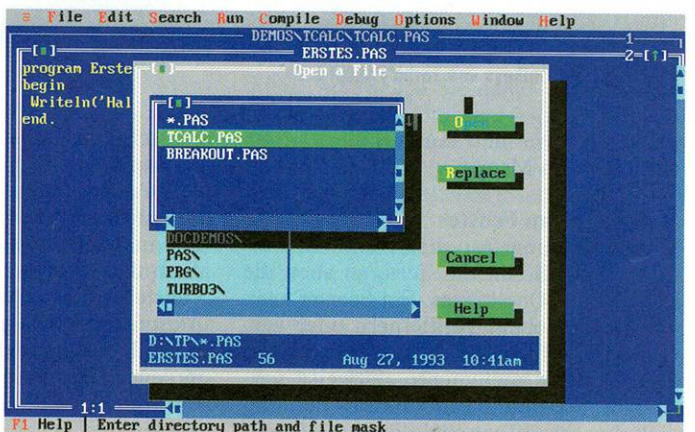


Bild 4. Mit Hilfe von Auswahlbalken und Cursor-Tasten treffen Sie die Auswahl unter den aufgezeichneten Texten.

Menübefehl	Bedeutung	Hotkey
System	öffnet das Pull-down-Menü <i>System</i>	[Alt-Leertaste]
File	öffnet das Pull-down-Menü <i>File</i>	[Alt-F]
File/Save	speichert Datei, die sich im aktiven Fenster befindet	[F2]
File/Open	öffnet ein Dialogfenster zum Laden einer Datei	[F3]
File/Exit	beendet Borland C++	[Alt-X]
Edit	öffnet das Pull-down-Menü <i>Edit</i>	[Alt-E]
Edit/Undo	macht die letzte Aktion rückgängig	[Alt-Rückschritt]
Edit/Cut	verschiebt den markierten Text in die Zwischenablage	[Umschalt-Entf]
Edit/Copy	kopiert den markierten Text in die Zwischenablage	[Strg-Einf]
Edit/Paste	fügt den Text aus der Zwischenablage in das aktive Fenster ein	[Umschalt-Einf]
Edit/Clear	löscht den markierten Text	[Strg-Entf]
Search	öffnet das Pull-down-Menü <i>Search</i>	[Alt-S]
Search/Search again	wiederholt den letzten Suchen- oder Ersetzen-Befehl	[Strg-L]
Search/Previous Error	sucht den vorhergehenden Fehler	[Alt-F7]
Search/Next Error	sucht den nächsten Fehler	[Alt-F8]
Run	öffnet das Pull-down-Menü <i>Run</i>	[Alt-R]
Run/Run	startet das Programm im aktiven Fenster	[Strg-F9]
Run/Program Reset	setzt das Programm im aktiven Fenster zurück	[Strg-F2]
Run/Go To Cursor	führt das Programm bis zur Cursor-Position aus	[F4]
Run/Trace Into	startet das Programm im Debugger-Modus und führt Unterprogramme im Einzelschrittmodus aus	[F7]
Run/Step Over	startet das Programm und führt Unterprogramme als einzelne Befehl aus	[F8]
Compile	ruft das Pull-down-Menü <i>Compile</i> auf	[Alt-C]
Compile/Compile to OBJ	kompiliert den Quelltext im aktiven Fenster in eine Obj-Datei	[Alt-F9]
Compile/Make EXE	kompiliert und linkt das Programm im aktiven Fenster in eine Exe-Datei	[F9]
Debug	ruft das Pull-down-Menü <i>Debug</i> auf	[Alt-D]
Debug/Inspect	öffnet ein Inspektor-Fenster	[Alt-F4]
Debug/Evaluate/Modify	wertet einen Ausdruck aus	[Strg-F4]
Debug/Watches/Add Watch	überwacht den Inhalt einer Variablen	[Strg-F7]
Debug/Call Stack	zeigt Aufrufe-Stack an	[Strg-F3]
Debug/Toggle Breakpoint	setzt oder löscht wechselweise einen Breakpunkt	[Strg-F8]
Project	öffnet das Pull-down-Menü <i>Project</i>	[Alt-P]
Options	öffnet das Pull-down-Menü <i>Options</i>	[Alt-O]
Window	öffnet das Pull-down-Menü <i>Window</i>	[Alt-W]
Window/Zahl	aktiviert das Fenster mit der entsprechenden Nummer	[Alt]+?
Window/Size/Move	ändert Position und/oder Größe des aktiven Fensters	[Strg-F5]
Window/Zoom	wechselweise Verkleinerung/Vergrößerung des aktuellen Fensters	[F5]
Window/Next	aktiviert das nächste Fenster in der Fensterliste	[F6]
Window/Previous	aktiviert das vorhergehende Fenster in der Fensterliste	[Umschalt-F6]
Window/Close	schließt das aktive Fenster	[Alt-F3]
Window/User Screen	zeigt den Bildschirm des ausgeführten Programms an	[Alt-F5]
Window/List	zeigt die Fensterliste an	[Alt-O]
Help	öffnet das Pull-down-Menü <i>Help</i>	[Alt-H]
Help/Contents	ruft kontextsensitive Hilfefunktion auf	[F1]
Help/Help on Help	ruft Hilfe für Hilfefunktion auf	[F1] [F1]
Help/Index	öffnet den Hilfe-Index	[Umschalt-F1]
Help/Previous Topic	öffnet den zuletzt aktiven Hilfebildschirm	[Alt-F1]
Help/Topic search	aktiviert die Syntaxhilfe im Editor	[Strg-F1]

Tabelle 2. Die Hotkeys von Borland C++

nehmen. Der Text kann wie gewohnt mit den Cursor-Tasten, [Einf], [Entf], [Pos1], [Ende] und [Rückschritt] editiert werden. Ist die Eingabe länger als das Texteingabefeld, rollt der Text durch das Feld, und an

beiden Enden erscheint ein Pfeil, mit dem sich, ähnlich wie bei den Schiebefeldern, der Text mit Hilfe der Maus durch den sichtbaren Bereich rollen läßt. Manche Texteingabefelder enthalten eine Auf-

zeichnungsliste, in der vorhergehende Eingaben dieses Feldes gespeichert werden. Dies ist bei in unregelmäßigen Abständen sich wiederholenden Eingaben praktisch, da man somit nicht denselben Text mehr als einmal eintippen muß. Ein Feld mit zugehöriger Aufzeichnungsliste wird durch einen Abwärtspfeil rechts neben dem Eingabefeld gekennzeichnet. Wenn Sie den Pfeil anklicken oder [Cursor ab] drücken, wird die Aufzeichnungsliste angezeigt (Bild 4). Hier können Sie mit Hilfe eines Auswahlbalkens, auf- und abwärts zu steuern über die Cursor-Tasten, einen Eintrag auswählen und diesen mit [Enter] bestätigen. Sie können statt dessen auch einen Doppelklick mit der Maus ausführen. In beiden Fällen wird die Liste geschlossen und die Auswahl in das Texteingabefeld übernommen. Möchten Sie keine der dort angezeigten potentiellen Eingaben übernehmen, so klicken Sie das Schließfeld des Fensters an oder drücken [Esc].

Eine Auswahlliste kann aus einer oder mehreren Spalten bestehen, die jeweils theoretisch beliebig viele Auswahlvarianten anbieten. Wie bei den Aufzeichnungslisten kann man den Auswahlbalken innerhalb einer Spalte mit den Tasten [Cursor auf] und [Cursor ab] steuern und die Auswahl mit [Enter] bestätigen. Zwischen zwei Spalten kann man mit [Cursor links] oder [Cursor rechts] wechseln. Einfacher ist wie immer natürlich der direkte Doppelklick auf einen Listenauswahlpunkt. Mit den waagerechten und/oder senkrechten Bildrolleiten lassen sich die Auswahlpunkte beziehungsweise Spalten mit der Maus durchwandern. Ist innerhalb der Liste ein blinkender Unterstrich sichtbar, bedeutet dies, daß eine Hilfefunktion für die Suche von Begriffen aktiviert ist. In diesem Fall können Sie den ersten Buchstaben des gesuchten Begriffs eintippen, worauf Turbo Pascal automatisch den Auswahlbalken auf dem ersten Auswahlpunkt plaziert, der mit diesem Buchstaben beginnt.

● Radioknöpfe und Markierungsfelder

Hat der Benutzer die Auswahl zwischen einigen wenigen, immer identischen Einstellungsfunktionen, werden diese in der Regel wegen der besseren Übersichtlichkeit statt in Listenform in Gestalt von Radioknopf-Auswahlfeldern beziehungsweise Markierungsfeldern dargestellt (Bild 5). Der prinzipielle Unterschied zwischen beiden Verfahren besteht darin, daß Radioknopf-Auswahlfelder sich gegenseitig ausschließende Auswahlmöglichkeiten anbieten, das heißt, immer nur eine Auswahl getroffen werden kann, während mehrere Markierungsfelder gleichzeitig angewählt werden können, da sie sich nicht gegenseitig ausschließen. Sehen Sie sich hierzu das Dialogfenster in Bild 5 an, das die Optionen für eine Suchen-und-Ersetzen-Funktion festlegt. Die Einstellung der Suchrichtung im Quelltext (vorwärts oder rückwärts) ist durch das

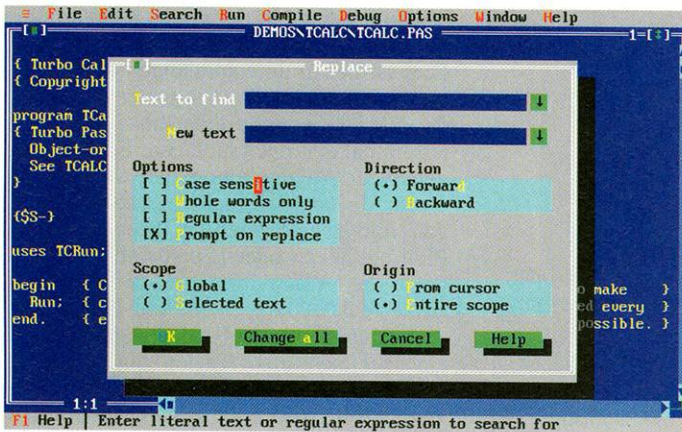


Bild 5. Sogenannte Radioknöpfe symbolisieren feste Einstellungen.

Radioknopf-Auswahlfeld *Direction* realisiert, in dem man die Suchrichtung *Forward* oder *Backward* einstellen kann. Beide Suchrichtungen schließen einander natürlich aus. Dagegen ist das Eingabefeld *Options* als Markierungsfeld umgesetzt, da man zum Beispiel die Auswahlkriterien *Case Sensitive* (Groß- und Kleinschreibung beachten) und *Whole Words Only* (nur nach ganzen Wörtern suchen) beliebig kombinieren kann.

Ein Markierungsfeld wird wechselweise aktiviert und deaktiviert, indem man zunächst die Gruppe mit [Tab] oder [Umschalt-Tab] anwählt, mit den Cursor-Tasten das Feld ansteuert und schließlich die Leerschritt-Taste drückt. Ob das Feld aktiv ist, erkennen Sie an dem x zwischen den rechteckigen Klammern ([x] = ausgewählt, [] = nicht ausgewählt). Einfacher ist das Anklicken mit der Maus oder die Eingabe eines Shortcuts durch [Alt-n], wenn n der hervorgehobene Buchstabe ist. Bei den Radioknopf-Auswahlfeldern kann man die Auswahl direkt mit den Cursor-Tasten treffen, da durch die Hervorhebung eines Auswahlpunktes automatisch alle anderen Auswahlpunkte deaktiviert werden.

Borland C++

Um den C++-Compiler von Borland aufzurufen, tippen Sie in der Kommandozeile `TC [Return]` ein. Da das Produkt aus dem gleichen Hause wie Turbo Pascal kommt, ist die Bedienungsführung identisch (Bild 6). Dies bedeutet, daß alles zu Fenstern, Menüs, Dialogfenstern und so weiter auch für Borland C++ gilt. Der einzige Unterschied besteht darin, daß der Compiler natürlich eine etwas andere Menüstruktur beinhaltet. Tabelle 2 zeigt die Hotkeys von Borland C++.

wird, so daß zumindest das Preis-Leistungs-Verhältnis ungeschlagen sein dürfte.

Der Interpreter wird aus der Kommandozeile mit dem Befehl

```
QBASIC [Return]
```

gestartet.

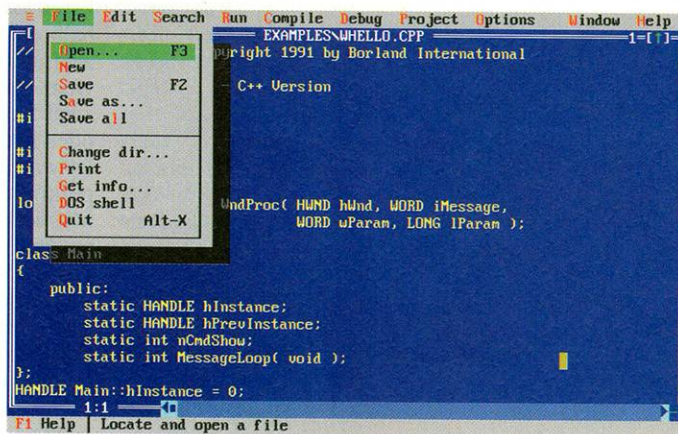


Bild 6. Borland C++ weist die gleiche Benutzeroberfläche wie Turbo Pascal auf, was Programmierern beider Sprachen zugute kommt.

Der Willkommensbildschirm (Bild 7) läßt die Wahl zwischen sofortigem Beginn der Arbeit [Esc] oder einer – spätestens beim zweitenmal todlangweiligen – Einführung (Enter), die aber auch beim ersten Aufruf kaum weiterhilft, da sie auf die praktischen Probleme nicht eingeht. Die integrierte Entwicklungsumgebung von Quick Basic arbeitet wie die von C++ und Turbo Pascal menügesteuert. Die Menüzeile wird mit [Alt] aktiviert, sonst gibt es wie bei den Borland-Compilern für alle Menüs und

Quick Basic

Der Quick-Basic-Interpreter wird als Teil von MS-DOS beziehungsweise PC-DOS (IBM) mitgeliefert. Die Bedienungsführung erinnert an die Borland-Produkte, ist jedoch längst nicht so ausgereift und komplett. Allerdings muß man auch bedenken, daß Quick Basic kostenlos mitgeliefert

zahlreiche Menüpunkte Shortcuts (Bild 8, Tabelle 3). Natürlich läßt sich auch hier der direkte Anklickweg mit der Maus beschreiten, oder Sie wählen wieder die Menüpunkte mit den Cursor-Tasten an und bestätigen anschließend mit [Enter].

Quick Basic zeigt die Shortcuts übrigens grundsätzlich nur auf Anforderung an. Es ist also zunächst einmal die Alt-Taste zu drücken, bevor die Shortcuts in der Menüleiste hervorgehoben werden.

Die Fenstersteuerung ist nicht sehr komfortabel. So läßt sich immer nur ein Quelltext in das einzig existierende Editor-Fenster laden. Dieses ist demzufolge auch nicht verschiebbar oder schließbar. Es läßt sich lediglich verkleinern, indem man die untere Fensterrande anklickt und bei gedrückt gehaltener Maustaste nach oben oder unten verschiebt.

Der freie Platz unterhalb des Editor-Fensters wird dann von einem Fenster mit dem Namen Direkt eingenommen, in dem man Basic-Befehle direkt ausführen kann, ohne sie in den Programmquelltext zu integrieren. Die Eingabe

```
print "HALLO"
```

in diesem Fenster würde sofort auf dem Ausgabebildschirm den Text »HALLO« ausgeben.

Hier zeigt sich der (einzige) Vorteil des Interpreters gegenüber einem Compiler: Da die Übersetzung zur Laufzeit erfolgt, kann ein Basic-Befehl auch ohne Exe-Datei ausgeführt werden.

Die Schiebeleisten und das Zoom-Feld in der rechten oberen Fensterecke funktionieren im Turbo-Pascal-Abschnitt wie beschrieben, während der Titelfeld ausschließlich zur Anzeige dient. Das Fenster läßt sich hiermit nicht verschieben. Mit dem Menüpunkt *Ansicht/Aufteilen* kann man das Editor-

Fenster in zwei Teile aufsplitten. Allerdings ist es nicht erlaubt, in den zweiten Teil einen anderen Quelltext zu laden als

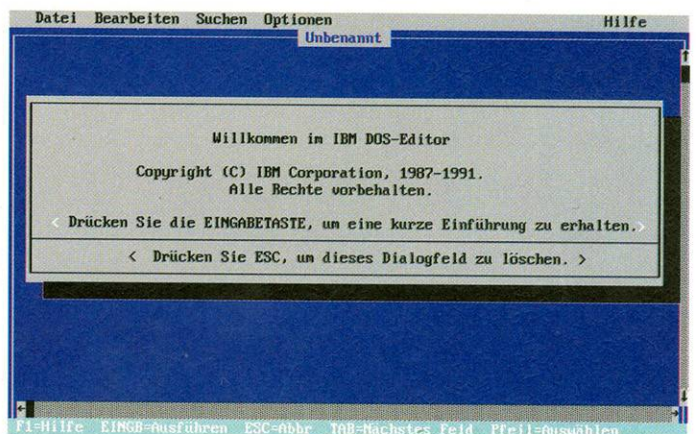


Bild 7. Der DOS-Editor heißt Sie willkommen und stellt eine kurze Einführung zur Verfügung.

Menübefehl	Bedeutung	Hotkey
Datei	öffnet das Pull-down-Menü <i>Datei</i>	[Alt-D]
Bearbeiten	öffnet das Pull-down-Menü <i>Bearbeiten</i>	[Alt-B]
Bearbeiten/Ausschneiden	verschiebt den markierten Text in die Zwischenablage	[Umschalt-Entf]
Bearbeiten/Kopieren	kopiert den markierten Text in die Zwischenablage	[Strg-Einf]
Bearbeiten/Einfügen	fügt den Text aus der Zwischenablage in das aktive Fenster ein	[Umschalt-Einf]
Bearbeiten/Löschen	löscht den markierten Text	[Entf]
Ansicht	öffnet das Pull-down-Menü <i>Ansicht</i>	[Alt-A]
Ansicht/SUBS	zeigt geladene Unterprogramme oder Funktionen an	[F2]
Ansicht/Ausgabebildschirm	zeigt den Ausgabebildschirm des ausgeführten Programms an	[F4]
Suchen	öffnet das Pull-down-Menü <i>Suchen</i>	[Alt-S]
Suchen/Weitersuchen	sucht die nächste Stelle des mit Suchen/Suchen definierten Suchbegriffs	[F3]
Ausführen	öffnet das Pull-down-Menü <i>Ausführen</i>	[Alt-F]
Ausführen/Start	startet das Programm	[Umschalt-F5]
Ausführen/Fortsetzen	setzt Programmausführung nach Unterbrechung fort	[F5]
Debug	ruft das Pull-down-Menü <i>Debug</i> auf	[Alt-U]
Debug/Einzelschritt	führt die nächste Programmanweisung im Einzelschrittmodus aus	[F8]
Debug/Prozedurschritt	führt die nächste Programmanweisung aus, wobei Prozeduraufrufe als einzelne Anweisung angesehen werden	[F10]
Debug/Haltepunkt Ein/Aus	setzt oder löscht wechselweise einen Breakpunkt	[F9]
Optionen	öffnet das Pull-down-Menü <i>Optionen</i>	[Alt-O]
Hilfe	öffnet das Pull-down-Menü <i>Hilfe</i>	[Alt-H]
Hilfe/Thema	aktiviert die Syntaxhilfe im Editor	[F1]
Hilfe/Verwendung von Hilfe	ruft Hilfe für Hilfefunktion auf	[Umschalt-F1]

Tabelle 3. Die Hotkeys von Quick Basic

in den ersten. Der einzige Sinn dieser Funktion besteht darin, daß man zwei unterschiedliche Stellen des Quelltextes parallel bearbeiten kann, ohne ständig mit den Schieberegeln operieren zu müssen. Durch erneute Anwahl desselben Menüpunktes vereinen sich die beiden Fenster-teile wieder.

Dialogfenster

Auch Quick Basic enthält zahlreiche Dialogfenster, auf die alle Menüpunkte führen, die mit drei Punkten enden. Bild 9 zeigt ein Beispiel für ein Dialogfenster der Suchen-und-Ersetzen-Funktion. Allgemein kann man sagen, daß die Quick-Basic-

Funktionen wesentlich einfacher gehalten sind als die Borland-Pendants. So vermißt man bei den Texteingabefeldern die Aufzeichnungslisten sowie die Pfeile zum Durchrollen eines langen Eingabetextes und beim Suchen und Ersetzen die Parametereinstellung zur Vorwärts- oder Rückwärtssuche.

Wie gewohnt springt man mit [Tab] oder [Umschalt-Tab] zwischen den Eingabefeldern hin und her. Die Shortcuts der einzelnen Felder werden nur dann farblich herausgestellt, wenn man einmal die Alt-Taste drückt, nachdem das Dialogfenster geöffnet wurde. Die Auswahl in Listen oder Selektion von Markierungsfel-

dern ist zum beschriebenen Vorgehen identisch.

Wer schon einmal mit Turbo Pascal oder Borland C++ gearbeitet hat, kommt, insgesamt gesehen, sehr einfach mit Quick Basic zurecht.

Turbo-Assembler

In aller Kürze läßt sich die Entwicklungsumgebung des Turbo-Assemblers abhandeln. Sie existiert nämlich erst gar nicht, da es sich um einen reinen Kommandozeilen-Assembler handelt, dem Sie beim Aufruf die Quelltext-Datei als Parameter übergeben. Wenn Sie als Beispiel ein Programm mit dem Namen »demos.asm« geschrieben haben, würde die Assemblierung durch den Kommandozeilenbefehl

TASM DEMO <Return>

gestartet. Sie können sich daher zur Quelltexterzeugung eines Editors Ihrer Wahl bedienen. Besitzen Sie Turbo Pascal oder C++, ist der dort integrierte Editor natürlich erste Wahl. Sie können natürlich auch noch einen Schritt weiter gehen und einen professionellen Texteditor wie Multiedit oder Boxer (beide sind auch als Shareware erhältlich) einsetzen. Sie erlauben es sogar, von der Oberfläche aus den Compiler zu starten und unterstützen eine Fehleranzeige bei Compiler-Abbrüchen.

Sie sind in jedem Fall reinen Textverarbeitungsprogrammen wie beispielsweise MS-Word vorzuziehen. Diese beinhalten die Gefahr, daß Sie Ihr Programm irrtümlich mit Formatanweisungen speichern. Der Turbo-Assembler weiß aber damit nichts anzufangen und wird mit einer Fehlermeldung abbrechen. Besitzen Sie weder einen Editor noch eines der Borland-Produkte, nehmen Sie eben mit dem DOS-Editor Edit vorlieb. Dieser ist übrigens mit den gleichen Funktionen ausgestattet wie Quick Basic. In der Tat wird durch den Befehl *Edit* Quick Basic mit dem Parameter */Editor* gestartet, der sämtliche Menüfunktionen bezüglich des Interpreters unterdrückt.

(Frank Riemenschneider/hs/bj)

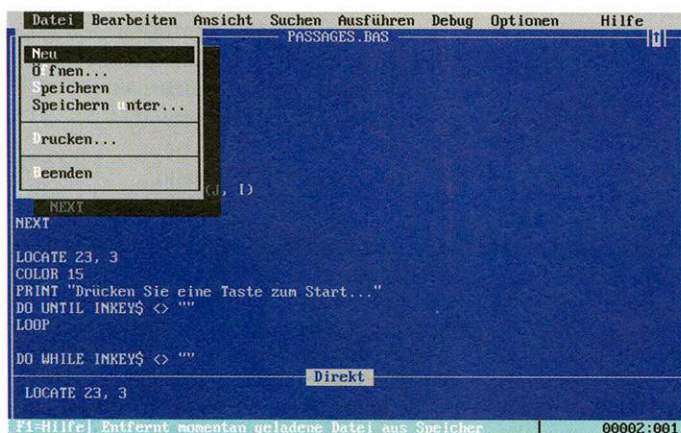


Bild 8. Auch der DOS-Editor erlaubt die Auswahl über Pull-down-Menüs.

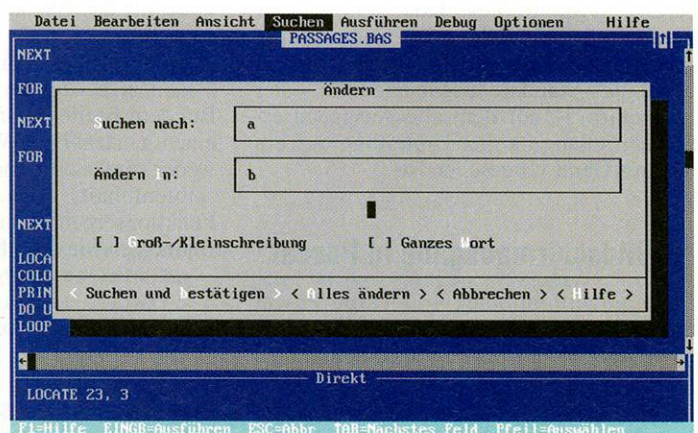


Bild 9. Ein Beispiel für ein Dialogfenster der Suchen-und-Ersetzen-Funktion

Basiselement der Programmierung

Obwohl es nahezu unbegrenzt viele gangbare Wege gibt, ein Programm zu entwickeln, besteht dies zumindest aus sieben wichtigen Komponenten: aus Ein- und Ausgabe von Daten, Datentypen, Operationen, Anweisungen für die bedingte Ausführung, Schleifen und Unterprogrammen. Lernen Sie mit uns die schrittweise Umsetzung des Elementes »Ausgabe« in den Sprachen Pascal, C, Basic und Assembler kennen.

Jedes Programm besitzt eine definierte Mindeststrahlenstruktur, in die die eigentlichen Anweisungen eingebunden sind (Bild 1). In der Praxis ist dieser einfache Grundaufbau durch die Integration von Bibliotheken, die Definition von Variablen, Strukturen, Konstanten sowie von Unterprogrammen und Funktionen wesentlich erweitert. Es ist jedoch durchaus denkbar, selbst mit den Minimalstrukturen ein funktionsfähiges Programm zu erzeugen.

Wie Sie sicher bereits erkannt haben, ist der Programmieraufwand in Assembler wesentlich größer als in einer Hochsprache. Dies liegt jedoch nicht daran, daß ein Hochsprachenprogramm die Anweisungen zur Speichersegmentierung oder Beendigung eines DOS-Programms (Interrupt 21h) nicht benötigt. Bei Hochsprachen, wie beispielsweise Pascal, werden sie vom Compiler automatisch in die Exe-Datei eingebunden, ohne daß der Programmierer etwas davon merkt.

Ein Programm macht grundsätzlich nichts anderes, als Daten oder Anweisungen einzulesen, sie zu verarbeiten und die Ergebnisse wieder auszugeben. Betrachten wir deshalb gleich die Ausgabe.

Hierunter versteht man jede Information, die ein Programm nach »außen« weiterleitet. Das Ausgabemedium selbst (Bildschirm, Diskette, Festplatte, serielle oder parallele Schnittstelle) spielt dabei prinzipiell keine Rolle. Vielmehr existiert ein sogenanntes Standardausgabemedium – der Bildschirm –, auf dem alle Ausgaben erfolgen, solange keine Umleitung auf ein anderes Gerät vorgesehen ist.

Bildschirmausgabe in Pascal

Die Laufzeitbibliothek von Turbo Pascal 6.0 enthält zur Ausgabe die Prozeduren *write* und *writeln*. Sie unterscheiden sich darin, daß nach *writeln* die nächste Ausgabe wieder am Anfang der folgenden Zeile beginnt. Bei *write* bleibt der Cursor unmittelbar nach der Ausgabe stehen.

Die Syntax für beide Prozeduren lautet:

```
writeln(Element1, Element2, Element3, ..., Element N)
oder
```

```
write(Element1, Element2, Element3, ..., Element N)
```

Bei »Element« kann es sich um einen numerischen Wert, eine Zeichenkette (in das Zeichen »« eingeschlossen), einen Variableninhalt, eine Konstante oder das Ergebnis einer Funktion handeln. Mehrere aufeinanderfolgende Ausgaben werden nicht automatisch durch Leerzeichen oder andere Trennzeichen differenziert, sondern die Trennung ist manuell vorzunehmen.

Hierfür zwei Beispiele:

```
write('A','B','C');
```

ergibt die Ausgabe

```
ABC
```

während

```
write('A',' ','B',' ',' ','C');
```

die Ausgabe

```
A,B,C
```

hervorrufen.

Einfacher wäre selbstverständlich

```
write('A,B,C');
```

gewesen. Das Semikolon am Zeilenende ist übrigens der typische Zeilenabschluß eines Turbo-Pascal-Befehls.

Durch einen zusätzlichen Parameter für jedes auszugebende Element läßt sich die minimale Feldbreite (Anzahl der Zeichen), die *write* für das Element ausgeben soll, definieren.

```
write(Element1:Breite1, ..., Element N:Breite N)
```

Bei dem Breite-Parameter muß es sich um einen ganzzahligen Wert handeln, der entweder direkt oder indirekt über einen Variableninhalt, eine Konstante oder ein Funktionsergebnis anzugeben ist.

Betrachten Sie dazu folgendes Beispiel:

```
writeln('1234567890')
write(1:3,13:3,1245:3);
```

führt zur Ausgabe von

```
1234567890
```

```
1 131245
```

Man erkennt eine rechtsbündige Formatierung. So sind der Zahl 1 zwei Leerzeichen vorangestellt, womit sich eine Gesamtlänge

Turbo Pascal

```
BEGIN
```

```
..... Hier stehen die
      Pascal-Anweisungen ....
```

```
END
```

Borland C++

```
main()
```

```
{ .....
```

```
..... Hier stehen die
      C-Anweisungen...
```

```
.....}
```

Quick Basic

```
.....
```

```
..... Hier stehen die Quick-
      Basic-Anweisungen
```

```
.....
```

```
END
```

Turbo Assembler

```
.MODEL SMALL
```

```
.STACK 100h
```

```
.DATA
```

```
.CODE
```

```
mov ax,@data
```

```
mov ds,ax
```

```
.....
```

```
.....Hier stehen die
      Assembler-Anweisungen
```

```
.....
```

```
mov ah,4Ch
```

```
int 21h
```

```
END
```

Bild 1. Rumpfprogramme für Turbo Pascal, Borland C++, Quick Basic und Turbo Assembler

ge von drei Zeichen ergibt. Interessant ist, daß Ausgaben, die länger als die eingestellte Breite sind, nicht abgeschnitten, sondern in voller Breite ausgegeben werden. Die Ausgabe der Zahl 1245 ist also unbeschnitten und nicht, wie dies durch die Breite versucht wurde einzustellen, nur dreistellig. Man darf den Parameter daher wirklich nur als Minimalbreite interpretieren.

Dezimalbrüche erscheinen im Exponentialformat und linksbündig. Hier läßt sich ebenfalls die Breite bestimmen, welche in diesem Fall aber genau einzuhalten ist. Bei zu großen Nachkommastellen wird auf die letzte ausgegebene Dezimalstelle gerundet und der Rest abgeschnitten:

	Ausgabe:
	12345678901234567890
write(19.524);	1.9524000000E+01
write(19.524:20);	1.9524000000E+01
write(19.524:10);	1.952E+01
write(-19.524:10);	-1.952E+01
write(19.524:8);	2.0E+01
write(19.524:6);	2.0E+01

Hierbei fallen zwei Dinge ins Auge. Zum einen beginnen positive Zahlen jeweils mit einem Leerzeichen, denn Turbo Pascal reserviert immer eine Stelle für das negative Vorzeichen. Zum anderen wird jede Dezimalzahl mit mindestens einer Nachkommastelle in der Mantisse (die Zahl vor E) ausgegeben, so daß Breitenangaben, die kleiner als 8 sind, ignoriert und auf acht Zei-

chen aufgerundet werden. Breiten, die größer als der Standardwert 15 ausfallen, führen nicht zur Verlängerung des Nachkommanteils der Mantisse (1.9524000000000000 anstelle von 1.9524000000), sondern zum Auffüllen der überflüssigen Stellen vor der Zahl mit Leerzeichen.

Bei Dezimalzahlen läßt sich zusätzlich ein zweiter Parameter angeben, wenn die Ausgabe im Fixkommaformat erfolgen soll.

```
write(Element1:Breite1:Nachkomma, ..., Element N:Breite N:Nachkomma N)
```

»write« arbeitet in diesem Fall wie bei den ganzen Zahlen rechtsbündig, wie folgende Beispiele belegen:

```
12345678901234
write(1965.3006:14:8); 1965.30060000
write(1965.3006:14:3); 1965.301
write(1965.3006:14:0); 1965
write(1965.3006:8:5); 1965.30060
write(1965.3006:3:2); 1965.30
```

Die letzten beiden Beispiele zeigen, daß die Anzahl der Nachkommastellen auf jeden Fall eingehalten wird. Reicht die festgelegte Gesamtbreite nicht mehr aus, führt dies zum linksbündigen Auffüllen mit der entsprechenden Anzahl benötigter Leerzeichen.

Bildschirmausgabe in C

Der C-Befehl für eine formatierte Ausgabe lautet *Printf*. Ihm übergibt der Programmierer als Argumente einen Format-String und die auszugebenden Daten. Die Formatierungsanweisung enthält dabei neben normalen Zeichen, deren Ausgabe unverändert erfolgt, die Konvertierungsanweisungen für die Daten. Diese genügen folgender Syntax:

% [Flags] [Breite] [.Präzision] [F|N|h|l|L] Typ

Alle Konvertierungsanweisungen beginnen mit einem Prozentzeichen, um sie von »normalen« Zeichen zu unterscheiden. Die weiteren Parameter besitzen folgende Bedeutung:

● Typ

Dies ist eine notwendige Angabe, die den Typ des auszugebenden Parameters festlegt (Tabelle 1).

● [Flags]

Über diese Zeichenfolge legen Sie wahlweise Vorzeichen, Dezimalpunkte, führende und folgende Nullen, oktale und hexadezimale Präfixe sowie links- und rechtsbündige Ausgabe fest (Tabelle 2).

● [Breite]

Diese nicht zwingend erforderliche Angabe bestimmt die mindestens auszugebende Zeichenzahl. Sie läßt sich entweder direkt im Format-String oder indirekt über einen Stern

(*) angeben. Im zweiten Fall muß die Breite nach der Formatierungsanweisung direkt vor dem auszugebenden Parameter stehen. Wenn die Ausgabe weniger Zeichen enthält, als durch Breite angegeben ist, wird sie in Abhängigkeit von [Flags] durch Leerzeichen beziehungsweise Nullen aufgefüllt. Enthält sie hingegen mehr Zeichen, ignoriert der Compiler die Breite. In keinem Fall hat dies ein Abschneiden der Ausgabe zur Folge, um [Breite] genügen zu können (Tabelle 3).

● [.Präzision]

Auch sie ist wahlweise verwendbar und beinhaltet, wie viele Zeichen maximal auszugeben sind (bei ganzen Zahlen die Minimal-

zahl der Ziffern). Diese ist entweder direkt im Format-String (von [Breite] durch einen Dezimalpunkt getrennt) oder indirekt über einen Stern (*) anzugeben. Bei letzterer Technik muß die Präzision nach der Formatierungsanweisung direkt vor dem auszugebenden Parameter stehen. Werden sowohl Breite als auch Präzision über den Stern definiert, sind der auszugebenden Zahl, durch ein Komma getrennt, zunächst Breite und dann Präzision voranzustellen (Tabelle 4).

● [F|N|h|l|L]

Hierbei handelt es sich um eine nicht zwingend erforderliche Größenangabe des Parameters. Diese Größenangabe ist nur selten

Beispiele für Ausgabe der ganzen Zahl 1965

Formatstring, zerlegt in	Typ	Flags	Breite	Präzision	Ausgabe
PRINTF("%d\n", 1965)	d				1965
PRINTF("%6d\n", 1965)	d		6		1965
PRINTF("%-6d\n", 1965)	d	-	6		1965
PRINTF("%+d\n", 1965)	d	+	6		+1965
PRINTF("% 6d\n", 1965)	d	Leerz.	6		1965
PRINTF("%- 6d\n", 1965)	d	-, Leerz.	6		1965
PRINTF("%+6d\n", 1965)	d	-, +	6		+1965
PRINTF("%06d\n", 1965)	d		06		001965
PRINTF("%-06d\n", 1965)	d	-	06		1965
PRINTF("%+06d\n", 1965)	d	+	06		+001965
PRINTF("%-+06d\n", 1965)	d	-, +-	06		+1965
PRINTF("%o\n", 1965)	o				3655
PRINTF("%x\n", 1965)	x				7ad
PRINTF("%X\n", 1965)	X				7AD
PRINTF("%#o\n", 1965)	o	#			03655
PRINTF("%#x\n", 1965)	x	#			0x7ad
PRINTF("%#X\n", 1965)	X	#			0X7AD

Beispiele für Ausgabe der gebrochenen Zahl 1965.3006:

Formatstring, zerlegt in	Typ	Flags	Breite	Präzision	Ausgabe
PRINTF("%f\n", 1965.3006)	f				1965.300600
PRINTF("%12f\n", 1965.3006)	f		12		1965.300600
PRINTF("%-12f\n", 1965.3006)	f	-	12		1965.300600
PRINTF("%+12f\n", 1965.3006)	f	+	12		+1965.300600
PRINTF("% 12f\n", 1965.3006)	f	Leerz.	12		1965.300600
PRINTF("%e\n", 1965.3006)	e				1.965301e+03
PRINTF("%12e\n", 1965.3006)	e		12		1.965301e+03
PRINTF("%-12e\n", 1965.3006)	e	-	12		1.965301e+03
PRINTF("%+12e\n", 1965.3006)	e	+	12		+1.965301e+03
PRINTF("% 12e\n", 1965.3006)	e	Leerz.	12		1.965301e+03
PRINTF("%10.4f\n", 1965.3006)	f		10	4	1965.3006
PRINTF("%10.2f\n", 1965.3006)	f		10	2	1965.30
PRINTF("%10.6e\n", 1965.3006)	e		10	6	1.965301e+03
PRINTF("%10.0f\n", 1965.3006)	f		10	0	1965
PRINTF("%*.7f\n", 14, 7, 1965.3006)	f		14	7	1965.3006000
PRINTF("%10.4g\n", 1965.3006)	g		10	4	1965
PRINTF("%10.1g\n", 1965.3006)	g		10	1	2e+03

Beispiele für Ausgabe der Zeichenkette "Hallo Welt !"

Formatstring, zerlegt in	Typ	Flags	Breite	Präzision	Ausgabe
PRINTF("%s\n", "Hallo Welt !")	f				Hallo Welt !
PRINTF("%15s\n", "Hallo Welt !")	f		15		Hallo Welt !
PRINTF("%-15s\n", "Hallo Welt !")	f		15		Hallo Welt !

Bild 2. Beispielausgaben für Printf anhand von Zahlen und Texten

Typ	Parameter	Beschreibung der Ausgabe
d	ganze Zahl	vorzeichenbehaftete ganze Zahl (Dezimalsystem)
i	ganze Zahl	vorzeichenbehaftete ganze Zahl (Dezimalsystem)
o	ganze Zahl	vorzeichenlose ganze Zahl (Oktalsystem)
u	ganze Zahl	vorzeichenlose ganze Zahl (Dezimalsystem)
x	ganze Zahl	vorzeichenlose ganze Zahl (Hexadezimalsystem unter Verwendung der Buchstaben A bis F)
X	ganze Zahl	vorzeichenlose ganze Zahl (Hexadezimalsystem unter Verwendung der Buchstaben A bis F)
f	Zahl (Fixkomma)	vorzeichenbehaftete Zahl der Form [-]v.vvvv.nnnn, wobei die Anzahl der Nachkommastellen (falls vorhanden) durch Präzision festgelegt wird
e	Zahl (Exponential-schreibweise)	vorzeichenbehaftete Zahl der Form [-]v.nnnn e[+/-]xxx, wobei grundsätzlich nur eine Ziffer vor dem Komma steht, die Anzahl der Nachkommastellen (falls vorhanden) durch Präzision festgelegt wird und der Exponent mindestens zwei Ziffern umfaßt (wird bei Bedarf mit führenden Nullen aufgefüllt)
g	Zahl	ganzzahlig gerundete vorzeichenbehaftete Zahl, wobei das verwendete Format vom Wert und der Angabe für Präzision abhängt. Nachfolgende Nullen entfallen. Das e-Format kommt dann zum Einsatz, wenn der Exponent größer als Präzision oder kleiner als -4 ist.
E	Zahl	wie e-Format, jedoch mit dem Zeichen E (statt e) vor dem Exponenten
G	Zahl	wie g-Format, jedoch mit dem Zeichen E (statt e) vor dem Exponenten, wenn das e-Format verwendet wird
c	Zeichen	einzelnes Zeichen
s	String	Ausgabe einer Zeichenkette bis zum Erreichen eines Null-Zeichens oder der durch Präzision festgelegten Zeichenzahl
%		Ausgabe des Zeichens »%«

Tabelle 1. Typ-Einstellung für Printf-Anweisung

erforderlich. Man erkennt, daß die Printf-Anweisung von C der Write-Anweisung von Pascal im Hinblick auf die Flexibilität weit überlegen ist. Man kann allerdings in Turbo Pascal ohne weiteres per Hand eine Ausgaberroutine programmieren, die dasselbe leistet wie *Printf*, so daß man seine Wahl der Programmiersprache nicht davon abhängig machen sollte. Bild 2 zeigt zahlreiche Beispiele für die Ausgabe von *Printf* mit unterschiedlichen Format-Strings. Durch die Zeichenkonstante `\n` wird der Textausgabe ein Zeilenvorschub angefügt.

Bildschirmausgabe in Quick Basic

Der Quick-Basic-Interpreter stellt zwei unterschiedliche Ausgabeanweisungen zur Verfügung: *Print* und *Print Using*. Mit der *Print*-Anweisung ist es lediglich möglich, numerische Werte oder Zeichenketten unformatiert auszugeben. Dies bedeutet bei Strings, daß sie grundsätzlich linksbündig und in voller Länge zur Ausgabe gelangen. Bei Zahlen erfolgt dies im Fixkommaformat, soweit sie kleiner als 10 000 000 sind. Alle größeren Zahlen erscheinen im Exponentialformat. Grundsätzlich wird das erste Zeichen für ein etwaiges negatives Vorzeichen reserviert, so daß bei positiven Zahlen das erste Zeichen ein nicht unterdrückbares Leerzeichen darstellt. Der einzige Einfluß bei *Print* besteht darin, zu bestimmen, in welcher Form die Ausgabe mehrerer Daten hintereinander in einer Zeile erfolgen soll. Trennt man die Ausdrücke durch Kommata, wird mit der folgenden

Flag/Typ	Bedeutung
-	linksbündige Ausgabe. Wenn nötig (siehe [Breite]), werden Leerzeichen angefügt. Ist das Flag nicht angegeben, wird rechtsbündig ausgegeben und links mit Nullen oder Leerzeichen aufgefüllt.
+	numerische Ausgabe mit Vorzeichen, das heißt, positiven Werten wird ein Pluszeichen vorangestellt. Ist das Flag nicht angegeben, wird nur negativen Werten ein Minuszeichen, positiven Werten überhaupt nichts vorangestellt.
Leerzeichen	Positive Werte werden mit einem führenden Leerzeichen ausgegeben. Falls dieses Flag und das +-Flag fehlen, werden positive Werte ohne Präfix ausgegeben.
#	Konvertierung in sogenannter alternativer Form. Die Formatierung wird durch den Typ festgelegt:
Typ c,s,d,i,u	keine Wirkung auf Ausgabe
Typ o	Falls das Argument ungleich Null ist, wird ihm 0 vorangestellt
Typ x,X	Voranstellen von 0x oder 0X
Typ e,E,f	Hier wird in jedem Fall ein Dezimalpunkt ausgegeben, auch wenn die Zahl keine Nachkommastellen enthält.
Typ g,G	wie e,E. Zusätzlich werden jedoch abschließende Nullen nicht unterdrückt.

Tabelle 2. Flags-Einstellung für Printf-Anweisung

Ausgabe um 14 Zeichen hinter dem Beginn der davorliegenden begonnen. So würde beispielsweise der Befehl

```
PRINT "Hallo", "Peter"
```

die Ausgabe

```
Hallo Peter
```

hervorrufen. Zwischen Hallo und Peter treten neun Leerzeichen auf. Die Summe dieser neun Zeichen plus die Länge von Hallo (fünf Zeichen) ergibt gerade die geforderten 14 Zeichen. Ist der Text jedoch länger, wird er nicht etwa gekürzt, sondern ein 14-Zeichen-Block übersprungen. Der Text würde daher je nach Länge erst 28, 42, 56 und so weiter Zeichen hinter dem vorangegangenen beginnen. Was der Ausgabe des 24 Zeichen umfassenden Textes »Hallo Till, Mann oh Mann« folgt, würde durch

vier Leerzeichen abgetrennt, da 24+4 das nächste Vielfache von 14 ergibt. Statt eines Kommas können Sie die Ausgabe auch durch Semikolon, Leerzeichen oder Tabulator trennen, was dazu führt, daß die Ausgaben bei Texten direkt aneinander anschließen. Der Befehl

```
PRINT "Hallo"; "Hallo"
```

oder

```
PRINT "Hallo" "Hallo"
```

ergäbe somit die Ausgabe

```
HalloHallo
```

Zwischen Zahlen wird ein Leerzeichen eingefügt, um sie optisch voneinander zu trennen. Bei positiven Zahlen entsteht zusammen mit dem Leerzeichen am Anfang der zweiten Zahl leider eine unschöne Lücke mit zwei Leerzeichen.

Als Alternative zum deshalb in der Praxis kaum einsetzbaren *Print*-Befehl bietet Quick Basic die Anweisung *Print Using* an, bei der eine – wenn auch gegenüber C eingeschränkte – Formatierung unterstützt ist. Die allgemeine Syntax von *Print Using* lautet

```
PRINT USING Format Ausdruck [; ,]
```

wobei es sich bei *Format* um einen String handelt, der die Formatierung von *Ausdruck* festlegt. Tabelle 5 zeigt die erlaubten Zeichen für eine Formatierungsanweisung.

Breite	Bedeutung
Zahl	Es werden mindestens <i>Zahl</i> Zeichen ausgegeben. Die Ausgabe wird bei Bedarf durch vorangestellte Leerzeichen (Voreinstellung) beziehungsweise abschließende Leerzeichen (Flag -) aufgefüllt.
0Zahl	Es werden mindestens <i>Zahl</i> Zeichen ausgegeben. Die Ausgabe wird bei Bedarf durch vorangestellte Nullen (Voreinstellung) beziehungsweise abschließende Nullen (Flag -) aufgefüllt.
*	Die Angabe für Breite steht direkt vor dem auszugebenden Parameter

Tabelle 3. Breite-Einstellung für Printf-Anweisung

Präzision/Typ	Bedeutung
keine Angabe	Vorgabe 1 für Typen d,i,o,u,x und X Vorgabe 6 für Typen e, E und F keine Grenzen für c und s alle relevanten Ziffern bei g und G
.0	Standardvorgaben für d,i,o,u,x und X, keine Angabe von Dezimalpunkt und Nachkommastellen für e, E und f. Ist der auszugebende Wert bei den Typen d,i,o,u,x oder X Null, wird überhaupt kein Zeichen ausgegeben.
.Zahl	Ausgabe von <i>Zahl</i> Zeichen beziehungsweise <i>Zahl</i> Dezimalstellen. Ist die Ausgabe länger, wird sie verkürzt beziehungsweise gerundet. Die Wirkung hängt vom verwendeten Parameter-Typ ab:
Typen d,i,o,u	Auffüllung auf <i>Zahl</i> Zeichen (kürzere Ausgabe als <i>Zahl</i> Zeichen) durch x und X vorangestellte Nullen beziehungsweise keine Veränderung (Abschneidung) bei längerer Ausgabe als <i>Zahl</i> Zeichen
Typen e,E,f	<i>Zahl</i> legt die Anzahl der Stellen nach dem Komma fest, wobei wenn nötig gerundet wird.
Typen g,G	<i>Zahl</i> legt die Anzahl der Dezimalstellen fest (Vor- und Nachkommastellen), wobei wenn nötig gerundet wird.
Typ c	keine Wirkung
Typ s	Ausgabe von höchstens <i>Zahl</i> Zeichen

Tabelle 4. Präzision-Einstellung für Printf-Anweisung

Hierbei ist zu beachten, daß die Anzahl der Vorkommastellen durch die Summe aller Format-Zeichen vor dem Dezimalpunkt festgelegt wird. Wenn beispielsweise die Zahl 12.67 vier Vorkommastellen und zwei Nachkommastellen verwenden soll, muß der Format-String »#####.« lauten. Sofern Sie jedoch das Sonderzeichen »+« einbauen, um die Ausgabe mit einem Pluszeichen einzuleiten, dürfen Sie nicht, wie leicht anzunehmen wäre, »+#####.« schreiben (hier würden Sie fünf Vorkommastellen erhalten), sondern Sie müssen ein »#« weglassen. Richtig ist also »+###.##.«.

Das Pluszeichen ersetzt genauso wie »\$« oder »*« ein #-Zeichen. Falls die Anzahl der Platzhalter für die Ausgabe nicht ausreicht, wird die Zahl auf keinen Fall vor dem Dezimalpunkt abgeschnitten, sondern komplett mit dem Vorsatz »%« ausgegeben. Das Prozentzeichen zeigt an, daß Quick Basic die gewünschte Formatierung nicht durchführen konnte.

Bildschirmausgabe in Assembler

Der Prozessor und damit auch die Assembler-Sprache kennen keinerlei Funktionen zur Bildschirmausgabe. Der Prozessor weiß nicht einmal, was ein Bildschirm ist. Zum Glück ist ein PC jedoch mit einem Betriebssystem ausgerüstet, das zumindest zwei Funktionen für die Ausgabe eines einzelnen Zeichens und einer Zeichenkette anbietet. Diese Funktionen, die übrigens auch in kompilierten C- und Pascal-Programmen sowie aus dem Quick-Basic-Interpreter aufgerufen werden, wenn Sie die Befehle *Printf*, *Write* oder *Print* benutzen, sind als sogenannte Software-Interrupts umgesetzt. Eine Interrupt-Routine wird durch den speziellen Prozessorbefehl *Int* aufgerufen. Theoretisch kann jeder PC bis zu 256 Software-Interrupts verwenden, die

von 0 bis 255 durchnummeriert sind. Für DOS-Spezifisches ist die Interrupt-Nummer 21h oder dezimal 33 zuständig, so daß der entsprechende Befehl

```
INT 21h
```

lautet. Da das Betriebssystem nicht nur eine, sondern in der Tat über 100 Funktionen anbietet, um die sich der Interrupt 21h kümmert, hat man sich das Prinzip der Funktionsnummer einfallen lassen. Dabei wird im Register AH eine Zahl von Null bis 6Ch (108 dezimal) geladen, bevor der INT 21h-Befehl ausgeführt wird. Die Interrupt-Routine prüft anschließend, welcher Code sich im Register AH befindet, und springt entsprechend den zuständigen Codeabschnitt an. Die Funktion 02h dient zur Ausgabe eines einzelnen Zeichens. Neben dem Register AH, das dann mit dem Code 02h zu laden ist, wird der ASCII-Code des auszugebenden Zeichens im Register DL übergeben. Jedem auf dem PC für die Ausgabe verfügbaren Zeichen ist ein Code zwischen 0 und 255 zugeordnet, der sogenannte ASCII-Code. Möchten Sie beispielsweise das Zeichen A auf dem Bildschirm ausgeben, könnten Sie folgendes Assembler-Programm benutzen:

```
.MODEL SMALL
.STACK 100h
.DATA
.CODE
mov ax,@data
mov ds,ax
mov ah, 02h
;DOS-Funktion
02h aufrufen
mov dl,65
```

```
;Code für "A"
mov ah,4Ch
int 21h
END
```

Um eine Zeichenkette auszugeben, rufen Sie die DOS-Funktion 09h auf. In Register DX ist dabei die Adresse der Zeichenkette im Speicher zu laden, bevor der *Int*-Befehl ausgeführt wird. Das Ende der Zeichenkette ist mit dem \$-Zeichen zu markieren. Um in einem Assembler-Programm Daten aller Art zu speichern, leiten Sie diese mit einer DB-Anweisung ein. Diese Informationen gehören ins Datensegment, auf das mit Hilfe des DS-Registers zugegriffen wird:

```
.MODEL SMALL
.STACK 100h
.DATA
text db "Hallo DOS-International",13,10,"$"
.CODE
mov ax,@data
mov ds,ax
mov ah, 09h ;DOS-Funktion
02h aufrufen
mov dx,OFFSET text ;Adresse von
"text" laden
mov ah,4Ch
int 21h
END
```

Die Assembler-Anweisung *Offset* sorgt dafür, daß die Adresse und nicht der Inhalt des Speichers ab dem Label (Sprungmarke) *text* geladen wird. Die Byte-Codes 13 und 10 rufen laut ASCII-Tabelle einen Zeilenvorschub hervor und lassen den Cursor an den Anfang der Zeile springen. Formatierungen, wie sie Hochsprachengebiete wie *Write* oder *Printf* anbieten, sind nicht unterstützt. Sie müssen daher den Ausgabetext mit umständlichen Assembler-Sequenzen selbst so »hinbiegen«, wie Sie die Ausgabe wünschen.

(Frank Riemenschneider/hs/bj)

Zeichen	Beschreibung
#	Position einer Ziffer
-	nach der Ziffer platziert, gibt nachfolgendes Minuszeichen für negative Zahlen aus
.	Position des Dezimalpunkts
,	wird links des Dezimalpunkts platziert, gibt Komma nach jeder dritten Ziffer aus
\$	gibt führendes \$ aus
*	füllt führende Leerstellen mit einem *
+	Position des Nummernzeichens
^	gibt im Exponentialformat aus
**\$	kombiniert ** und \$
Zeichen, die einen Text formatieren:	
&	gibt die gesamte Zeichenkette aus
\	gibt die ersten n Zeichen aus, wobei n die Anzahl der Leerzeichen zwischen den Schrägstrichen plus 2 ist
!	gibt nur das erste Zeichen der Zeichenkette aus

Tabelle 5. Formatierungsanweisungen für Print Using

Jonglieren mit Zahlen und Zeichen

Ein Programm dient in erster Linie zur Verarbeitung beliebiger Daten. Genauso, wie man zum Beispiel Unterschiede zwischen Fahrzeugtypen definieren kann (Pkw, Lkw, Motorrad und so weiter), sind auch die Programmdaten in verschiedene Gruppen unterteilbar. In diesem Beitrag erfahren Sie, welche Datentypenvarianten Ihnen Turbo Pascal, Borland C++, Quick Basic und Assembler bieten.

Alle Informationen, die Sie in einem Programm verarbeiten möchten, lassen sich in fünf grundlegende Datentypen unterteilen:

● Ganze Zahlen

Dieser numerische Datentyp dient zur Aufnahme von Zahlen ohne Nachkommaanteil (zum Beispiel 10, -20, 10 000 oder 0).

● Fließkommazahlen

Hierbei handelt es sich um Zahlen, die einen Nachkommaanteil besitzen. Sie werden entweder als Fixkommazahl (wie bei 123.98134) oder in wissenschaftlicher Notation, der sogenannten Exponential-

schreibweise, dargestellt (beispielsweise $3.673 \cdot 10^3$; auf PCs wird der Exponent 10 oft als E geschrieben und das Malzeichen weggelassen, also 3.673E+03).

● Texte

Texte können aus einzelnen Zeichen (A, O) oder Zeichenketten (Hallo, Welt!) bestehen.

● Wahrheitswerte

Dieser auch Boolesche Wert genannte Datentyp darf ähnlich einem Schalter nur zwei Werte annehmen: »wahr« oder »falsch«. Er wurde eingeführt, um das Ergebnis einer Prüfung (trifft zu oder trifft nicht zu) aufnehmen zu können.

● Zeiger

Zeiger enthalten keinen direkten Wert, sondern die Adresse eines Speicherbereiches, in dem sich schließlich ein Wert befindet. Sie zeigen also auf Werte.

Neben diesen isolierten Datentypen gibt es noch die Felder und Strukturen. Ein Feld (auch Array genannt) besteht aus mehreren identischen Datentypen. Nehmen wir ein Beispiel aus dem Leben: Wenn Sie die Anzahl der geschossenen Tore einer Fußballmannschaft einem der zuvor aufgeführten Datentyp zuordnen möchten, kommen Sie ins Grübeln. Natürlich würde eine ganze Zahl ausreichen, aber für welchen Spieltag stünde sie stellvertretend? Die Lösung wäre ein Feld, das aus 34 ganzen Zahlen besteht, wobei jedes Feldelement genau einem Spieltag zugeordnet ist.

Eine Struktur faßt dagegen theoretisch beliebig viele unterschiedliche Datentypen und auch Felder zu einer logischen Einheit zusammen, wie im Falle einer Adreßverwaltung: Zu jedem Eintrag gehören Name, Vorname, Straße, Postleitzahl, Ort und Telefonnummer. Die Adreßstruktur würde daher aus vier Texten und zwei ganzen Zahlen für Telefon und Postleitzahl bestehen.

Da sich die Datentypen bei der Assembler-Programmierung doch erheblich von den übrigen drei Sprachen unterscheiden, werden sie separat am Ende dieses Beitrags behandelt.

Variablen und Konstanten

Um Daten in einem Pascal-Programm zu verwenden, bestehen grundsätzlich zwei Alternativen: entweder in Form einer Variablenanweisung oder durch Definition mit der Compiler-Anweisung *Const* am Programmanfang als Konstante. Im letzten Fall ordnet man Zahl, Text, Zeiger, Feld oder Struktur einen Namen zu, unter dem sich die Konstante künftig ansprechen läßt:

```
CONST Hausnummer = 14;
BEGIN
  WRITELN(Hausnummer);
END.
```

In C wird eine Konstante mit der *#DEFINE*-Anweisung bestimmt:

```
#INCLUDE <stdio.h>
#define Hausnummer 14;
INT MAIN()
{
  PRINTF("%d\n", Hausnummer);
  RETURN(0);
}
```

In Quick Basic sähe das Programm schließlich so aus:

```
CONST Hausnummer = 14
PRINT Hausnummer
END
```

Durch dieses triviale Programm würde die Zahl 14 auf dem Bildschirm ausgegeben. Natürlich hätte man auch gleich

```
WRITELN(14);
oder
PRINTF("%d\n", 14)
beziehungsweise
PRINT 14
```

schreiben und die Definition der Konstanten sparen können. Die Definition einer Konstanten ergibt aber aus zweierlei Gründen Sinn: Zum einen sagt der Name (hier: Hausnummer) natürlich mehr über die Bedeutung eines Wertes aus als lediglich die Zahl selbst. Die Zahl 14 hätte ein Tag, ein Geldbetrag, ein veralteter Mehrwertsteuersatz oder sonst etwas sein können. Durch die Bezeichnung »Hausnummer« weiß man sofort, welchem Zweck die Zahl 14 in diesem Zusammenhang dient. Der zweite Grund kommt dann zum Tragen, wenn eine Konstante mehrmals verwendet wird. Angenommen, Sie wollen die Hausnummer im Programmverlauf nicht nur einmal, sondern vielmals ausgeben:

```
BEGIN
  WRITELN(14);
  WRITELN(14);
  WRITELN(14);
  WRITELN(14);
END.
```

in Pascal beziehungsweise

```
#INCLUDE <stdio.h>
```

```
INT MAIN()
{
  PRINTF("%d\n", 14);
  PRINTF("%d\n", 14);
  PRINTF("%d\n", 14);
  PRINTF("%d\n", 14);
  RETURN(0);
}
```

in C (auf ein entsprechendes Quick-Basic-Beispiel wurde verzichtet, da es sehr einfach ausfällt). Problematisch wird die Angelegenheit mit besagter Verfahrensweise dann, wenn Sie das Programm für den gleichen Zweck benutzen, jedoch Hausnummer 11 einsetzen wollen: Sie müssen nun vier Programmzeilen ändern, bei großen Projekten könnten es jedoch auch leicht einige hundert sein. Besser wäre es mit folgender Programmtechnik verlaufen:

```
CONST Hausnummer = 14;
BEGIN
  WRITELN(Hausnummer);
  WRITELN(Hausnummer);
  WRITELN(Hausnummer);
  WRITELN(Hausnummer);
END.
```

beziehungsweise

```
#INCLUDE <stdio.h>
#define Hausnummer 14;
INT MAIN()
{
  PRINTF("%d\n", Hausnummer);
```



```
PRINTF("%d\n", Hausnummer);
PRINTF("%d\n", Hausnummer);
PRINTF("%d\n", Hausnummer);
RETURN(0);
}
```

Durch Änderung der Zeile mit der Konstantendefinition in

```
CONST Hausnummer = 11;
#DEFINE Hausnummer 11;
```

oder

```
CONST Hausnummer = 11
```

in Quick Basic entfielen die Änderung zahlreicher einzelner Quelltextzeilen.

Die zweite Variante zur Verwendung von Daten in einem Programm ist die Einführung einer Variablen statt einer Konstanten. Zur vorhergehenden Technik der Konstantendeklaration existiert ein wesentlicher Unterschied: Einer Variablen muß man – außer bei der Interpreter-Sprache Quick Basic – explizit einen Datentyp zuweisen, da sie am Programmanfang im Gegensatz zur Konstanten noch keinen definierten Inhalt besitzt und der Compiler wissen muß, wieviel Speicher er reservieren soll. Das folgende Beispielprogramm unterscheidet sich vom vorangegangenen dadurch, daß die Variable *Hausnummer* erst zur Laufzeit des Programms einen Wert erhält:

```
VAR Hausnummer : Integer;
BEGIN
    Hausnummer := 11;
    WRITELN(Hausnummer);
    WRITELN(Hausnummer);
    WRITELN(Hausnummer);
    WRITELN(Hausnummer);
END.
```

in Pascal beziehungsweise

```
#INCLUDE <stdio.h>
INT Hausnummer;
INT MAIN()
{
    Hausnummer = 11;
    PRINTF("%d\n", Hausnummer);
    PRINTF("%d\n", Hausnummer);
    PRINTF("%d\n", Hausnummer);
    PRINTF("%d\n", Hausnummer);
}
```

```
RETURN(0);
}
```

in C. In der ersten Zeile unseres Pascal-Programms deklarieren wir eine Variable mit dem Datentyp *Integer*, die C-Version erreicht dies in der zweiten Zeile durch die *Int*-Anweisung, der daraufhin der Wert 11 zugewiesen wird. In Quick Basic braucht der Typ einer Variablen nicht zwangsläufig festgelegt zu werden. Falls man die Deklaration unterläßt, nimmt der Interpreter an, daß der Programmierer eine Variable einfacher Genauigkeit wünscht. Ein äquivalentes Quick-Basic-Programm sähe daher wie folgt aus:

```
Hausnummer = 14
PRINT Hausnummer
END
```

Die Bedeutung einer Variablen liegt darin, daß sie im Gegensatz zur Konstanten eine Benutzereingabe aufnehmen kann. Die zugehörigen Funktionen zur Eingabe von Daten sind *Readln* bei Turbo Pascal, *Scanf* in Borland C++ und *Input* für Quick Basic. Das folgende Pascal-Programm erfragt vom Benutzer die Eingabe einer Zahl und gibt sie viermal aus:

```
VAR Hausnummer : Integer;
BEGIN
    WRITELN("Bitte geben Sie eine Zahl ein !");
    READLN(Hausnummer);
    WRITELN(Hausnummer);
    WRITELN(Hausnummer);
    WRITELN(Hausnummer);
    WRITELN(Hausnummer);
END.
```

In C sieht das kleine Programm wie folgt aus:

```
#INCLUDE <stdio.h>
INT Hausnummer;
INT MAIN()
{
    PRINTF("Bitte geben Sie eine Zahl ein : ");
    SCANF("%d", &Hausnummer);
    PRINTF("%d\n", Hausnummer);
    PRINTF("%d\n", Hausnummer);
}
```

```
PRINTF("%d\n", Hausnummer);
PRINTF("%d\n", Hausnummer);
RETURN(0);
}
```

oder schließlich in Quick Basic:

```
INPUT "Bitte geben Sie eine Zahl ein :"; Hausnummer
PRINT Hausnummer
PRINT Hausnummer
PRINT Hausnummer
PRINT Hausnummer
END
```

Variablen sind – außer in Quick Basic – genauso wie Konstanten am Anfang des Programms zu definieren. Sie müssen mit einem Buchstaben (a bis z oder A bis Z) oder einem Unterstrich () beginnen. Auf dieses Zeichen können beliebige andere Buchstaben, Zahlen oder Unterstriche folgen. Deutsche Umlaute sind genauso wenig zulässig wie das ß. Bitte beachten Sie, daß Turbo Pascal im Gegensatz zu C nicht zwischen Klein- und Großschreibung unterscheidet. Variablennamen wie *Hallo*, *hallo* oder *HALLO* sind in Pascal identisch, in C jedoch nicht.

Des weiteren gilt zu beachten, daß für den Namen nur die ersten 63 Zeichen (Pascal) beziehungsweise 32 Zeichen (Borland C++) relevant sind. Folgen weitere Zeichen, ist dies Compiler-intern nicht mehr von Bedeutung. Falls Sie also zwei Variablen oder Konstanten definiert haben, die in den ersten 63 beziehungsweise 32 Zeichen übereinstimmen, behandelt der Compiler sie als identisch, ob Sie es wollen oder nicht. Schließlich muß der Programmierer auch noch beachten, daß Namen nicht mit reservierten Schlüsselwörtern übereinstimmen dürfen. So sind zum Beispiel in Turbo Pascal *Begin*, *End* oder *Write* genauso wenig zulässig wie *Read* oder *Var*, während in Borland C++ aus diesem Grund beispielsweise *Short* und *Void* als Variablen- sowie Prozedur- oder Funktionsnamen entfallen. Kommen wir nun zu den verfügbaren Datentypen.

Die Integer-Datentypen

Für ganze Zahlen stellt Turbo Pascal fünf und Borland C++ sieben sogenannte Integer-Typen bereit; Quick Basic bietet leider nur zwei mit den Namen *Integer* und *Long* an. Künftig finden Sie die Datentypen für C immer in Klammern hinter den Pascal-Äquivalenten. Der Datentyp *Byte* (*Unsigned Char*) belegt 1 Byte Speicherplatz und nimmt Zahlen von 0 bis +255 auf. Der Datentyp *Shortint* (*Char*) verwendet ebenfalls 1 Byte des Speichers, doch die Zahlen liegen im Bereich von -128 und +127. Bei der Unterteilung in positive und negative Zahlen wurde eine willkürliche Aufteilung vorgenommen, ebenso gut hätte man zum Beispiel auch Zahlen von -100 bis +155 nehmen können. Die nahezu symmetrische

Aufteilung (128 negative Zahlen, 127 positive Zahlen und die Null) scheint aber für die meisten Anwendungsfälle am sinnvollsten zu sein und ist auch aus dem Bereich der Assembler-Programmierung (beispielsweise durch Branch-Befehle) bekannt.

Zwei weitere Datentypen, *Integer* (*Int* oder *Short Int*) und *Word* (*Unsigned Int*) belegen jeweils 2 Byte Speicherplatz und decken daher einen größeren Zahlenbereich ab. Der Typ *Word* (*Unsigned Int*) vermag Zahlen von 0 bis +65 535 und der Datentyp *Integer* (*Int* oder *Short Int*) Zahlen von -32 768 bis +32 767 darzustellen. Hinsichtlich der Interpretation der Zahlen gilt das zuvor Gesagte.

Schließlich existiert noch ein 4 Byte belegender Datentyp mit dem Namen *Longint* (*Long*). Er kann Zahlen von -2 147 483 648 bis +2 147 483 647 aufnehmen und besitzt in Turbo Pascal kein vorzeichenloses Gegenstück wie *Byte* und *Word*.

Borland C++ hingegen kennt den Datentyp *Unsigned Long*, der ebenfalls 4 Byte belegt und vorzeichenlose Zahlen von 0 bis +4 294 967 295 darstellen kann. Tabelle 1 faßt die fünf Datentypen für ganze Zahlen in Turbo Pascal, Tabelle 2 die sieben Datentypen für Borland C++ zusammen.

In Pascal dürfen Sie diesen Datentypen statt einer Dezimalzahl auch hexadezimale Werte zuordnen, indem Sie ihnen ein \$-Zeichen voranstellen. Statt 160 ließe sich also auch \$A0 schreiben. Dies ist vor allem dann praktisch, wenn in Hand-

büchern oder Dokumentationen programmrelevante Werte in einer anderen Zahlenbasis vorliegen, wobei die hexadezimale Notation sehr häufig vorkommt (zum Beispiel Drucker-Steuerssequenzen oder Interrupt-Aufrufe). In C wird einer Hex-Zahl das Präfix 0x vorangestellt. Statt 255 würden Sie hier also 0xFF verwenden, um sich Umrechnungen zu ersparen. Weiterhin läßt es C auch zu, oktale Zahlen (Zahlenbasis 8) einzusetzen. In diesem Fall stellen Sie eine 0 voran, so daß das Gegenstück zu dezimal 60 oktal 074 lautet.

Die verschiedenen Datentypen sind in Turbo Pascal nicht ohne weiteres ineinander umrechenbar, wie folgendes Programm beweist:

```
var lang: longint;
    dummy : word;
begin
    lang := 10000000;
    dummy := lang*2;
    writeln(dummy);
end.
```

Datentyp	Format	Wertebereich
Shortint	1 Byte mit Vorzeichen	-128...0...+127
Byte	1 Byte ohne Vorzeichen	0...+255
Integer	2 Byte mit Vorzeichen	-32 768...0...+32 767
Word	2 Byte ohne Vorzeichen	0...+65 535
Longint	4 Byte mit Vorzeichen	-2 147 483 648...0...+2 147 483 647

Tabelle 1. Datentypen für ganze Zahlen in Turbo Pascal mit Wertebereichen

Zwar meldet der Compiler keinen Fehler, das ausgegebene Resultat ist jedoch offensichtlich falsch. Dies ist nicht verwunderlich, da die Multiplikation der Longint-Variablen *lang* mit 2 natürlich nicht in eine Word-Variable hineinpaßt. Die Folge ist, daß bei der Zuweisung in die Variable *dummy* die zwei höchstwertigen Bytes entfallen, wodurch sich Berechnungsfehler bei Zahlen größer als 65 535 einschleichen.

An diesem Beispiel ist deutlich zu erkennen, welche wichtige Rolle die korrekte Zuordnung von Variablen zu den dem Anwendungsfall entsprechenden Datentypen spielt. Eine »Reduzierung« des Ergebnisses auf Datentypgröße entsteht allerdings nicht, wenn das Resultat so klein ist, daß es sich durch eine Word-Variable darstellen läßt:

```
var lang: longint;
    dummy : word;
begin
    lang := 100;
    dummy := lang*2;
    writeln(dummy);
end.
```

Hier wird das richtige Ergebnis 200 ausgegeben, da die beiden abgeschnittenen Bytes bei der Zuweisung in *dummy* so wieso 0 enthielten (die Zahl 100 ist so klein, daß von den vier Byte einer Longint-Variablen tatsächlich nur eines mit

einem Wert ungleich 0 belegt ist). In C gilt das gleiche Prinzip. Während das Programm

```
#include <stdio.h>

unsigned long Lang;
unsigned int Dummy;

int main(void)
{
    Lang = 10000000;
    Dummy = Lang*2;
    printf("%u", Dummy);
    return(0);
}
```

ein falsches Ergebnis liefert, da das Long-Ergebnis des Ausdrucks *Lang*2* nicht in die Unsigned-Int-Variable *Dummy* hineinpaßt, liefert das Programm

```
#include <stdio.h>

unsigned long Lang;
unsigned int Dummy;

int main(void)
```

```
{
    Lang = 100;
    Dummy = Lang*2;
    printf("%u", Dummy);
    return(0);
}
```

das korrekte Resultat 200.

Auch die Hypothese »die zugewiesene Variable auf der linken Seite muß nur mindestens so groß sein wie der Ausdruck auf der rechten Seite« ist nicht zutreffend, wie folgende Beispiele eindrucksvoll belegen:

```
var mittel : word;
    dummy : longint;
begin
    mittel := 1000;
    dummy := mittel*mittel;
    writeln(dummy);
end.
```

beziehungsweise das C-Äquivalent:

```
#include <stdio.h>

unsigned int Mittel;
unsigned long Dummy;

int main(void)
{
    Mittel = 10000;
    Dummy = Mittel*Mittel;
    printf("%lu\n", Dummy);
    return(0);
}
```

Obwohl die Longint- beziehungsweise Long-Variable *Dummy* für das Produkt »1000*1000« groß genug dimensioniert wäre, gibt das Programm ein falsches Ergebnis aus. Wieso dies?

Hierzu müssen Sie etwas in die Niederungen des Compiler-Baus herabsteigen. Wenn der Compiler den Ausdruck »mittel*mittel« berechnet, reserviert er für das Ergebnis zunächst nur so viel Speicher, wie es die Variablen selbst – beziehungsweise die größere von beiden – erfordern, in diesem Fall also 2 Byte. Da das Resultat nicht in 2 Byte darstellbar ist, wird der höherwertige Teil abgeschnitten und das Ergebnis damit verfälscht. Erst jetzt findet die Zuweisung des falschen Rests der Longint-Variablen *dummy* statt, wobei die obersten zwei Byte unberührt blieben.

Die richtige Regel ist daher, daß schon bei der Berechnung eines Ausdrucks genug Speicher für das Ergebnis reserviert werden muß. Wäre in diesem Fall die Variable *mittel* vom Typ *Longint* beziehungsweise *Long* gewesen, hätte es keine Probleme gegeben. Wie verhält es sich aber, wenn die Komponenten eines Ausdrucks unterschiedliche Datentypen aufweisen?

Glücklicherweise belegt der Compiler für das Ergebnis soviel Speicher, wie der größere Datentyp benutzt. Berechnen Sie zum Beispiel in Turbo Pascal das Produkt aus einer Byte- und einer Word-Variablen, wird das Ergebnis vom Typ *Word* sein. Es gelten folgende Regeln:

- Die Typen *Char* und *Short* (*Short*) überführt der Compiler bei gemischten Datentypen in Ausdrücken in *Integer* (*Int*), den Typ *Byte* (*Unsigned Short*) in *Word* (*Unsigned Int*).
- Wenn ein Operand den Typ *Longint* (*Long*) verwendet, wird auch der andere dahingehend konvertiert.

Datentyp	Format	Wertebereich
Char	1 Byte mit Vorzeichen	-128...0...+127
Unsigned Char	1 Byte ohne Vorzeichen	0...+255
Int	2 Byte mit Vorzeichen	-32 768...0...+32 767
Short Int	2 Byte mit Vorzeichen	-32 768...0...+32 767
Unsigned Int	2 Byte ohne Vorzeichen	0...+65 535
Long	4 Byte mit Vorzeichen	-2 147 483 648...0...+2 147 483 647
Unsigned Long	4 Byte ohne Vorzeichen	0...+4 294 967 295

Tabelle 2. Datentypen für ganze Zahlen in Borland C++ mit Wertebereichen

- Sofern ein Operand vom Typ *Unsigned Long* vorkommt, wandelt der Compiler den anderen ebenfalls in *Unsigned Long* um.
- Tritt ein Operand vom Typ *Unsigned* auf, wird auch der andere in *Unsigned* überführt.
- Sind beide Operanden gleichen Typs, so gilt er auch für das Ergebnis.

Der Quick-Basic-Interpreter ist nicht so tolerant wie seine Compiler-Kollegen. Betrachten Sie hierzu bitte folgendes Programm:

```
A% = 170
```

```
B% = 170
C% = A% * B%
PRINT C%
END
```

In diesem Fall wurden zwei Integer-Variablen, A% und B%, sowie eine Long-Variablen, C%, definiert. Das Prozentzeichen sagt aus, daß es sich um Variablen des Typs *Integer* handelt. Hätte man nur A oder B geschrieben, hätte Quick Basic den Standard-Typ »einfache Genauigkeit« zugewiesen. Das &-Zeichen wiederum gibt unter Quick Basic Long-Variablen an. Das Programm arbeitet einwandfrei, da es als Ergebnis

28 900 ermittelt. Wenn man A% und B% statt 170 den Wert 300 zuweist, bricht der Interpreter jedoch mit der Fehlermeldung »Überlauf« ab. Anders als die Borland-Compiler reklamiert er sofort, daß das Resultat 90 000 nicht mehr in einer Integer-Variablen unterzubringen ist. Um gerade Programmieranfänger vor solchen versteckten Fehlern zu bewahren, arbeitet Quick Basic wesentlich formaler, das heißt weniger fehlertolerant als Turbo Pascal und Borland C++. Auf der anderen Seite besitzt der Programmierer aber auch weniger Alternativen, seiner Fantasie freien Lauf zu lassen.

Die Fließkomma-Datentypen

Für viele Zwecke reichen die ganzen Zahlen nicht aus. Wenn Sie zum Beispiel die Division 10/6 im ganzzahligen Format ausführen, ist Ihr Ergebnis falsch auf die ganze Zahl 1 abgerundet worden. Für diese Fälle bieten Turbo Pascal, Borland C++ und Quick Basic die sogenannten Fließkomma- oder Realzahlen an.

Im zweiten Beispiel ist die 5 nicht mehr erfaßbar, da die Anzahl der verfügbaren Dezimalstellen nicht ausreicht. In diesem Fall hätte man das Problem durch Einsatz des Datentyps *Comp* lösen können, in anderen Fällen (zum Beispiel periodischen Brüchen wie 1/3) würde aber auch hier ungenau gerechnet, da unendlich viele Dezi-

Routinen unterscheiden sich prinzipiell nicht von denen, die bei Verwendung von Realzahlen zum Einsatz kommen).

Der Emulator wird durch die Compiler-Anweisung

```
{ $N, $E+ }
```

eingebunden. Die Emulatorroutinen arbeiten allerdings wesentlich langsamer als ein mathematischer Coprozessor. Das folgende Beispielprogramm zeigt, welche Unterschiede bei der Verwendung der einzelnen Fließkommaformate entstehen können. Dabei wollen wir die Rechnung »Quadratwurzel aus 2« (auf eine zugegebenermaßen umständliche Weise) ausführen. Für den mathematischen Bereich stellt Turbo Pascal interne Anweisungen (Bibliotheksroutinen) bereit. Die Funktion *Sqrt* beispielsweise zieht die Quadratwurzel, während die Exp-Funktion die Rechnung e^x ausführt:

```
{ $N+, E+ }
```

```
VAR
dummy1: single;
dummy2: real;
dummy3: double;
dummy4: extended;
dummy5: comp;
i: byte;
BEGIN
  dummy1 := Sqrt(2)/Exp(5);
  dummy2 := Sqrt(2)/Exp(5);
  dummy3 := Sqrt(2)/Exp(90);
  dummy4 := Sqrt(2)/Exp(90);
  dummy5 := Sqrt(2)*Exp(43);
  for i:=1 to 7 do begin
    dummy1 := dummy1/Exp(5);
    dummy2 := dummy2/Exp(5);
    dummy3 := dummy3/Exp(90);
    dummy4 := dummy4/Exp(90);
    dummy5 := dummy5/Exp(0.1);
  end;
  for i:= 1 to 7 do begin
    dummy1 := dummy1*Exp(5);
```

Datentyp	Format	Wertebereich	Genauigkeit
Single	4 Byte	$1.5 \cdot 10^{-45}$ bis $3.4 \cdot 10^{+38}$	7 bis 8 Dezimalstellen
Real	6 Byte	$2.9 \cdot 10^{-39}$ bis $1.7 \cdot 10^{+38}$	11 bis 12 Dezimalstellen
Double	8 Byte	$5.0 \cdot 10^{-324}$ bis $1.7 \cdot 10^{+308}$	15 bis 16 Dezimalstellen
Extended	10 Byte	$1.9 \cdot 10^{-4951}$ bis $1.1 \cdot 10^{+4932}$	19 bis 20 Dezimalstellen
Comp	8 Byte	$-9.2 \cdot 10^{+18}$ bis $9.2 \cdot 10^{+18}$	18 bis 19 Dezimalstellen

Tabelle 3. Datentypen für Realzahlen in Turbo Pascal mit Wertebereichen und Genauigkeit

Eine Zahl dieses Datentyps besteht aus drei Elementen: dem Vorzeichen, der Mantisse und dem Exponenten. Die Realzahl wird dabei in folgendem Format angegeben:

```
[Vorzeichen] [Mantisse] * 10^ [Exponent]
```

Turbo Pascal stellt fünf, Borland C++ drei und Quick Basic zwei Realzahlformate zur Verfügung, die sich durch den Umfang des belegten Speichers und damit in der Anzahl der verfügbaren Bits für Mantisse und Exponent unterscheiden (Tabellen 3 und 4). Ein Sonderfall ist der Pascal-Datentyp *Comp*, der sich zwar besonders schnell verarbeiten läßt, allerdings keinen Exponenten enthält und dadurch nur einen sehr eingeschränkten Wertebereich aufweist. Er ist daher von den folgenden Aussagen ausgeschlossen.

Während der Umfang der Mantisse und der des Exponenten die maximale Genauigkeit festlegen, ist für den Wertebereich ausschließlich der Exponent zuständig. Die Genauigkeit einer Realzahl hängt nicht nur von der Größe der Zahl, sondern von der Position der einzelnen Ziffern in der Zahl selbst ab. Betrachten Sie dazu die Beispiele einer Realzahl in Tabelle 5, die durch den Datentyp *Real* mit elf Dezimalstellen beschrieben sein soll.

malstellen existieren. Das dritte Beispiel der Tabelle 5 zeigt, daß auch die Speicherung betragsmäßig kleiner Zahlen ungenau erfolgt, wenn sie mehr Dezimalstellen beinhalten, als die Mantisse aufzunehmen vermag.

Der einzige »normal« vorhandene Fließkommatyp in Turbo Pascal ist *Real*. Die übrigen vier Datentypen sind auf einen mathematischen Coprozessor von Typ 8087, 80287, 80387 oder den im 80486-Prozessor integrierten Coprozessor zugeschnitten. Sie stehen erst nach der Compiler-Anweisung

```
{ $N+ }
```

sowie bei Vorhandensein eines 80 x 87-Coprocessors bereit. Allerdings bietet Turbo Pascal 6.0 einen Emulator an, der beim Programmstart überprüft, ob ein mathematischer Coprozessor vorhanden ist, und bei Fehlen desselben eine Emulation per Software vornimmt (die hierfür verwendeten

Datentyp	Format	Wertebereich	Genauigkeit
Float	4 Byte	$3.4 \cdot 10^{-38}$ bis $3.4 \cdot 10^{+38}$	7 Dezimalstellen
Double	8 Byte	$1.7 \cdot 10^{-308}$ bis $1.7 \cdot 10^{+308}$	15 Dezimalstellen
Long Double	10 Byte	$3.4 \cdot 10^{-4932}$ bis $1.1 \cdot 10^{+4932}$	19 Dezimalstellen

Tabelle 4. Datentypen für Realzahlen in Borland C++ mit Wertebereichen und Genauigkeit

Zahl	gespeichert als	Genauigkeit
1234567000000000000000	1.2345670000*10 ²¹	100 Prozent
1234567000000000000000	1.2345670000*10 ²¹	< 100 Prozent
2.987654321987654321	2.9876543219*10 ⁰	< 100 Prozent

Tabelle 5. Beispiele einer Realzahl mit Genauigkeit und Abweichungen

```

dummy2 := dummy2*EXP(5);
dummy3 := dummy3*EXP(90);
dummy4 := dummy4*EXP(90);
dummy5 := dummy5*EXP(0.1);
end;
writeln('SINGLE-Ergebnis
:',dummy1*EXP(5));
writeln('REAL-Ergebnis :',dum-
my2*EXP(5));
writeln('DOUBLE-Ergebnis
:',dummy3*EXP(90));
writeln('EXTENDED-Ergebnis
:',dummy4*EXP(90));
writeln('COMP-Ergebnis :',dum-
my5*EXP(43));
END.

```

Nach dem Betrachten des Programms müßte man vermuten, daß viermal das gleiche Resultat ausgegeben wird. Doch die Praxis sieht leider anders aus (Tabelle 6). Das Ergebnis wird von oben nach unten

die aufsteigende Genauigkeit der Datentypen *Float*, *Double* und *Long Double* (Tabelle 4). Quick Basic bietet zwei Datentypen, *Single* und *Double*, an, die im Aufbau und Wertebereich den gleichnamigen Turbo-Pascal-Typen (Tabelle 3) entsprechen. Der Datentyp *Single* ist dabei der erwähnte Standardtyp »einfache Genauigkeit« und wird durch das Suffix »!« gekennzeichnet, während eine Double-Variable durch das Suffix »#« bestimmt ist:

```

dummy1 = SQR(2) / EXP(5)
dummy2! = SQR(2) / EXP(5)
dummy3# = SQR(2) / EXP(90)
FOR i% = 1 TO 7
  dummy1 = dummy1 / EXP(5)
  dummy2! = dummy2 / EXP(5)
  dummy3# = dummy3# / EXP(90)
NEXT i%
FOR i% = 1 TO 7
  dummy1 = dummy1 * EXP(5)

```

zunehmend genauer, da die Datentypen immer mehr Speicher für Mantisse und Exponent zur Verfügung stellen. Gleiches gilt prinzipiell für ein C-Programm und

```

dummy2! = dummy2 * EXP(5)
dummy3# = dummy3# * EXP(90)
NEXT i%
PRINT ("STANDARDTYP-Ergebnis :");
dummy1 * EXP(5)
PRINT ("SINGLE-Ergebnis :"); dum-
my2! * EXP(5)
PRINT ("DOUBLE-Ergebnis :"); dum-
my3# * EXP(90)
END

```

Man sieht, daß die Variablen *dummy1* und *dummy2!* identische Ergebnisse liefern, da sie vom gleichen Typ sind. Sofern Sie Ihre Single-Variablen also als solche kennzeichnen möchten, können Sie zur besseren Übersicht das Suffix »!« verwenden. Aber auch wenn Sie es weglassen, würde sich am Programmablauf nichts ändern.

Datentyp	Ergebnis
Single	1.41421353438104E+0000
Real	1.41421356236850E+0000
Double	1.41421356238136E+0000
Extended	1.41421356237310E+0000
Comp	1.41421356237310E+0000

Tabelle 6. Abhängigkeit eines Realzahlresultates vom verwendeten Datentyp

Die Text-Datentypen

Turbo Pascal besitzt zwei Datentypen, mit denen sich Zeichen speichern lassen: Der Datentyp *Char* ist dabei für einzelne, in Hochkommata eingeschlossene Zeichen zuständig (zum Beispiel 'A', '1', 'n'). Bitte beachten Sie, daß auch Zahlen unbedingt in Hochkommata gesetzt sein müssen, da der Compiler sie sonst als Zahlentyp interpretiert.

Jedem Zeichen ist ein eigener ASCII-Code zugeordnet. Aus diesem Grund sind Klein- und Großbuchstaben nicht identisch. Ein Vergleich der Char-Konstanten a und A würde somit das Ergebnis »falsch« hervorrufen. Zahlreiche Zeichen lassen sich nicht direkt über die Tastatur eingeben. Um diese trotzdem verarbeiten zu können, bietet Turbo Pascal drei Alternativen: Steuerzeichen von [Strg-A] bis [Strg-Z] darf man durch die Zuweisung '^A' bis '^Z' ersetzen, wobei das erste Zeichen quasi die Strg-Taste nachbildet. Die Sequenz '^/' steht stellvertretend für die Escape-Taste. Alle anderen Zeichen lassen sich durch direkte Eingabe des ASCII-Codes hinter einem Doppelkreuz verwenden, so zum Beispiel 'A' durch '#65', 'a' durch '#133' und so weiter. Mit dieser Technik steht also auch der bequemen Integration von Grafikzeichen in den Quelltext nichts im Weg. Eine letzte Alternative besteht darin, das Zeichen über die numerische Tastatur einzutippen. Sofern Sie beispielsweise das Zeichen mit dem ASCII-Code 200 als Char-Datentyp eingeben möchten, halten

Sie nach dem ersten Hochkomma die Alt-Taste gedrückt und geben auf dem numerischen Ziffernblock nacheinander die Zahlen 2, 0 und nochmals 0 ein. Wenn Sie nun die Alt-Taste loslassen, erscheint das Zeichen mit dem ASCII-Code 200 auf dem Bildschirm. Ein abschließendes Hochkomma macht die Eingabe perfekt.

Letztere Methode steht ebenfalls bei Borland C++ bereit. Doppelkreuz und das Zeichen ^ lassen sich hier allerdings nicht verwenden. Dafür kann man durch den Einsatz sogenannter Escape-Sequenzen jedes beliebige Zeichen ausgeben. Eine Escape-Sequenz wird durch das Zeichen »\« eingeleitet. Als nächstes folgt entweder ein Sonderzeichen (Tabelle 7), das direkt die Ausgabe eines zugeordneten Zeichens bewirkt, oder ein hexadezimaler beziehungsweise oktaler ASCII-Code, der das Zeichen repräsentiert. Das folgende Programm gibt das Grafikzeichen mit dem ASCII-Code 200 (hexadezimal C8) aus:

```

#include <stdio.h>

char *zeichen;
int main(void)
{
  zeichen = '\xC8';
  printf("%c\n", zeichen);
  return(0);
}

```

Die bekannteste Escape-Sequenz ist sicherlich »\n«, womit der Programmierer einen Zeilenvorschub realisiert.

In Turbo Pascal besteht der Datentyp *String* aus einer Aneinanderreihung von Zeichen des Typs *Char* sowie einem Byte, in dem die momentane Länge der Zeichenkette gespeichert ist. Standardmäßig reserviert Turbo Pascal bei der Definition einer Variablen des Typs *String* Platz für 255 Zeichen sowie ein Byte, das die aktuelle Stringlänge enthält, da mit einem Byte maximal 256 Zahlen darstellbar sind. In den allermeisten Fällen wird durch diese Definition Speicherplatz verschwendet, da beispielsweise ein Ortsname in einer Adreßverwaltung maximal 50 Zeichen benötigen dürfte. Dem Programmierer steht es daher frei, bei der Festlegung die 255 Zeichen auf eine kleinere Zeichenzahl einzuschränken, indem er die tatsächlich maximal benötigte Anzahl in rechteckigen Klammern hinter dem Schlüsselwort *String* angibt:

```

String[40] { Reservierung von ma-
ximal 40 Zeichen }
String[25] { Reservierung von ma-
ximal 25 Zeichen }
String[255] { Reservierung von ma-
ximal 255 Zeichen, entspricht
STRING ohne Parameter }

```

Falls Sie einer Variablen, deren Länge Sie künstlich beschnitten haben, eine umfangreichere Eingabe zuweisen, wird diese automatisch nach dem letzten reservierten Zeichen abgeschnitten, wie folgendes Beispielprogramm zeigt:

```

var
kurz : string[5];
mittel: string[15];
lang : string;

```


Sequenz	Bedeutung
\a	Alarmton
\b	Backspace
\f	Seitenvorschub
\n	Zeilenvorschub
\r	Carriage Return (Wagenrücklauf)
\t	Tabulator horizontal
\v	Tabulator vertikal
\\	umgekehrter Schrägstrich (Backslash)
\'	Anführungszeichen
\"	Gänsefüßchen
\?	Fragezeichen
\x??	hexadezimaler ASCII-Code mit zwei Ziffern
\O???	oktaler ASCII-Code mit drei Ziffern

Tabelle 7. Escape-Sequenzen in Borland C++

```
BEGIN
lang := 'Hallo Compiler, was tust
Du da?';
mittel := lang;
kurz := mittel;
writeln(lang,#13,#10,mittel,
#13,#10,kurz);
end.
```

Die Bildschirmausgabe ergibt

```
Hallo Compiler, was tust Du da?
Hallo Compiler,
Hallo
```

Wie beim Datentyp *Char* muß der Datentyp *String* in Hochkommata eingeschlossen sein. Möchte man ein Hochkomma selbst als Zeichen speichern, so gibt man es doppelt ein (dies gilt übrigens auch für den Char-Typ). Sie dürfen auch Sonderzeichen über die Methode mit dem Zeichen »'^« oder durch das Doppelkreuz mit anschließendem ASCII-Code verwenden. Hier ist allerdings darauf zu achten, daß die Sonderzeichen außerhalb der Hochkommata stehen. Sind sie nämlich darin eingeschlossen, interpretiert der Compiler sie als normale Textzeichen:

```
var
sdemo : string[70];
cdemo : char;
```

```
BEGIN
cdemo := '';
writeln(cdemo);
sdemo := 'Das ist Hans'' Haus !';
writeln(sdemo);
sdemo := 'Dies ist falsch, da das
Zeichen #133 nicht ausgegeben wird
!';
writeln(sdemo);
sdemo := 'Dies ist richtig, da
das Zeichen '#133' ausgegeben wird
!';
writeln(sdemo);
end.
```

Die Bildschirmausgabe sieht nach dem Start des Beispielprogramms wie folgt aus:

```
'
Das ist Hans' Haus !
Dies ist falsch, da das Zeichen
#133 nicht ausgegeben wird !
Dies ist richtig, da das Zeichen à
ausgegeben wird !
```

Wichtig ist bei einer Kombination von Text in Hochkommata und Sonderzeichen, daß diese direkt aneinander anschließen. Es ist beispielsweise nicht zulässig, zwischen einem Hochkomma und dem folgenden Doppelkreuz ein Leerzeichen zu setzen.

In Borland C++ existiert kein eigener Datentyp für Zeichenketten. Sie sind explizit als Feld aus Char-Zeichen zu definieren. Hierzu wird die Variable um zwei recht-eckige Klammern erweitert, zwischen denen eine Zahl steht, die angibt, wie viele aufeinanderfolgende Bytes der Compiler für Zeichen reservieren soll. So würde zum Beispiel die Variable

```
CHAR Text[40];
```

Platz für 39 Zeichen reservieren, da immer ein abschließendes Nullbyte mit einzuplanen ist. Prinzipiell ist der Pascal-Datentyp *String* nichts anderes als ein Char-Feld. In C wurde lediglich kein zusätzlicher Datentyp eingeführt.

Um einem Char-Feld einen Inhalt zuzuweisen, gibt es eine unbequeme und eine bequeme Technik. Die unbequeme besteht darin, bei jedem Feldelement (Zeichen) eine einzelne Zuweisung vorzunehmen:

```
#include <stdio.h>

char Text[40];
int main(void)
{
Text[0] = 'D';
Text[1] = 'O';
Text[2] = 'S';
printf("%s\n",Text);
return(0);
}
```

Wie gewünscht kommt der Text *DOS* zur Ausgabe. Sicherlich denken Sie, daß es bequemer wäre, eine Zuweisung wie in Pascal vorzunehmen:

```
#include <stdio.h>

char Text[40];
int main(void)
{
Text = "DOS";
printf("%s\n",Text);
return(0);
}
```

Wenn Sie dieses Programm jedoch kompilieren, wird mit einer Fehlermeldung in der Zeile

```
Text = "DOS";
```

abgebrochen. Doch aus welchem Grund? Die Stringzuweisung in C unterscheidet sich von allen anderen Programmiersprachen. Eine direkte Zuweisung, wie sie hier versucht wurde, ist nur dann erlaubt, wenn die Variable über einen Zeiger deklariert wurde. Ansonsten muß man die *strcpy*-Funktion benutzen. Folgendes Beispielprogramm arbeitet daher einwandfrei:

```
#include <stdio.h>
#include <string.h>
char Text[40];
int main(void)
{
strcpy(Text,"DOS");
printf("%s\n",Text);
return(0);
}
```

Quick Basic kennt nur einen Texttyp. Er wird durch das Suffix »\$« gekennzeichnet und nimmt Zeichenketten beliebiger Länge auf.

Der Datentyp Zeiger

Ein Zeiger enthält im Gegensatz zu den bislang vorgestellten Datentypen keinen direkten Wert, sondern die Speicheradresse eines Wertes, er zeigt auf einen Wert. Bei der Variablendeklaration unterscheidet sich ein Zeiger äußerlich nur unwesentlich von einem direkten Datentyp:

```
VAR
text : string;
zeigertext : ^string;
zahl : integer;
zeigerzahl : ^integer;
```

in Pascal oder

```
CHAR Text[255];
CHAR *Zeigertext;
INT Zahl;
INT *Zeigerzahl;
```

in C. Der interne Unterschied ist wesentlich eklatanter als der äußerliche: Die Variablen *text* und *zahl* sind vom Datentyp *String* beziehungsweise *Integer* und belegen im Speicher 256 beziehungsweise 2 Byte. Bei *zeigertext* und *zeigerzahl* handelt es sich dagegen um Zeiger, die stets 4 Byte Speicher belegen (eine Adresse setzt sich aus einem jeweils 16 Bit oder 2 Byte

großen Segment- und Offsetanteil zusammen). Interessant ist, daß man bei der Zeigerdeklaration einen Datentyp angeben muß, der festlegt, auf welchen Datentyp der Zeiger einmal zeigen wird. Den Speicherplatz für eine Variable, auf die ein Zeiger weist, reserviert in Pascal die Prozedur *New*:

```
NEW(zeigertext);
NEW(zeigerzahl);
```

Hiermit nimmt der Programmierer zwei Speicherbereiche von 256 beziehungsweise 2 Byte Umfang für seine Aufgaben in Beschlag und setzt die Zeiger auf die Adresse, ab welcher der reservierte Speicher zur Verfügung steht. Von nun an darf

er den Bereich nutzen, indem er bei der Datenzuweisung dem Variablennamen das Zeichen ^ voranstellt:

```
zeigertext^ := 'Hallo Welt !';
zeigerzahl^ := 4;
```

Durch das Zeichen ^ geht hervor, daß nicht der Zeiger selbst, sondern der Bereich, auf den er zeigt, mit den folgenden Daten gefüllt wird. Am Programmende ist der Speicher allerdings wieder freizugeben. Hierzu dient die Prozedur *Dispose*:

```
DISPOSE(zeigertext);
DISPOSE(zeigerzahl);
```

C reserviert bei der Definition eines Zeigers automatisch Speicherplatz, ein Ersatz für die Prozeduren *New* und *Dispose* ist demnach nicht erforderlich. Wie zuvor erwähnt, dürfen Sie bei einem über einen Zeiger deklarierten Char-Feld Texte wie gewohnt zuweisen:

```
#include <stdio.h>
```

```
char *Text;
int main(void)
{
    Text = "Na, funktioniert also
    doch!";
```

```
printf("%s\n",Text);
return(0);
}
```

Wahrscheinlich fragen Sie sich, warum man diesen umständlichen Weg über Zeiger gehen muß. Ein Argument besteht darin, daß sich über Zeiger wesentlich mehr Daten verwalten lassen als über den direkten Weg. Alle auf herkömmliche Weise deklarierten Variablen und Konstanten legt der Compiler in einem nur 64 KByte großen Datensegment an. Haben Sie beispielsweise vor, eine umfangreiche Datenbank zu programmieren, reicht dieser Platz oft nicht mehr aus. Angenommen, jedes Adreßfeld besteht aus insgesamt 40 Zeichen für Namen, Ort, Straße und ähnliches, so könnten Sie damit genau

$65536/40 = 1638$

Adressen speichern. Da Ihr Programm jedoch noch andere Variablen benötigt, runden wir auf 1500 Adressen ab. Bei der Arbeit mit Zeigern belegt jede Variable allerdings nur 4 Byte. Sie könnten damit im Datensegment Zeiger für

$65536/4=16384$

abgerundet etwa 16 000 Adressen verwal-

ten. Natürlich benötigt man Speicher für die Adressen selbst. Der Clou ist jedoch, daß er nicht mehr innerhalb des Datensegments liegen muß. Falls der Mem-Befehl des Betriebssystems beispielsweise 635 000 Byte für ausführbare Programme anzeigt und Ihr Programmcode und das Datensegment zusammen 200 000 Zeichen belegen, verfügen Sie immer noch über 435 000 Byte für Ihre Adressen. Die Gegenrechnung

$16000 \text{ Adressen} \times 40 \text{ Byte} = 640000$

zeigt, daß die Ablage von 16 000 Adressen dieser Größe im Hauptspeicher scheitert. Immerhin könnten Sie aber

$435000/40 = 10875$

Adressen direkt im Zugriff halten. Dies sind 725 Prozent mehr als bei der herkömmlichen Methode mit der direkten Variablendeklaration.

Ein weiterer Vorteil betrifft den Einsatz von Unterprogrammen. Sofern Sie einem Unterprogramm Daten zur Bearbeitung übergeben möchten, ist es natürlich viel einfacher, einen Zeiger auf die Daten, als die Informationen selbst zu übergeben, besonders wenn es sich um umfangreiche Felder handelt.

Wahrheitswerte in Turbo Pascal

Variablen und Konstanten des Datentyps *Boolean* belegen im Speicher nur 1 Byte und beschränken sich daher auf die Zustände *True* (Bedingung trifft zu) oder *False* (Bedingung trifft nicht zu). Ein Boolesches Ergebnis erhält man bei Vergleichen oder If-Abfragen. Die Konstanten *True* und *False* sind in Turbo Pascal vordefiniert und lassen sich direkt zuwei-

sen. Das folgende Programm demonstriert den Einsatz einer Booleschen Variablen. Betrachten Sie dabei das Ergebnis einer Eingabe zwischen 1 und 10 sowie anderen Eingaben:

```
var
    eingabe : byte;
    bool : boolean;
BEGIN
```

```
writeln('Bitte geben Sie eine
    Zahl zwischen 1 und 10 ein
    ');
readln(eingabe);
bool := ((eingabe >= 1) and
    (eingabe <= 10));
if bool = TRUE then writeln('Sehr
    schön gemacht !')
else writeln('Habe Ihre Fehlein-
    gabe zur Kenntnis genommen!');
END.
```

In C existiert kein äquivalenter Datentyp.

Felder oder Arrays

Felder erlauben es dem Programmierer, mehrere gleichartige Datentypen gewissermaßen in durchnummerierter Form zu speichern. Das klassische Beispiel ist wieder die Adreßverwaltung, bei der man die Adressen zum Beispiel von 1 bis 100 mit Nummern versieht. Statt sich für jede der 100 Straßen einen anderen Variablennamen ausdenken zu müssen, wäre es doch viel einfacher, einen einzigen Namen sozusagen durchnummeriert von 1 bis 100 zu benutzen. Genau dies leisten Felder: Jede Variable oder Konstante wird mit einem Index versehen, der die Einzelposition innerhalb des Feldes angibt.

Ein Konstantenfeld eines bestimmten Datentyps wird in Turbo Pascal mit der Zeile

```
Name : ARRAY[X..Y] OF Datentyp =
    (CX,CX+1,CX+2,...,CY-1,CY);
```

und in C mit

```
Datentyp Name[Anzahl] =
    {C0,C1,C2,...,CAnzahl};
```

definiert. Dabei sind x und y die Indizes, in

deren Bereich die Feldelemente liegen sollen beziehungsweise ist *Anzahl* die Nummer des größten Feldelements. Betrachten Sie zur Klarstellung folgendes Beispiel in Pascal:

```
CONST
    TEXT : ARRAY[0..2] OF STRING[10] =
    ('A','B','C');
    ZAHL : ARRAY[3..7] OF BYTE =
    (1,2,3,4,5);
    REALZAHL : ARRAY[0..5] OF REAL =
    (1.2,2.3,3.4,4.5,5.6,6.7);
```

sowie in C:

```
CHAR TEXT[10][3] = {"A","B","C"};
UNSIGNED CHAR ZAHL[5] =
    {1,2,3,4,5};
FLOAT REALZAHL[6] =
    {1.2,2.3,3.4,4.5,5.6,6.7};
```

Die Anzahl der Elemente in den Klammern muß immer mit der Anzahl der Feldelemente übereinstimmen, die sich in Pascal aus der Rechnung $Y-X+1$ ergibt. Bei dem Feld *Zahl* sind daher beispielsweise

$7-3+1=5$

Elemente zu deklarieren, die man anschließend über die Indizes 3, 4, 5, 6 und 7 in eckigen Klammern abrufen kann:

```
WRITELN(ZAHL[3]) { Ausgabe 1 }
WRITELN(ZAHL[4]) { Ausgabe 2 }
WRITELN(ZAHL[5]) { Ausgabe 3 }
WRITELN(ZAHL[6]) { Ausgabe 4 }
WRITELN(ZAHL[7]) { Ausgabe 5 }
```

Um auf ein bestimmtes Feldelement zuzugreifen, muß man also nur den Feldnamen und – getrennt durch eckige Klammern – den Index angeben. Genauso verhält es sich in C:

```
PRINTF("%d",ZAHL[0]) /* Ausgabe 1
*/
PRINTF("%d",ZAHL[1]) /* Ausgabe 2
*/
PRINTF("%d",ZAHL[2]) /* Ausgabe 3
*/
PRINTF("%d",ZAHL[3]) /* Ausgabe 4
*/
PRINTF("%d",ZAHL[4]) /* Ausgabe 5
*/
```

In C ist es nicht wie in Pascal gestattet, mit einem Index ungleich 0 zu beginnen, vielmehr besitzt das erste Element immer den Index 0.

Bei Variablenfeldern ist die Syntax ähnlich, jedoch werden die Feldelemente erst im Programmverlauf mit Daten belegt. In Pascal sieht das wie folgt aus:

```
VAR
  TEXT : ARRAY[0..2] OF STRING[10];
  ZAHL : ARRAY[3..7] OF BYTE;
  REALZAHL : ARRAY[0..5] OF REAL;
BEGIN
  TEXT[0] := 'Hallo !';
  ZAHL[6] := 4;
  REALZAHL[3] := 234.567;
END.
```

In C dagegen so:

```
#include <stdio.h>

unsigned char ZAHL[5];
float REALZAHL[4];
int main(void)
{
  ZAHL[2] = 111;
  REALZAHL[3] = 123.456;
  printf("%d\n%3.3f\n", ZAHL[2], REAL-
  ZAHL[3]);
  return(0);
}
```

Nicht nur »gewöhnliche« Datentypen, sondern auch Zeiger sind in Feldern integrierbar:

```
var
  textzeiger : array[0..2] of
  ^string;
BEGIN
  new(textzeiger[1]);
  textzeiger[1]^ := 'Hallo, Welt
  !';
  writeln(textzeiger[1]^);
  dispose(textzeiger[1]);
END.
```

in Turbo Pascal oder

```
#include <stdio.h>
char *text[20];
int main(void)
{
  text[1] = "Hallo";
  text[2] = "Welt";
  printf("%s\n%s\n", text[1], text[2])
  ;
  return(0);
}
```

in Borland C++. Hierbei wurde ein aus drei Zeigern bestehendes Feld angelegt. Jedem Zeiger ist ein Feldelement zugeordnet. Es gibt noch eine andere Methode, auf

die wir später noch näher eingehen: ein einzelner Zeiger auf ein komplettes Feld.

Auch Quick Basic erlaubt es, Felder zu definieren. Hierzu existieren die Dim- und die Redim-Anweisungen. Der Unterschied zwischen beiden besteht darin, daß man die mit *Redim* festgelegten Dimensionen später dynamisch verändern darf, während sie bei *Dim* statisch, also unveränderbar sind. Hinter dem Schlüsselwort *As* läßt sich auf Wunsch der Datentyp des Feldes angeben. Falls man diese Angabe wegläßt, wird wie immer der Standardtyp *Single* benutzt:

```
DIM A(1 TO 2) AS INTEGER
A(1) = 12
A(2) = 20
PRINT A(1), A(2)
END
```

ist dasselbe wie

```
DIM A(1 TO 2) AS INTEGER
A%(1) = 12
A%(2) = 20
PRINT A%(1), A%(2)
END
```

Die %-Zeichen im zweiten Beispiel sind überflüssig, da das Feld ohnehin schon als Integer-Feld ausgelegt ist. Das folgende Programm demonstriert den Einsatz der Redim-Anweisung:

```
REDIM A(1 TO 2) AS INTEGER
A(1) = 12
A(2) = 20
PRINT A(1), A(2)
REDIM A(3 TO 4) AS INTEGER
A(3) = 34
A(4) = 56
PRINT A(1), A(2), A(3), A(4)
END
```

Wichtig sind in diesem Zusammenhang drei Dinge: Zum einen muß der Datentyp bei der Neudefinition derselbe wie bei der ursprünglichen Festlegung sein. Hätten Sie statt

```
REDIM A(3 TO 4) AS INTEGER
```

zum Beispiel

```
REDIM A(3 TO 4) AS LONG
```

geschrieben, wäre der Interpreter davon ausgegangen, daß Sie ein neues, zweites Feld definieren wollen. Er hätte demgemäß mit der Fehlermeldung *Doppelte Definition* abgebrochen, da diesem zweiten Feld illegalerweise derselbe Variablenname zugewiesen wurde wie einem

bereits bestehenden. Die zweite wichtige Tatsache betrifft die Initialisierung eines Feldes. Bei der Ausführung einer Dim- oder Redim-Anweisung setzt der Interpreter alle Feldelemente auf 0. Das bedeutet, daß im Beispiel die Print-Anweisung für die Elemente A(1) und A(2) den Wert 0 ausgibt, obwohl ihnen zuvor zwei Werte ungleich 0 zugewiesen wurden. Die Redim-Anweisung hat die alten Variableninhalte gelöscht.

Schließlich darf der Programmierer in Quick Basic die untere Grenze für die Dimension weglassen, womit sie automatisch bei 0 beginnt. Es ist somit zulässig, die Definition

```
DIM A(0 TO 10)
```

vereinfacht auch als

```
DIM A(10)
```

zu schreiben.

Abschließend stellen wir Ihnen noch ein mehrdimensionales Feld vor, selbst wenn es in der Praxis nicht so häufig zum Einsatz kommt wie das eindimensionale. Angenommen, Sie schreiben ein Mathematikprogramm, das räumlich (in der dritten Dimension) mit Temperaturfeldern rechnet. Jede Temperatur im Raum ergibt sich aus einer Funktion der Koordinaten x, y und z. Sie könnten ein Temperaturfeld in Turbo Pascal wie folgt definieren:

```
Temperatur : array[0..9,0..9,0..9]
of real;
Jede neue Dimension wird durch ein Komma getrennt angegeben. Um die Temperatur an einem Punkt auszugeben, benutzen Sie folgende Zeile:
```

```
writeln('Temperatur : ',Tempera-
tur[2,3,4]);
```

Theoretisch dürfen Sie so viele Dimensionen einführen, wie Sie möchten. Beachten Sie aber, daß die Gesamtsumme des durch das Feld belegten Speichers 64 KByte nicht überschreitet. In Borland C++ würde die Definition

```
FLOAT Temperatur[9][9][9]
```

und die Ausgabe

```
PRINTF("%f", Temperatur[2][3][4]);
```

lauten, in Quick Basic die Definition

```
DIM Temperatur(1 TO 9,1 TO 9,1 TO 9)
```

und die Ausgabe

```
PRINT Temperatur(1,2,3)
```

Strukturen

Oft steht ein Programmierer vor dem Problem, daß er mehrere unterschiedliche Datentypen zu einer logischen Einheit zusammenfassen möchte.

Das klassische Beispiel ist wiederum eine Adreßverwaltung. Eine Adresse besteht dabei aus verschiedenen Komponenten (Name, Postleitzahl, Telefon und ähnliches), die in der Regel auf unterschiedliche Datentypen zurückgreifen. Um das Programm logisch besser zu

strukturieren und den Zusammenhang zwischen den einzelnen Adreßkomponenten zu verdeutlichen, ist mit Hilfe der Record-Anweisung (Turbo Pascal) beziehungsweise Struct-Anweisung (Borland C++) eine Struktur definierbar:

```
VAR
  Adresse: RECORD
    Ort, Strasse:
    string[40];
```

Telefon, Postleitzahl:

```
longint;
```

Name, Vorname:

```
string[30];
```

```
END;
```

```
BEGIN
```

```
  Adresse.Name := 'Maier';
  Adresse.Vorname := 'Peter';
  Adresse.Strasse := 'DMV-Strasse
  26';
  Adresse.Ort := 'München';
  Adresse.Postleitzahl := 80456;
  Adresse.Telefon := 0894237867;
END.
```


beziehungsweise in C

```
#include <stdio.h>
#include <string.h>

struct _TADRESSE
{
    char ort[40], strasse[40];
    char name[30], vorname[30];
    long telefon;
    long postleitzahl;
};

struct _TADRESSE Adresse;
int main(void)
{
    strcpy(Adresse.name, "Maier");
    strcpy(Adresse.vorname, "Peter");
    strcpy(Adresse.strasse, "DMV-
    Strasse 26");
    strcpy(Adresse.ort, "München");
    Adresse.postleitzahl = 80456;
    Adresse.telefon = 894237867;
    return 0;
}
```

In diesem Beispiel besteht die Struktur aus sechs Feldern, die insgesamt drei unterschiedliche Datentypen aufweisen. Wichtig ist der Abschluß der Record-Struktur mit der End-Anweisung. Sie dürfen im Pro-

gramm auf jedes Strukturfeld zugreifen, indem Sie den Namen der Struktur (hier: Adresse) und getrennt durch einen Punkt den Feldnamen angeben. Dadurch steigt die Transparenz des Programms wesentlich. Natürlich lassen sich auch Felder einer Struktur definieren:

```
VAR
    Adresse = ARRAY[0..10] OF RECORD
        Ort, Strasse:
            string[40];
        Telefon, Postleitzahl:
            longint;
        Name, Vorname:
            string[30];
    END;

BEGIN
    Adresse[4].Name := 'Maier';
    Adresse[4].Vorname := 'Peter';
    Adresse[4].Strasse := 'DMV-
    Strasse 26';
    Adresse[4].Ort := 'München';
    Adresse[4].Postleitzahl :=
    80456;
    Adresse[4].Telefon :=
    0894237867;
```

END.

in Pascal beziehungsweise

```
#include <stdio.h>
#include <string.h>
struct _TADRESSE
{
    char ort[40], strasse[40];
    char name[30], vorname[30];
    long telefon;
    long postleitzahl;
};

struct _TADRESSE Adresse[10];
int main(void)
{
    strcpy(Adresse[4].name, "Maier");
    strcpy(Adresse[4].vorname, "Pe-
    ter");
    strcpy(Adresse[4].strasse, "DMV-
    Strasse 26");
    strcpy(Adresse[4].ort, "Mün-
    chen");
    Adresse[4].postleitzahl = 80456;
    Adresse[4].telefon = 894237867;
    return 0;
}
```

in C. Man muß einfach nur beachten, daß der Index immer nach dem Strukturnamen erscheint.

Definition von neuen Datentypen

Flexibel wie Turbo Pascal, Borland C++ und Quick Basic nun einmal sind, erlauben sie die Definition neuer Datentypen. Hierzu verwenden Sie in Pascal und Quick Basic die Type- beziehungsweise in C die ähnlich klingende Typedef-Anweisung. Hat man zum Beispiel in Pascal mit

```
TYPE Peter = String[25];
beziehungsweise in C mit
typedef char willi[25];
oder in Quick Basic mit
```

```
TYPE Peter
    DUMMY AS STRING * 25
END TYPE
```

den neuen Datentyp *Peter* definiert, lassen sich von nun an Konstanten und Variablen dieses Datentyps einführen:

```
VAR Eins : Peter;
CONST Zwei: Peter = 'Hallo';
beziehungsweise in C
willi Eins;
willi Zwei = "Hallo";
oder in Quick Basic
```

```
COMMON Eins AS Peter
Eins.dummy = "Hallo"
```

In Quick Basic ist ein neuer Datentyp nur im Zusammenspiel mit der Einführung einer Struktur, das heißt mindestens einem Strukturelement, festlegbar. Um eine Variable, die kein Feld darstellt, mit einem selbstdefinierten Datentyp zu belegen, eignet sich die Common-Anweisung, die die Variable für alle Unterprogramme und Prozeduren verfügbar macht. Der Stern hinter der Angabe *String* beschränkt in Quick Basic die Länge einer Zeichenkette, hier auf 25 Zeichen. Die Definition neuer Datentypen ist vor allen Dingen dann er-

forderlich, wenn Strukturen anzulegen sind, auf die in herkömmlicher Weise kein Zugriff besteht. Als Beispiel dient uns ein Zeiger auf ein Feld. Mit der »konventionellen« Anweisungsfolge

```
VAR
    _Adresse : ARRAY[0..10] OF ^Adres-
    se
```

beziehungsweise

```
struct Adresse * _Adresse[10];
```

würde ein Feld von elf Zeigern angelegt. In diesem Fall erwartet Pascal für jedes Feldelement einzeln die Reservierung von Speicherplatz mit einer New-Anweisung. Um nur einen Zeiger auf das gesamte Feld zu definieren, was Platz im Datensegment spart, da für das gesamte Feld nur noch 4 Byte nötig sind, und in Pascal nur eine New-Initialisierung für das komplette Feld erfordert, mußte man in Pascal logischerweise die Anweisung

```
VAR
    _Adresse : ^ARRAY[0..10] OF Adres-
    se
```

ausführen. Der Compiler meldet jedoch einen Fehler, da *Array* kein bekannter Datentyp ist. Mit der Neudefinition eines Datentyps erreicht man sein Ziel aber dennoch:

TYPE

```
_Adresse = ARRAY[0..10] OF RECORD
    Ort, Strasse:
        string[40];
    Telefon, Postleitzahl:
        longint;
    Name, Vorname:
        string[30];
    END;
```

VAR

```
Adresse : ^_Adresse;
```

BEGIN

```
    NEW(Adresse);
    Adresse^[4].Name := 'Maier';
    Adresse^[4].Vorname := 'Peter';
    Adresse^[4].Strasse := 'DMV-
    Strasse 26';
    Adresse^[4].Ort := 'München';
    Adresse^[4].Postleitzahl :=
    80456;
    Adresse^[4].Telefon :=
    0894237867;
    DISPOSE(Adresse);
END.
```

beziehungsweise in C

```
#include <stdio.h>
#include <string.h>
```

```
typedef struct
{
    char ort[40], strasse[40];
```

	Definition über	Anzahl Zeiger	Speicher im Datensegment
Zeigerfeld	XYZ=ARRAY[X..Y] OF ^...	Y-X+1	4*(Y-X+1)
Zeiger auf Feld	TYPE _XYZ=ARRAY[X..Y] OF ... VAR XYZ: ^_XYZ	1	4

Tabelle 8. Vergleich zwischen Zeigerfeld und Zeiger auf Feld in Turbo Pascal


```
char name[30], vorname[30];
long telefon;
long postleitzahl;
} _Adresse[10];
_Adresse *Adresse;
int main(void)
{
    strcpy(Adresse[4]->name, "Mai-
er");
    strcpy(Adresse[4]->vorname, "Pe-
ter");
    strcpy(Adresse[4]->strasse, "DMV-
Strasse 26");
    strcpy(Adresse[4]->ort, "Mün-
chen");
    Adresse[4]->postleitzahl =
80456;
    Adresse[4]->telefon = 894237867;
    return 0;
}
```

In diesem Beispielprogramm wird das Strukturfeld nicht als Variable, sondern als neuer Datentyp `_Adresse` definiert. Anschließend erzeugen wir eine Variable, die einen Zeiger auf den neuen Datentyp darstellt. Da der neue Datentyp `_Adresse` ein Feld mit jeweils sechs Elementen umfaßt, ist unser Vorhaben gelungen: Über einen einzigen Zeiger besitzen wir Zugriff auf ein ganzes Feld.

Definition über	Anzahl Zeiger	Speicher im Datensegment
Zeigerfeld ... *XYZ[Zahl]	Zahl+1	4*(Zahl+1)
Zeiger auf Feld TYPEDEF ... _XYZ[10]	1	4
_XYZ *XYZ		

Tabelle 9. Vergleich zwischen Zeigerfeld und Zeiger auf Feld in Borland C++

Bitte beachten Sie auch den unterschiedlichen Zugriff auf die Feldelemente im Vergleich zu dem Feld aus Zeigern: Hier steht in Pascal das Zeichen »^« direkt hinter dem Feldnamen, da der Zeiger ja nicht ein einzelnes Feldelement, sondern das Feld als ganzes adressiert. Zuvor bei dem Feld aus Zeigern stand das ^-Zeichen jeweils hinter dem Index (zum Beispiel: `Adresse[4]^Ort`), da jedem Feldelement ein eigener Zeiger zugeordnet war. In C wurde der Punkt (`Adresse[4].Ort`) durch den neuen Operator `->` ersetzt. Hiermit kommt zum Ausdruck, daß über einen einzelnen Zeiger auf ein Strukturelement zugegriffen wird. Die Tabellen 8 und 9 verdeutlichen den Unterschied beider Zugriffsvarianten.

In Quick Basic stehen die besprochenen Zeigeroperationen nicht zur Verfügung. Es

lassen sich jedoch mit Hilfe neuer Typdefinitionen Strukturen einführen, die man auf direktem Weg nicht erreichen könnte, da ein Record-Befehl wie in Turbo Pascal fehlt. Das folgende Beispielprogramm definiert einen neuen Datentyp, der sich aus mehreren »herkömmlichen« Datentypen zusammensetzt. Anschließend wird eine Feldvariable eingeführt. Der Zugriff auf die einzelnen Feldelemente erfolgt durch die Trennung mit einem Punkt:

```
TYPE Haus
    Strasse AS STRING * 20
    Wert AS LONG
END TYPE
DIM Immobilie(1 TO 20) AS Haus
Immobilie(1).Strasse = "Pappelweg"
Immobilie(1).Wert = 352000
PRINT Immobilie(1).Strasse, Immo-
bilie(1).Wert
```

Variante Strukturen

In vielen Fällen benötigt der Programmierer Strukturen mit sozusagen »flexiblem« Aufbau. Während er für eine Adresse sicherlich immer Namen, Orte und Straßen braucht, sieht dies bei der Beschreibung einer Immobilie ganz anders aus: Sehen Sie sich zum Beispiel den Vergleich Eigentumswohnung im Mehrfamilienhaus und Einfamilienhaus an. Gemeinsam besitzen beide sicherlich

die Elemente Ort, Straße und Wohnfläche. Damit sind die Gemeinsamkeiten aber auch schon erschöpft. Während für die Wohnung noch das Obergeschoß und die Unterscheidung links und rechts (vom Treppenhaus) maßgeblich ist, interessieren bei Häusern noch die Grundstücksgröße und die Anzahl der Stockwerke. Um beide Objekte parallel zu verwalten, könnte man entweder zwei unterschiedli-

che Strukturen definieren oder eine Struktur, die alle Feldelemente berücksichtigt. Beide Techniken weisen entscheidende Nachteile auf: Am schlechtesten fährt der Programmierer mit zwei Strukturen. Hier müßte er beispielsweise Sortier Routinen und ähnliches zweifach umsetzen und noch eine künstliche Schnittstelle zwischen beiden Strukturen einbauen. Die zweite Alternative entgeht zwar diesen Nachteilen, verschwendet aber Speicher, da bei Wohnung und Haus die jeweils nicht benötigten Felder leer bleiben.

Variante Strukturen in C++

Leider existiert in C++ keine ähnlich komfortable Unterstützung varianter Strukturen wie in Pascal. Sie dürfen jedoch sogenannte Unions definieren. Hierbei handelt es sich um Strukturen, bei denen jeweils nur ein Element aktiv sein kann. Welches Element aktiv ist, bestimmt die Zuweisung. Betrachten Sie folgendes Beispiel:

```
union demounion
{
    int x;
    char y[30];
    float z;
} demo;
```

Wenn Sie der Variablen `demo.x` einen Wert zuweisen, besitzen Sie keinen Zugriff auf `demo.y` und `demo.z`. Erhält anschließend `demo.z` einen Wert per Zuweisung, verliert `demo.x` seine Gültigkeit und so weiter. Sie sehen, daß man mit dieser

Primitivkonstruktion keinen varianten Record von Pascal nachempfinden kann. Der einzige Ausweg besteht darin, für jede Variation eine Union anzulegen und alle Unions in eine Struktur einzubauen. Das Immobilienprogramm würde in C++ wie folgt aussehen:

```
#include <stdio.h>
#include <string.h>

typedef enum {Links, rechts} Rich-
tung;
typedef enum {Haus, Wohnung} _Im-
mobart;
```

```
struct gemeinsam
{
    char Ort[40], Strasse[40];
    long Postleitzahl;
    unsigned int Wohnflaeche;
```

```
_Immobart Immobilienart
};
```

```
union diff_1
{
    unsigned int Obergeschoss;
    unsigned int Grundstueck;
};
```

```
union diff_2
{
    float Stockwerke;
    Richtung Seite;
};
```

```
struct Immobilie
{
    struct gemeinsam alle;
    union diff_1 unter1;
    union diff_2 unter2;
} _Objekt;
```

```
int main(void)
```



```
{
_Objekt.alle.Immobilienart = Wohnung;
strcpy(_Objekt.alle.Ort, "Hannover");
strcpy(_Objekt.alle.Strasse, "Pappelweg 4a");
_Objekt.alle.Postleitzahl = 30559;
_Objekt.alle.Wohnflaeche = 125;
_Objekt.unter1.Obergeschoss = 2;
_Objekt.unter2.Seite = Links;

_Objekt.alle.Immobilienart = Haus;
strcpy(_Objekt.alle.Ort, "Hannover");
strcpy(_Objekt.alle.Strasse, "Werner-Blumenberg-Weg 26");
_Objekt.alle.Postleitzahl = 30845;
_Objekt.alle.Wohnflaeche = 154;
_Objekt.unter1.Grundstueck = 232;
```

```
_Objekt.unter2.Stockwerke = 2.5;
return 0;
}
```

Bevor Sie den (Programmier-)Schock Ihres Lebens bekommen, lassen Sie uns das Programm erläutern. In den zwei Unions *diff_1* und *diff_2* sind jeweils zwei Alternativen bezüglich Wohnung sowie Haus untergebracht. Die Struktur *gemeinsam* enthält alle weiteren Felder, die allen Wohnobjekten gemeinsam sind. In der ersten Zeile wurde ein neuer Aufzählungs-Datentyp bestimmt, der nur den Inhalt *links* oder *rechts* annehmen kann. Später legen wir in Union *diff_2* die Variable *Seite* dieses Datentyps fest, um bei Mehrfamilienhäusern die Wohnungslage, gesehen vom Treppenhaus, definieren zu können. Die eigentliche Hauptstruktur heißt wie bei unserem Pascal-Programm

Immobilie und besteht aus der Struktur *gemeinsam* sowie den beiden Unions. Die Zuweisung eines Datenelements erfolgt hierbei dreidimensional: Durch den ersten Punkt greifen Sie auf die Struktur *alle* beziehungsweise eine der beiden Unions zu, durch den zweiten Punkt schließlich auf deren Elemente. Der Unterschied zwischen Wohnung und Haus wird hier nicht mehr durch die Variable *Immobilientyp*, sondern direkt durch die Zuweisung erledigt. Ordnet man zum Beispiel der Union-Variablen *unter1* einen Feldinhalt für das Element *Grundstueck* zu, bedeutet dies, daß es sich um ein Haus handeln muß. Die Pseudovariablen *Immobilienart* dient ausschließlich dazu, bei späteren Abfragen zwischen Haus und Wohnung unterscheiden zu können.

Variante Strukturen in Turbo Pascal

Als Lösung bietet Turbo Pascal die sogenannten varianten Strukturen an. Hier richtet sich die Anzahl und Bezeichnung der Elemente nach einer Indexvariablen, die Sie im Programmverlauf auch undefinieren dürfen. Betrachten Sie hierzu folgendes Beispielprogramm:

```
TYPE
  Immobilienart = (Wohnung, Haus);
  Richtung = (Links, Rechts);
  Immobilie = RECORD
    Ort, Strasse:
      string[40];
    Postleitzahl:
      longint;
    Wohnflaeche : word;
    case immobart: Immobilienart of
      Wohnung: (Obergeschoss: byte;
        Richtung);
      Haus: (Grund-
```

```
stueck : word;
werke : real);
END;
VAR _objekt : Immobilie;

BEGIN
  _Objekt.immobart := Wohnung;
  _Objekt.Ort := 'Hannover';
  _Objekt.Strasse := 'Pappelweg 4a';
  _Objekt.Postleitzahl := 30559;
  _Objekt.Obergeschoss := 2;
  _Objekt.Seite := Links;

  _Objekt.immobart := Haus;
  _Objekt.Ort := 'Hannover';
  _Objekt.Strasse := 'Werner-Blumenberg-Weg 26';
  _Objekt.Postleitzahl := 30845;
  _Objekt.Grundstueck := 232;
  _Objekt.Stockwerke := 2.5;
END.
```

Zunächst werden zwei Datentypen *Richtung* und *Immobilienart* als Konstantenfelder definiert, das heißt, sie können ausschließlich die Inhalte *links* oder *rechts* beziehungsweise *Wohnung* oder *Haus* annehmen. In der folgenden Struktur *Immobilie* dient eine Case-Anweisung dazu, den Inhalt der Variablen *Immobart* vom Datentyp *Immobilienart* abzufragen. Falls *Immobilienart* den Inhalt *Wohnung* aufweist, führt das Programm die zwei Strukturelemente *Seite* und *Obergeschoss* ein, beim Inhalt *Haus* hingegen die Strukturelemente *Grundstueck* und *Stockwerke*. Bitte beachten Sie, daß sich beide Situationen gegenseitig ausschließen: Sie können nicht gleichzeitig auf alle vier Elemente zugreifen. Das Programm definiert jeweils einmal eine Struktur einer Wohnung und eines Hauses, wie dies die Indexvariable *Immobart* vorschreibt. Da sie Bestandteil der Struktur ist, müssen Sie ihr, wie auch den anderen Elementen, den Namen der Struktur, getrennt durch einen Punkt, voranstellen.

Integer-Datentypen in Assembler

Alle PC-Prozessoren speichern ganze Zahlen im Byte- oder Wortformat. Die entsprechenden Turbo-Assembler-Anweisungen lauten *DB* beziehungsweise *DW*, gefolgt von beliebig vielen Zahlen. Möchten Sie eine Variable erst zur Programmlaufzeit initialisieren, ist dies durch ein Fragezeichen zu realisieren. Betrachten Sie folgendes Beispielprogramm:

```
.MODEL SMALL
.STACK 100h
.DATA
Eins DB 1
Zwei DW 10000
Drei DB ?
Vier DW ?
```

```
.CODE
mov ax,@data
mov ds,ax
mov al,Eins ;-> Wert 1 nach AL
mov bx,Zwei ;-> Wert 20000 nach BX
mov Drei,al ;-> Wert 1 in Variable
Drei speichern
mov Vier,bx ;-> Wert 10000 in Variable
Vier speichern
mov ah, 4Ch
int 21h
END
```

Der Unterschied bei der Variablenbezeichnung in Assembler gegenüber den Hochsprachen besteht darin, daß der Programmierer hier direkt Speicherzellen adressiert.

Während sich zum Beispiel in Pascal einer Variablenbezeichnung immer nur ein Wert des zugehörigen Datentyps zuordnen läßt, können Sie einem Label (Sprungmarke; Zettel, Etikett) durch Kommata getrennt theoretisch beliebig viele Zahlen zuweisen, auf die Sie durch die Adresse des Labels plus einem Offset (Versatzwert) zugreifen:

```
.MODEL SMALL
.STACK 100h
.DATA
Eins DB 1,2,3,4
Zwei DW 10000,15000,20000
Drei DB ?,?
Vier DW ?,?
.CODE
mov ax,@data
mov ds,ax
mov al,Eins+3 ;-> Wert 3 nach AL
mov bx,Zwei+1 ;-> Wert 15000 nach
```



```

BX
mov Drei+1,al ;-> Wert 3 in Adresse Drei plus ein Byte speichern
mov Vier+1,bx ;-> Wert 15000 in Adresse Vier plus ein Byte speichern
mov ah, 4Ch
int 21h
END

```

Natürlich darf man jeder Byte- oder Wortvariablen auch eine eigene Bezeichnung zukommen lassen, was grundsätzlich auch sinnvoll ist, da die Bedeutung ja erst aus der Bezeichnung des Inhalts hervorgeht. Es gibt aber auch Fälle zusammenhängender Datenblöcke (zum Beispiel Felder), bei denen der Zugriff auf einzelne Elemente

über Offsets die bessere Lösung darstellt. Bei den Prozessoren 80386 und 80486, die auf Wunsch mit 32-Bit-Werten (Doppelwörtern) rechnen, existiert noch eine weitere Anweisung: *DD*. Sie erlaubt es, 32-Bit-Werte im Speicher abzulegen:

```

.MODEL LARGE
.386
.STACK 100h
.DATA
Eins DD 100000
Zwei DD ?
.CODE
mov ax,@data
mov ds,ax
mov eax,Eins ;-> Wert 100000 nach AL

```

```

mov Zwei,eax ;-> Wert 100000 ab Label Zwei ablegen
mov ah, 4Ch
int 21h
END

```

Die Assembler-Anweisung *.386* aktiviert die speziellen Befehle für die 80386- und 80486-Prozessoren.

Mit dem *Dup*-Operator legen Sie mehrere aufeinanderfolgende Werte gleichen Inhalts an. Wenn zum Beispiel zehn Byte-werte mit dem Inhalt 1 einzurichten wären, könnten Sie statt umständlich

```

DB 1,1,1,1,1,1,1,1,1,1

```

auch einfach

```

DB 10 DUP (1)

```

schreiben.

Die Datentypen von Turbo Assembler

Die Assembler-Sprache stellt ein Abbild der Maschinensprache des in Ihrem PC eingesetzten Prozessors dar. Dieser kennt nur ganzzahlige Werte, so daß man mit Fließkommazahlen oder Zeichenketten prinzipiell nicht arbeiten könnte. »Prinzipiell« deshalb, weil sich natürlich kaum ein vernünftiges Programm ohne

Bearbeitung von Zeichenketten oder Fließkommazahlen entwickeln läßt. Vielleicht haben Sie bereits festgestellt, daß das Betriebssystem für Ausgabeoperationen Funktionen zur Verfügung stellt. Genauso sind auch Routinen programmierbar, die mit Fließkommazahlen rechnen und diese auf dem Bildschirm

als Fixkommazahl oder im wissenschaftlichen Format ausgeben. Sie müssen sich jedoch immer über einen Punkt im klaren sein: Diese Funktionen sind nicht Bestandteil des Befehlsumfangs des Prozessors und damit auch nicht der Assembler-Sprache. Der Programmierer ist daher, anders als bei Quick Basic, Turbo Pascal und Borland C++, gezwungen, seine Funktionsbibliotheken selbst zusammenzubasteln.

Zeichenketten und Strukturen in Assembler

Da eine Zeichenkette nichts anderes als eine Aneinanderreihung von Bytewerten darstellt, bei denen jeweils der ASCII-Code für ein spezielles Zeichen steht, bietet der Turbo Assembler dem Programmierer zur Vereinfachung die direkte Eingabe der Zeichen in Gänsefüßchen an. Statt

```

DB 65,66,67,68,69,70,13,10

```

könnte man dies auch einfacher als

```

DB "ABCDEF",13,10

```

formulieren.

Interessanterweise erlaubt auch Assembler, zusammengesetzte Strukturen zu definieren, auf deren Elemente man ähnlich wie in Pascal oder C zugreift. Lediglich die Datentypen der Strukturelemente sind natürlich auf die ganzzahligen Formate Byte, Wort und Doppelwort beschränkt. Das folgende Beispiel bestimmt eine Struktur *Arbeitnehmer* mit den drei Elementen *Name*, *Alter* und *Einkommen*:

```

ARBEITNEHMER STRUC
NAME DB "Platz für 31 Zeichen des Namens"
ALTER DB ?
EINKOMMEN DD ?
ARBEITNEHMER ENDS

```

Um im Datensegment eine Variable des Typs *Arbeitnehmer* anzulegen, lassen sich auf Wunsch hinter dem Strukturnamen in spitzen Klammern Parameter angeben, die die Feldelemente initialisieren. Mit der Zeile

```

Maier ARBEITNEHMER <"Peter Maier",34,85000>

```

wird die Variable *Maier* des Typs *Arbeitnehmer* eingerichtet, wobei der Name mit Peter Maier, das Alter mit 34 und das Einkommen mit 85 000 vorbelegt werden würden. Falls die Werte erst im Programmverlauf eintragen möchten, steht es Ihnen frei, die Struktur auch im »Rohzustand« anzulegen:

```

Maier ARBEITNEHMER

```

In diesem Fall würden der Struktur keinerlei Anfangswerte übermittelt. Schließlich sind Strukturen ebenfalls partiell initialisierbar. Im Beispiel

```

Maier ARBEITNEHMER <,35,>

```

würde ausschließlich das Feld *Alter* mit dem Wert 35 belegt, während Name- und Einkommen-Feld frei blieben. Mit dem *Dup*-Operator legt der Programmierer bei Bedarf Felder von Strukturen an:

```

Mitarbeiter ARBEITNEHMER 10 DUP (<>)

```

Der Zugriff auf ein Feldelement erfolgt analog zu C oder Pascal, indem hinter dem Variablennamen, getrennt durch einen Punkt, das Feldelement spezifiziert wird:

```

mov cl, [Maier.Alter]
mov bx, OFFSET [Maier.Name]
mov eax, [Maier.Einkommen]

```

Durch das Schlüsselwort *Offset* erhält das Register BX die Adresse der Zeichenkette *Peter Maier*. Hätte man es weggelassen, wäre der Inhalt ab der Adresse *Maier*, also die ersten beiden Zeichen P und E geladen worden, genauso wie das Register CL mit dem Inhalt der Speicherzelle *Alter* und das

Register EAX mit dem Inhalt der vier Byte ab der Speicherzelle *Einkommen* belegt werden würde.

Natürlich muß man sich fragen, was Strukturen in einer Sprache, die ohnehin nur mit Speicheradressen rechnet, überhaupt verloren haben. So hätten Sie natürlich statt

```

mov cl, [Maier.Alter]

```

auch

```

mov cl, [Maier+31]

```

schreiben können, da der Name 31 Zeichen belegt (Offsets 0 bis 30). In der Tat bringt eine Strukturdefinition keinerlei funktionelle Vorteile, dafür aber mehr Bequemlichkeit für den Programmierer und vor allen Dingen eine größere Fehler-sicherheit Ihrer Programme: Stellen Sie sich vor, Sie würden sich plötzlich entscheiden, das Feld für den Namen auf 80 Zeichen zu erweitern, weil Sie einen Herrn Otto-Wilhelm Kraftmayer-Schnarchenberger eingestellt haben. Mit der Strukturdefinition brauchen Sie nur eine einzige Zeile zu verändern, während bei der konventionellen Methode jede Zugriffszeile auf ein Feldelement zu korrigieren wäre. Die Anweisung

```

mov cl, [Maier+31]

```

müßte man zum Beispiel jedesmal in

```

mov cl, [Maier+80]

```

revidieren. Geht schließlich eine Zeile vergessen, funktioniert das Programm nicht mehr. Nach dieser Einführung in die Datentypen unterschiedlicher Programmiersprachen sollten Sie eigene Versuche unternehmen und sich mit dieser wichtigen Materie praktisch anfreunden.

(Frank Riemenschneider/bj)

tinen zum Setzen und Auslesen eines Punktes langsam, überträgt sich die mangelnde Geschwindigkeit automatisch auf das gesamte Grafikpaket. Um diesen Flaschenhals zu beseitigen, muß man das BIOS daher durch direkte Programmierung des Grafik-Controllers umgehen.

Das Video-RAM, also der Speicherbereich, in dem die Grafikinformatio-
nen liegen, befindet sich auf der Grafik-
karte und umfaßt mindestens 256 KByte.
Da jeder Grafikpunkt bei 16 Farben durch
4 Bits repräsentiert wird ($2^4=16$),
benötigt man bei einer Auflösung von
640 x 480 Punkten

$$640 \times 480 \times 4/8 = 153600$$

Byte. Der Adreßraum des PC stellt aller-
dings für VGA- und EGA-Videokarten nur
ein 64 KByte großes Segment ab
A000:0000 zur Verfügung, über das sich
auf das Video-RAM zugreifen läßt. Die
Lösung des Problems besteht darin, daß
man dieses Segment technisch betrachtet
in vier sogenannte Bitplanes aufgeteilt hat,
wobei jede Bitplane für jeden Grafikpunkt
nur ein Bit enthält, das gesetzt oder nicht
gesetzt sein kann. Die Farbe eines Grafik-
punktes wird somit durch die Kombination
jeweils eines korrespondierenden Bits in
jeder Bitplane bestimmt (Bild 1).

Da alle vier Bitplanes über den gleichen
Adreßraum ansprechbar sind, mußte man
einen Mechanismus konzipieren, der ei-
nen selektiven Zugriff auf eine bestimm-
te Bitplane erlaubt. Würde man einen
»normalen« Lesebefehl dazu verwenden,
den Inhalt eines Bytes des Video-RAMs
zu ermitteln, könnte die Grafikkarte ja
nicht wissen, welche der vier Bitplanes
gemeint ist.

Die Funktion der Latch-Register

Um dieses Hindernis zu überspringen,
wurden vier sogenannte Latch-Register,
die gewissermaßen als Puffer zwischen der
CPU und den vier Bitplanes stehen, einge-
führt. Bei einem Lesezugriff auf das Vi-
deo-RAM wird das angesprochene Byte
jeweils aus jeder Bitplane in ein Latch-Re-
gister übertragen (Bild 2). Wenn Sie also
beispielsweise das Byte an der Adresse
A000:0005 mit den Befehlen

```
mov di, 0005
mov ax, A000
mov es, ax
mov al, es:[DI]
```

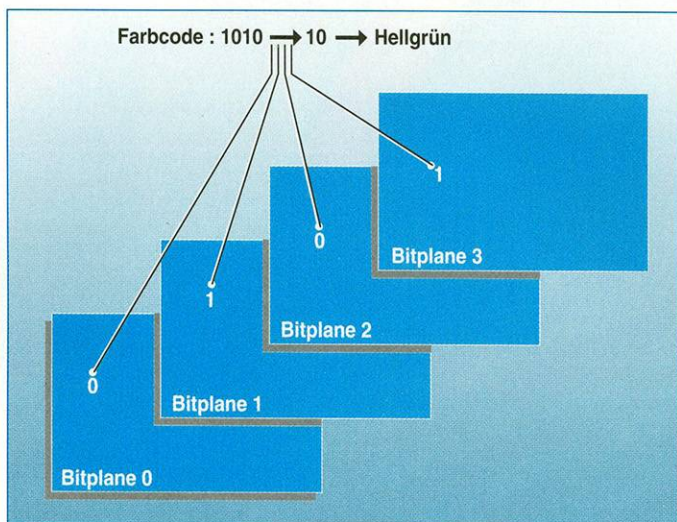


Bild 1. Farbcodierung eines Grafikpunktes

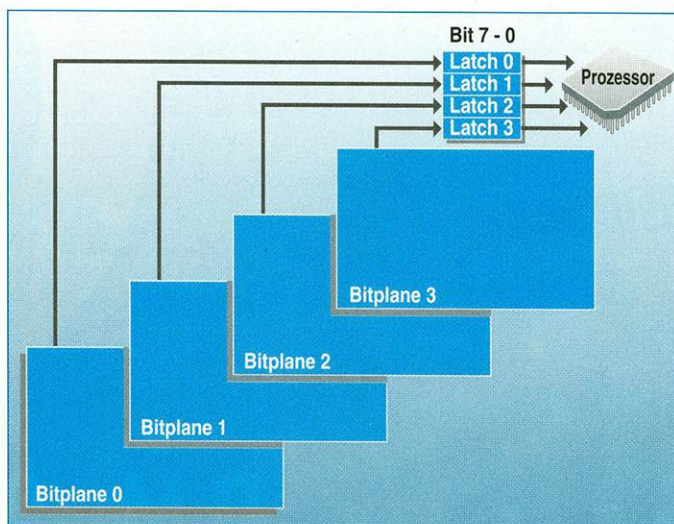


Bild 2. Laden der Latch-Register bei Lesezugriff auf das Video-RAM

auslesen, findet sich immer das sechste
Byte (das erste beginnt bei Offset 0) in ei-
nem Latch-Register. Auf die gleiche Wei-
se verfährt man bei einem Schreibzugriff,
der den Inhalt der Latch-Register jeweils
an der durch den Schreibbefehl festgeleg-
ten Position in die vier Bitplanes überführt
(Bild 3).

Die Latch-Register allein nützen jedoch
überhaupt nichts, da man die Daten ja
über die CPU in den Hauptspeicher trans-
ferieren möchte beziehungsweise Daten
aus dem Hauptspeicher in das Video-
RAM schreiben muß. Da die CPU jedoch
gleichzeitig immer nur ein Byte lesen
oder schreiben kann, war ein Weg zu fin-
den, um festzulegen, welches Latch-Re-
gister von einem Lese- sowie Schreibzu-
griff betroffen ist. Wären nämlich alle
vier Latch-Register gleichzeitig von der
Bearbeitung betroffen, würden sie
zwangsläufig immer den gleichen Inhalt
aufweisen, so daß man sich den ganzen
Aufwand hätte schenken können. Dieses
Prinzip könnte nur noch eine zweifarbige
Darstellung unterstützen (Bit in allen Bit-

planes gesetzt oder in allen Bitplanes
gelöscht), für die man konsequenterweise
nur eine Bitplane benötigen würde und
deren Daten damit komplett in dem 64-
KByte-Adreßraum ab A000:0000 ableg-
bar wären.

Die Lösung der selektiven Adressierung
der Latch-Register liegt im Grafik-Con-
troller jeder EGA- und VGA-Karte, indem
er neun Register zur Verfügung stellt, über
die Sie alle wichtigen Einstellungen vor-
nehmen (Tabelle 2).

An der Port-Adresse 03CEh befindet sich
ein Adreßregister, in das Sie die Nummer
des Registers laden, auf das künftig zuge-
griffen werden soll. Danach steht es Ih-
nen frei, das Register über eine Beschrei-
bung des Datenregisters, das sich an der
Port-Adresse 03CFh befindet, mit dem
gewünschten Wert zu initialisieren. Beide
Zugriffe sind jedoch auch in einem 16-

Bit-Befehl zusam-
menfaßbar. Die In-
tel-Prozessoren
bieten hierzu den
Befehl

OUT DX, AX

an. In diesem Fall
ist das DX-Regi-
ster mit der Port-
Adresse 03CEh,
das Register AL
mit der Register-
nummer und das
Register AH mit
dem Wert, den Sie
in das Register
schreiben möchten,
zu laden. Soll bei-
spielsweise das
Mode-Register
(05) des Grafik-
Controllers den In-
halt 01 erhalten,

wäre folgende Befehlssequenz einzuset-
zen:

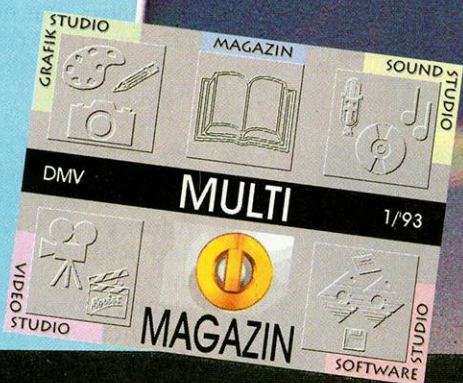
```
mov dx, 03ceh
mov ax, 0105h
out dx, ax
```

Ein Lesezugriff auf die Register des Gra-
fik-Controllers ist übrigens nur bei VGA-
Karten unterstützt, bei EGA-Grafikkarten
führt ein Leseversuch zu zufälligen und
unsinnigen Resultaten, die mit den

Registernummer	Belegung	Voreinstellung
00h	Set/Reset	00h
01h	Enable/Reset	00h
02h	Color Compare	00h
03h	Function Select	00h
04h	Read Map Select	00h
05h	Mode	00h
06h	Miscellaneous	-
07h	Color don't care	0Fh
08h	Bit Mask	FFh

Tabelle 2. Belegung der neun Register des EGA/VGA-Grafik-Controllers

Erleben Sie jetzt



...für nur DM 25,-! ▶

Das MULTI-CD-MAGAZIN bringt Ihnen die Zukunft auf den Bildschirm!

Denn jetzt wird Ihnen die ganze PC-Welt auf völlig neue und dabei höchst unterhaltsame Art und Weise präsentiert: Sie bestimmen per Maus selbst, was Sie lesen, sehen oder hören möchten – ob die heißesten News aus der Computerszene, spannende Artikel, ausführliche Hardwarechecks oder interessante Softwaretests!

Steigen Sie jetzt ein in die faszinierende Multimedia-Welt – auf CD-ROM!

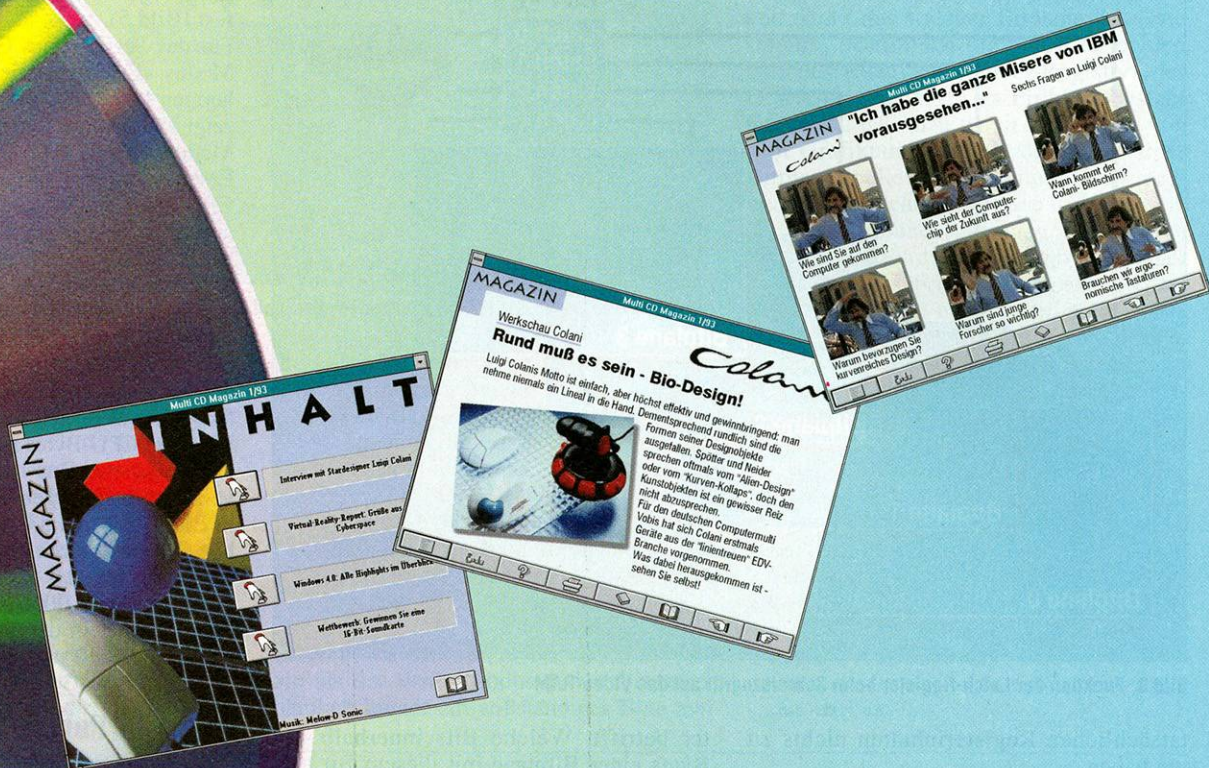
Übrigens: Jede Ausgabe enthält eine große Auswahl an Software- sowie Shareware-Spielen, -Fonts, -Cliparts und -Sounds, Filmberichten und Interviews aus der Computer-Branche.

Hardware-Voraussetzung:

Ein 386er PC, Windows 3.1., 4 MB-Arbeitsspeicher und eine Soundkarte genügen – und Ihr Multimedia-Erlebnis kann losgehen!

JOIN THE FUTURE – MIT DEM MULTI-CD-MAGAZIN!

zt Multimedia ...



▲ Karte sofort abschicken! ▲

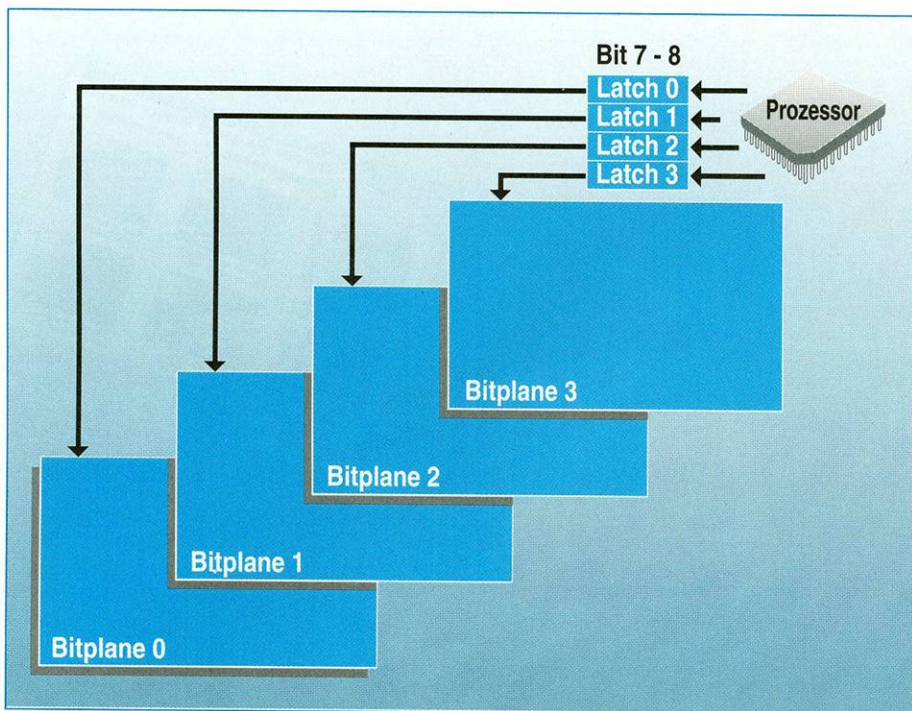


Bild 3. Funktion der Latch-Register beim Schreibzugriff auf das Video-RAM

tatsächlichen Registerinhalten nichts zu tun haben.

Der Lese- beziehungsweise Schreibzugriff auf die Latch-Register wird vornehmlich über das Mode-Register festgelegt. Hier können Sie vier unterschiedliche Arten des Schreibzugriffs (Write Modi) und zwei Varianten des Lesezugriffs (Read Modi) definieren (Bild 4). Für die Grafikprogrammierung von Bedeutung sind der Write Modus 2 und der Read Modus 0, denen wir uns anschließend widmen. Die folgende Befehlssequenz lädt das Mode-Register mit diesen beiden Modi:

```
mov dx, 03ceh
mov ax, 0205h
out dx, ax
```

Der Write Modus funktioniert wie folgt: Die untersten vier Bit des geschriebenen CPU-Bytes werden mit dem adressierten Byte im Video-RAM bitweise in allen Bitplanes verknüpft, wobei Bit 0 nur Bitplane 0, Bit 1 nur Bitplane 1 und so weiter

ter betrifft. Welche Bits innerhalb des Bytes einer Bitplane mit diesem Bit mathematisch zu verbinden sind, hängt vom Inhalt des Bit-Mask-Registers ab. Jedes in diesem Register gesetzte Bit wird mit dem

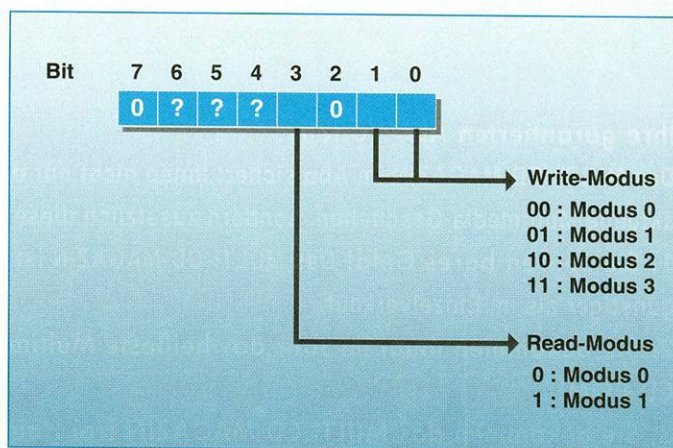


Bild 4. Belegung des Mode-Registers des EGA/VGA-Grafik-Controllers

Spalte	0-7	8-15	16-23	24-31	32-39	40-47	...	600-607	608-615	616-623	624-631	632-639
Zeile 0	0	1	2	3	4	5	...	75	76	77	78	79
Zeile 1	80	81	82	83	84	85	...	155	156	157	158	159
Zeile 2	160	161	162	163	164	165	...	235	236	237	238	239
Zeile 3	240	241	242	243	244	245	...	315	316	317	318	319
...	...	...	...	...	...	...	...	...	...	...	...	...
Zeile 476	38080	38081	38082	38083	38084	38085	...	38155	38156	38157	38158	38159
Zeile 477	38160	38161	38162	38163	38164	38165	...	38235	38236	38237	38238	38239
Zeile 478	38240	38241	38242	38243	38244	38245	...	38315	38316	38317	38318	38319
Zeile 479	38320	38321	38322	38323	38324	38325	...	38395	38396	38397	38398	38399

Tabelle 3. Offsetadresse innerhalb des Video-RAMs in Abhängigkeit der Spalten- und Zeilenkoordinaten

CPU-Bit verknüpft. Die Art der Verknüpfung legt das Function-Select-Register fest (Bild 5).

Für das Setzen eines Grafikpunktes ist der Modus »Ersetzen« wichtig, da er den vorherigen Inhalt des Bits überschreibt. Bild 6 zeigt zwei Anwendungsbeispiele des Write Modus 1. Oben wird angenommen, daß der Ersetzen-Modus eingestellt ist und das CPU-Byte den binären Inhalt 00001010 aufweist (es soll also ein Grafikpunkt in hellgrüner Farbe gesetzt werden). Weiterhin sei vorausgesetzt, daß das Bit-Mask-Register den binären Inhalt 00000100 enthält, womit in jedem Latch-Register nur Bit 2 betroffen ist. Man sieht, daß in Bitplane 0 und Bitplane 2 Bit 2 des adressierten Grafikbytes jeweils gelöscht wird, da Bit 0 und Bit 2 des CPU-Bytes nicht gesetzt sind. Die eingeschalteten Bits 1 und 3 des CPU-Bytes veranlassen hingegen das Setzen von Bit 2 in Bitplane 1 und 3. Im unteren Bildteil sehen Sie das gleiche Beispiel, mit dem Unterschied, daß das Bit-Mask-Register den binären Inhalt 00001001 aufweist. Hier werden daher in den Bitplanes 0 und 2 jeweils die Bits 0 und 3 gelöscht und in den Bitplanes 1 und 3 die Bits 1 und 2 gesetzt.

Punkte setzen ohne BIOS-Bremse

Mit den damit gewonnenen Kenntnissen ist es relativ einfach, eine Plot-Routine zum Setzen eines einzelnen Grafikpunktes zu entwickeln. Eine Voraussetzung besteht jedoch darin, daß man für jeden Grafikpunkt die korrespondierende Byte-Adresse im Video-RAM und die Bit-Position innerhalb dieses Bytes kennt. Da es auf Dauer unzumutbar ist, eine Adressierung auf diese Weise vorzunehmen, entwickeln wir noch ein Modul, das eine Angabe

in x- und y-Koordinaten ($0 \leq x \leq 639$, $0 \leq y \leq 479$) in die effektive Adresse umrechnet. Der Anwender kann dann dieses Plot-Modul wie die BIOS-Routine bequem mit einer Zeilen- und Spaltenangabe als Parameter aufrufen.

Eine Bitplane ist so organisiert, daß das erste Byte an der Adresse A000:0000 die obere linke Bildschirmecke repräsentiert, also die Koordinaten (0,0) bis (7,0). Das oberste Bit stellt dabei die Koordinate (0,0) und das unterste Bit die Koordinate (7,0) dar. Die folgenden Bytes repräsentieren die Fortsetzung der Zeile von links nach rechts. Das Byte an der Adresse A000:0001 beinhaltet die Koordinaten

(8,0) bis (15,0), das Byte an der Adresse A000:0002 die Koordinaten (16,0) bis (23,0) und so weiter, bis schließlich das Byte an der Adresse A000:0040 die erste Bildschirmzeile mit den Koordinaten (631,0) bis (639,0) abschließt. Das nächste Byte beginnt in der zweit-

obersten Zeile und umfaßt die Koordinaten (0,1) bis (7,1). Auf diese Weise ist die Bitplane mit fortlaufenden Adressen dem Bildschirm von links nach rechts und von oben nach unten zugeordnet (Tabelle 3). Da ein Byte im Video-RAM immer acht Grafikpunkte enthält, könnte man den zugehörigen Offset für einen Grafikpunkt der Koordinate (x,y) im Segment A000 einfach mit der Formel

Adresse = $y \cdot 80 + x/8$ berechnen, da jede Zeile mit 640 Punkten 80 Byte umfaßt. Die Bitmaske, also das Bit innerhalb des Bytes, das dem Grafikpunkt entspricht, läßt sich durch die Operation

Bit-Position = $7 - (x - (x/8) \cdot 8)$ berechnen, wobei die Division $x/8$ natürlich ein ganzzahliges Ergebnis liefert. Diese kompliziert aussehende Formel ist programmtechnisch sehr einfach durch die Befehle

```
and al, 7
mov dl, 80h
shr dl, al
umsetzbar, wenn das Register AX die x-Koordinate enthält. Dabei beschreibt allein der Befehl
and al, 7
die Rechnung  $x - (x/8) \cdot 8$ .
```

Koordinatenkomfort durch calxy

Listing 2 zeigt die Berechnungsroutine *calxy*, der in den Registern AX und BX die x- und y-Koordinaten übergeben werden und die im Registerpaar ES:DI die Adresse des korrespondierenden Bytes im Video-RAM sowie im Register DL die Bitmaske zurückliefert. Listing 3 enthält eine Plot-Routine, der man in den Registern AX und BX die x- und y-Koordinaten und im Register DI die Farbe für den Grafikpunkt übergibt. Zunächst wird *calxy* aufgerufen und die Bitmaske mit den Befehlen

```
mov ah, dl
mov dx, 3CEh
mov al, 08h
out dx, ax
in das Bit-Mask-Register des Grafik-Controllers eingetragen. Schließlich laden Sie mit dem Befehl
mov ch, [es:di]
```

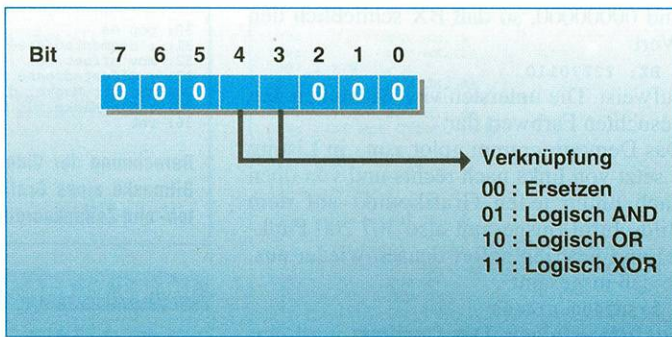


Bild 5. Belegung des Function-Select-Registers des EGA/VGA-Grafik-Controllers

die Latch-Register erst einmal, bevor Sie die Farbe mit dem Befehl

```
mov [es:di], cl
```

hineinschreiben. Sofern der Lesebefehl unterbleibt, hätte man fälschlicherweise den zufälligen Inhalt, der sich zuvor in den Latch-Registern befand, manipuliert und in das Video-RAM zurückgeschrieben. Nicht ganz so einfach ist die Angelegenheit beim Auslesen eines Grafikpunktes. Der zuvor erwähnte Read-Modus 0 funktioniert nämlich nach einem anderen Prinzip als der Write-Modus 1. Hierbei wird nämlich immer ein vollständiges Byte aus

einem der vier Latch-Register zur CPU übergeben. Aus welchem Latch-Register und damit aus welcher Bitplane das Byte stammt, bestimmen die untersten beiden Bits des Read-Map-Select-Registers des Grafik-Controllers. Das Problem besteht nun darin, daß man damit die Inhalte von acht Grafikpunkten, allerdings nur einer Bitplane geliefert bekommt. Von programmtechnischem Interesse ist jedoch ein Bit pro Bitplane für einen bestimmten Punkt. Es bleibt dem Programmierer daher nichts anderes übrig, als nacheinander das vollständige Byte aus allen vier Bitplanes zu lesen, das gewünschte Bit zu isolieren und aus diesen vier Bits einen Farbwert zu erzeugen. Listing 4 stellt eine solche Programmsequenz dar. Zunächst wird wiederum die Routine *calxy* aufgerufen, die das zum adressierten Grafikpunkt korrespondierende Byte im Video-RAM bestimmt. Mit den Befehlen

```
mov ch, dl ; Bitmaske retten
xor bl, bl
mov dx, 3CEh
mov ax, 0304h ; Bitplane 3 in Read-Map-Select-Register
testl:
out dx, ax
```

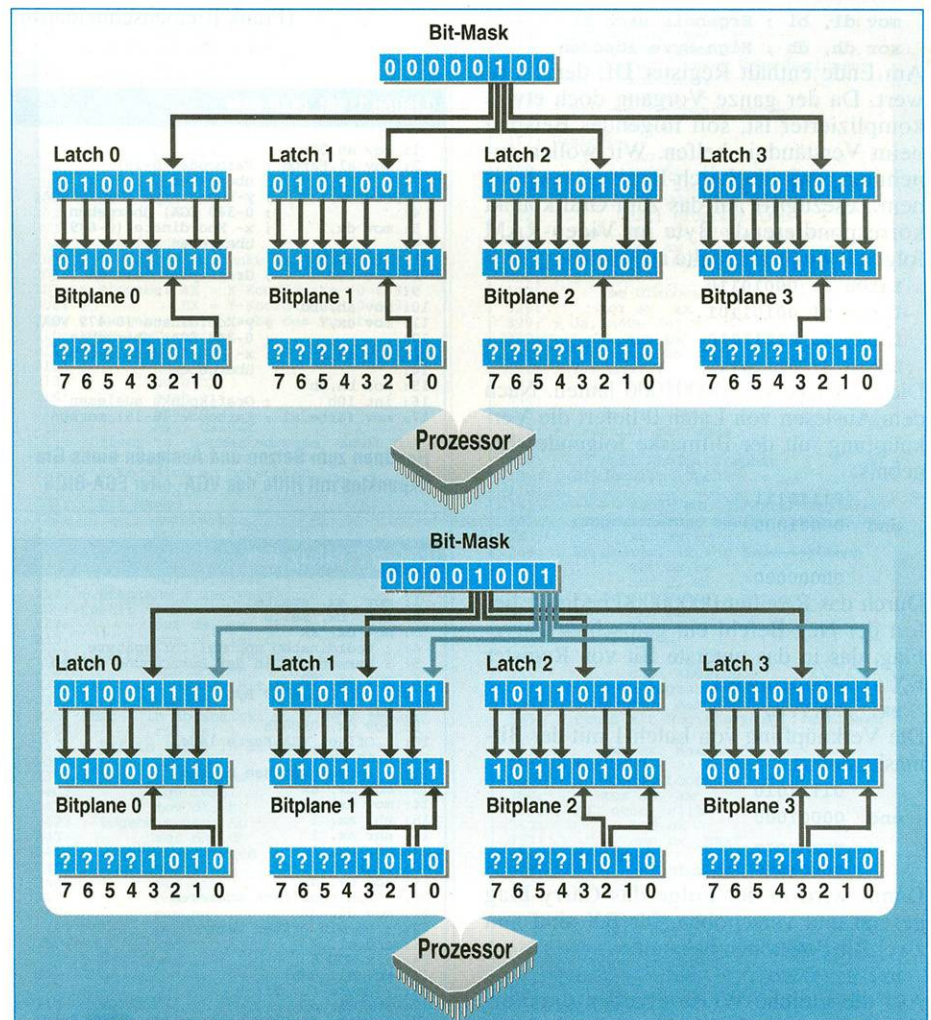


Bild 6. Funktion des Write-Modus 1 bei Function-Select »Ersetzen«


```
mov bh, [es:di] ; Byte aus
Video-RAM lesen
```

```
and bh, ch ; Bitmaske isolieren
```

wird das Byte aus Bitplane 3 gelesen und die durch *calcx* berechnete Bitmaske isoliert. Alle Bits außer dem Bitmasken-Bit sind damit gelöscht. Leider ist die Position des Bitmasken-Bits unbekannt, so daß man nicht unmittelbar Umschalt-Befehle einsetzen kann, sondern zu einem Trick greifen muß. Der Befehl

```
neg bh ; Bit 7 in Abhängigkeit des
Pixels setzen
```

setzt das Carry-Flag immer dann, wenn der Operand ungleich Null ist, ansonsten wird es gelöscht. Da alle Bits außer dem Bit an der Position der Bitmaske 0 enthalten, wird das Carry-Flag also ausschließlich dann gesetzt, wenn das isolierte Bit innerhalb des Grafikbytes gesetzt ist. Mit dem Befehl

```
rol bx, 1 ; Carry-Bit in Bit 0 in BL
transferieren
```

kann man den Inhalt des Carry-Flags auf einfache Weise in das unterste Bit eines Registers transferieren. Auf die gleiche Weise werden nun nacheinander die übrigen drei Bitplanes ausgelesen und behandelt:

```
dec ah ; Nächste Bitplane bearbeiten
jge test1
mov dl, bl ; Ergebnis nach DL
xor dh, dh ; High-Byte löschen
```

Am Ende enthält Register DL den Farbwert. Da der ganze Vorgang doch etwas komplizierter ist, soll folgendes Beispiel beim Verständnis helfen. Wir wollen annehmen, daß die Latch-Register nach einem Lesezugriff auf das zum Grafikpunkt korrespondierende Byte im Video-RAM folgende binären Inhalte aufweisen:

```
Latch 0: 00010110
Latch 1: 00101101
Latch 2: 01111010
Latch 3: 01110111
```

Die Bitmaske soll 00001000 lauten. Nach dem Auslesen von Latch 0 liefert die Verknüpfung mit der Bitmaske folgendes Ergebnis:

```
01110111
and 00001000
-----
00000000
```

Durch das Resultat 00000000 bedingt, liefert der Neg-Befehl ein gelöscht Carry-Flag, das in das unterste Bit von Register BX verschoben wird:

```
BX: ???????0
```

Die Verknüpfung von Latch 1 mit der Bitmaske liefert:

```
01111010
and 00001000
-----
00001000
```

Damit wird in der Folge das Carry-Flag gesetzt und verschoben, für BX sind nun zwei Bit-Positionen bekannt:

```
BX: ??????01
```

Auf die gleiche Weise ergeben die Verknüpfungen von Latch 1 und Latch 0 mit der Bitmaske die Ergebnisse 00001000

und 00000000, so daß BX schließlich den Wert

```
BX: ?????0110
```

aufweist. Die untersten vier Bit stellen den gesuchten Farbwert dar.

Das Demoprogramm »plot.asm« in Listing 5 setzt von links nach rechts und von oben nach unten jeden Grafikpunkt auf dem Bildschirm (insgesamt also 307 200 Punkte) und liest ihn sofort danach wieder aus, so daß insgesamt

```
2*307200=614400
```

Zugriffe erfolgen. Der Quelltext wird einfach mit TASM in eine Objekt-Datei assembliert und mit TLINK in eine Exe-Datei überführt. Parameter außer dem Dateinamen sind weder beim Aufruf des Assemblers noch des Linkers erforderlich.

Nach dem Programmstart werden zunächst die VGA-BIOS-Funktionen, anschließend nach einem Tastendruck die zuvor beschriebenen Routinen benutzt. Ein Vergleich zeigt, daß die direkte Programmierung um mehr als 400 Prozent schneller arbeitet. Noch gravierendere Geschwindigkeitssteigerungen lassen sich bei komplexen Routinen, zum Beispiel zum Zeichnen einer Linie oder Ellipse, erreichen, wenn man von den konventionellen Programmiernalgorithmen abweicht.

(Frank Riemenschneider/bj)

Listing 1: set_read.asm

```
1: mov ah, 0Ch
2: mov al, farbe ; Farbcodes (0-15)
; übergeben
3: mov cx, y ; y-Koordinate (0-479 VGA,
4: ; 0-349 EGA) übergeben
5: mov dx, x ; x-Koordinate (0-639)
6: ; übergeben
7: xor bh, bh
8: int 10h ; Grafikpunkt setzen
9:
10: mov ah, 0Dh
11: mov cx, y ; y-Koordinate (0-479 VGA,
12: ; 0-349 EGA) übergeben
13: mov dx, x ; x-Koordinate (0-639)
14: ; übergeben
15: xor bh, bh
16: int 10h ; Grafikpunkt auslesen
17: mov farbe, al ; Farbcodes (0-15) merken
```

Routinen zum Setzen und Auslesen eines Grafikpunktes mit Hilfe des VGA- oder EGA-BIOS

Listing 2: calcxy.asm

```
1: mov ax, spalte
2: mov bx, zeile
3: mov cx, ax
4: ; Koordinaten nochmal für spätere
5: ; Verwendung in der Berechnung retten
6: mov ax, 80
7: ; Pro Zeile 80 Byte
8: mul bx
9: mov dx, 0A000h
10: ; Offset VGA-Karte laden
11: push dx
12: ; Segmentadresse merken
13: mov dx, ax
14: mov ax, cx
15: shr ax, 1
16: shr ax, 1
17: shr ax, 1
18: ; x/8
19: add dx, ax
20: ; Spaltenoffset addieren
21: mov di, dx
22: ; DI mit Offset laden
23: and cl, 7
24: ; x MOD 8
25: mov dl, 80h
26: ; Maske für Bit 0
27: shr dl, cl
28: ; Maske für gewünschtes Bit durch Rotat
29: ; ion der ursprünglichen Maske erzeugen
```

```
30: pop es
31: ; Segmentadresse Video-RAM laden
32: mov offset, di
33: ; Offsetadresse in Video-RAM
34: mov Bit-Maske, dl
35: ; Bit-Maske für Grafikpunkt
36: ret
```

Berechnung der Video-RAM-Adresse und der Bitmaske eines Grafikpunktes aus den Spalten- und Zeilenkoordinaten

Listing 3: set_dot.asm

```
1: mov ax, spalte
2: ; Spalte (0-639) holen
3: mov bx, zeile
4: ; Zeile (0-479) holen
5: mov cx, farbe
6: ; Farbe holen
7: call CalcXY
8: ; Offset- und Segmentadresse, sowie die
9: ; Bit-Maske aus den X- und Y-Koordinaten
10: ; berechnen
11: mov ah, dl
12: ; Bit-Maske
13: mov dx, 3CEh
14: ; Port-Adresse d VGA-Grafik-Controllers
15: ; laden
16: mov al, 08h
17: ; Bit-Maske im Bit-Mask-Register setzen
18: out dx, ax
19: mov ch, [es:di]
20: ; Latch-Register laden
21: mov [es:di], cl
22: ; Farbe setzen
23: ret
```

Routine zum Setzen eines Grafikpunktes

Listing 4: read_dot.asm

```
1: mov ax, spalte
2: ; Spalte (0-639) holen
3: mov bx, zeile
4: ; Zeile (0-479) holen
5: call CalcXY
6: ; Offset- und Segmentadresse, sowie die
7: ; Bit-Maske aus den X- und
8: ; Y-Koordinaten berechnen
9: mov ch, dl
10: ; Bit-Maske
11: xor bl, bl
12: mov dx, 3CEh
13: ; Port-Adresse des VGA-Grafik-
14: ; Controllers laden
15: mov ax, 0304h
16: ; Bitplane 3 auslesen
17: testl:
18: out dx, ax
19: mov bh, [es:di]
20: ; Latch-Register laden
21: and bh, ch
22: ; Alle Bits außer Bit-Maske ausblenden
23: neg bh
24: ; Bit 7 in Abhängigkeit des Pixels
; setzen
25: rol bx, 1
26: ; Bit in Bit 1 in BL transferieren
27: dec ah
28: ; Nächste Bitplane bearbeiten
29: jge testl
30: mov farbe, bl
31: ; Ergebnis setzen
32: ret
```

Modul zum Auslesen eines Grafikpunktes

Listing 5: plot.asm

```
1: ; Programm: plot.asm
2: ; Funktion: Demo für die Grafikbefehle
3: ; PLOT und TEST
4: ; Sprache : Assembler
5: ; Autor : Frank Riemenschneider
6: ; (c) 1993 DMV Daten- und Medienverlag
7: ;
8:
9: IDEAL
10: ; IDEAL-Modus einschalten
11: MODEL SMALL
12: ; Speichermodell SMALL wählen
13: STACK
14: db 512 dup (?)
15: DATASEG
16: MaxX = 639
17: ; Maximale X-Koordinate
18: MaxY = 479
19: ; Maximale Y-Koordinate,
```

Demoprogramm für die Befehle Plot und Test


```

20: ; bei EGA auf 349 setzen !!!
21: CODESEG
22: ;
23: ; ASMINITGRAPHIC:
24: ; Schaltet die VGA-Karte in den Grafik
25: ; modus und löscht Bildschirm
26: ;
27: ; Eingabe: keine
28: ; Ausgabe: keine
29: PROC AsmInitGraphic
30:     push ax
31:     ; Für alle Fälle alle Register retten
32:     push bx
33:     push cx
34:     push dx
35:     push ds
36:     push di
37:     xor ah, ah
38:     ; Funktionsnummer laden
39:     mov al, 12h
40:     ; Grafikmodus 640 x 480 Punkte in 16
41:     int 10h
42:     ; Farben setzen
43:     mov dx, 3CEh
44:     ; Port-Adresse des VGA-Grafikcontroll
45:     ers laden
46:     mov ax, 0205h
47:     ; Read-Modus 0 und Write-Modus 2
48:     out dx, ax
49:     ; im Modusregister setzen
50:     mov ax, 0003h
51:     ; Verknüpfungsmodus = Ersetzen im
52:     ; Function-Select-Register setzen
53:     out dx, ax
54:     pop di
55:     ; Alle Registerinhalte wieder
56:     ; herstellen
57:     pop ds
58:     pop dx
59:     pop cx
60:     pop bx
61:     pop ax
62:     ret
63:     ; Back home
64: ENDP
65: ;
66: ; ASMINITTEXT:
67: ; Schaltet die VGA-Karte in den
68: ; Textmodus
69: ; Eingabe: keine
70: ; Ausgabe: keine
71: PROC AsmInitText
72:     push ax
73:     ; Für alle Fälle alle Register retten
74:     push bx
75:     push cx
76:     push dx
77:     push ds
78:     push di
79:     mov dx, 3CEh
80:     ; Port-Adresse des VGA-Grafikcontroll
81:     ers laden
82:     mov ax, 0005h
83:     ; Mode- und Function-Select-Register
84:     mit ihren
85:     ; Standardwerten laden
86:     out dx, ax
87:     mov ax, 0003h
88:     out dx, ax
89:     xor ah, ah
90:     ; Funktionsnummer laden
91:     mov al, 3
92:     ; Textmodus 80 x 25 Zeichen in 16 Far
93:     ben setzen
94:     int 10h
95:     pop di
96:     ; Alle Registerinhalte wieder
97:     ; herstellen
98:     pop ds
99:     pop dx
100:    pop cx
101:    pop bx
102:    pop ax
103:    ret
104:    ; Back home
105: ENDP
106: ;
107: ; CALCXY:
108: ; Berechnet Segment und Offset des
109: ; Byte im Video-RAM, sowie die Bit-
110: ; Maske für das passende Bit bei
111: ; gegebener X- und
112: ; Y-Koordinate
113: ; Eingabe: AX = X-Koordinate (0-639)
114: ; BX = Y-Koordinate (0-479)
115: ; Ausgabe: ES = Segmentadresse des
116: ; Bytes im Video-RAM
117: ; DI = Offsetadresse des Byt
118: ; es imVideo-RAM
119: ; DL = Bit-Maske des Bytes
120: ; im Video-RAM
121: PROC CalcXY
122:     push bx
123:     push cx
124:     push si
125:     mov cx, ax
126:     ; Koordinaten nochmal für spätere
127:     ; Verwendung in der Berechnung retten
128:     mov ax, 80
129:     ; Pro Zeile 80 Bytes
130:     mul bx
131:     mov dx, 0A000h
132:     ; Offset VGA-Karte laden
133:     mov es, dx
134:     ; Segmentadresse laden
135:     mov dx, ax
136:     mov ax, cx
137:     shr ax, 1

```

```

138:     shr ax, 1
139:     shr ax, 1
140:     ; x/8
141:     add dx, ax
142:     ; Spaltenoffset addieren
143:     mov di, dx
144:     ; DI mit Offset laden
145:     ;
146:     ; Berechnung der Bit-Maske
147:     and cl, 7
148:     ; x MOD 8
149:     mov dl, 80h
150:     ; Maske für Bit 0
151:     shr dl, cl
152:     ; Maske für gewünschtes Bit durch Rota
153:     ; tion der ursprünglichen Maske erzeugen
154:     pop si
155:     pop cx
156:     pop bx
157:     ret
158:     ; Back home
159: ENDP CalcXY
160: ;
161: ; VGAPLOT:
162: ; Setzt einen Bildpunkt im Video-RAM
163: ; der VGA-Karte
164: ;
165: ; Eingabe: AX=X-Koordinate des Pixels
166: ; BX=Y-Koordinate des Pixels
167: ; DI = Farbe Low-Byte
168: ; Ausgabe: keine
169: PROC AsmPlot
170:     push ax
171:     push bx
172:     push cx
173:     push dx
174:     push si
175:     push di
176:     mov cx, di
177:     ; Farbe retten
178:     call CalcXY
179:     ; Offset- u. Segmentadresse, sowie die
180:     ; Bit-Maske aus d. X- und Y-Koordi
181:     ; naten berechnen
182:     mov ah, dl
183:     ; Bit-Maske
184:     mov dx, 3CEh
185:     ; Port-Adresse d. VGA-Grafikcontrollers
186:     ; laden
187:     mov al, 08h
188:     ; Bit-Maske im Bit-Mask-Register setzen
189:     out dx, ax
190:     mov ch, [es:di]
191:     ; Latch-Register laden
192:     mov [es:di], cl
193:     ; Farbe setzen
194:     pop di
195:     pop si
196:     pop dx
197:     pop cx
198:     pop bx
199:     pop ax
200:     ret
201:     ; Back home
202: ENDP
203: ;
204: ; ASMTSTPIXEL:
205: ; Testet einen Punkt im Grafik-
206: ; Bildschirm
207: ; Eingabe: AX = X-Koordinate (0-639)
208: ; BX = Y-Koordinate (0-479)
209: ; Ausgabe: DX = Farbe des Punktes
210: PROC AsmTestPixel
211:     push ax
212:     push bx
213:     push cx
214:     push si
215:     push di
216:     call CalcXY
217:     ; Offset- u. Segmentadresse, sowie die
218:     ; Bit-Maske aus den X- und
219:     ; Y-Koordinaten berechnen
220:     mov ch, dl
221:     ; Bit-Maske
222:     xor bl, bl
223:     mov dx, 3CEh
224:     ; Port-Adresse des VGA-Grafik-
225:     ; Controllers laden
226:     mov ax, 0304h
227:     ; Bitplane 3 auslesen
228:     testl:
229:         out dx, ax
230:         mov bh, [es:di]
231:     ; Latch-Register laden
232:     and bh, ch
233:     ; Alle Bits außer Bit-Maske ausblenden
234:     neg bh
235:     ; Bit 7 in Abhängigk. d. Pixels setzen
236:     rol bx, 1
237:     ; Bit in Bit 1 in BL transferieren
238:     dec ah
239:     ; Nächste Bitplane bearbeiten
240:     jge testl
241:     mov dl, bl
242:     ; Ergebnis nach AL
243:     xor dh, dh
244:     ; High-Byte löschen
245:     pop di
246:     pop si
247:     pop cx
248:     pop bx
249:     pop ax
250:     ret
251:     ; Back home
252: ENDP
253: ;
254: ; BIOSPLOT:
255: ; Setzt einen Bildpunkt im Video-RAM
256: ; der VGA-Karte
257: ;

```

```

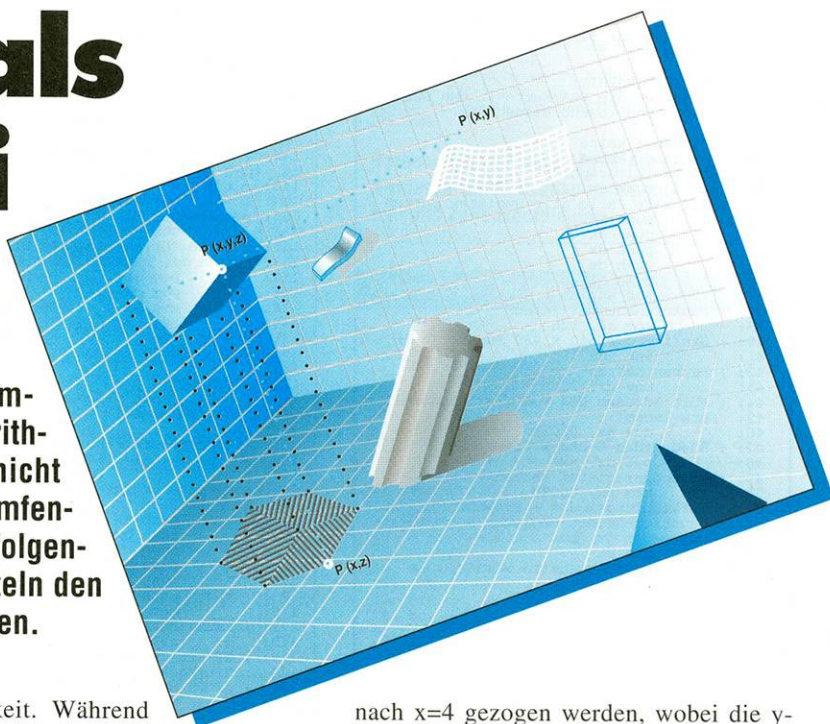
258: ; Eingabe: AX=X-Koordinate des Pixels
259: ; BX=Y-Koordinate des Pixels
260: ; DI=Farbe Low-Byte
261: ; Ausgabe: keine
262: PROC BIOSPlot
263:     push ax
264:     push bx
265:     push cx
266:     push dx
267:     mov cx, ax
268:     mov dx, bx
269:     mov ax, di
270:     mov ah, 0Ch
271:     xor bh, bh
272:     int 10h
273:     pop dx
274:     pop cx
275:     pop bx
276:     pop ax
277:     ret
278: ENDP
279: ;
280: ; BIOTESTPIXEL:
281: ; Testet e. Punkt im Grafik-Bildschirm
282: ;
283: ; Eingabe: AX = X-Koordinate (0-639)
284: ; BX = Y-Koordinate (0-479)
285: ; Ausgabe: DX = Farbe des Punktes
286: PROC BIOTestPixel
287:     push ax
288:     push bx
289:     push cx
290:     mov cx, ax
291:     mov dx, bx
292:     mov ah, 0Dh
293:     xor bh, bh
294:     int 10h
295:     mov dx, ax
296:     pop cx
297:     pop bx
298:     pop ax
299:     ret
300:     ; Back home
301: ENDP
302: GfxStart:
303:     mov ax, @data
304:     mov ds, ax
305:     call AsmInitGraphic
306:     ; Grafikmodus einschalten
307: ;
308: ; Punkt mit VGA-BIOS-Programmierung
309: ;
310: ; setzen/auslesen -----
311: ;
312: ; In oberer, linker Ecke beginnen
313:     xor bx, bx
314:     xor di, di
315:     ; Farbe auf Schwarz setzen
316:     dec di
317: GfxLoop:
318:     inc di
319:     and di, 15
320:     call BIOSPlot
321:     ; Punkt mit VGA-BIOS setzen
322:     call BIOTestPixel
323:     ; Punkt mit VGA-BIOS auslesen
324:     inc ax
325:     cmp ax, MaxX
326:     ; X > MaxX ?
327:     jbe GfxLoop
328:     xor ax, ax
329:     ; Ja, neue Zeile
330:     inc bx
331:     cmp bx, MaxY
332:     ; Y > MaxY ?
333:     jbe GfxLoop
334:     ; Ja, Ende
335:     xor ah, ah
336:     ; Tastatur abfragen
337:     int 16h
338:     call AsmInitGraphic
339:     ; Bildschirm löschen
340: ;
341: ; ----- Punkt mit direkter Register-
342: ; programmierung setzen/auslesen -
343: ;
344: ; In oberer, linker Ecke beginnen
345:     xor bx, bx
346:     xor di, di
347:     ; Farbe auf Schwarz setzen
348:     dec di
349: GfxLoop1:
350:     inc di
351:     and di, 15
352:     call AsmPlot
353:     ; Punkt m. Registerprogr. setzen
354:     call AsmTestPixel
355:     ; Punkt mit Registerprogr. auslesen
356:     inc ax
357:     cmp ax, MaxX
358:     ; X > MaxX ?
359:     jbe GfxLoop1
360:     xor ax, ax
361:     ; Ja, neue Zeile
362:     inc bx
363:     cmp bx, MaxY
364:     ; Y > MaxY ?
365:     jbe GfxLoop1
366:     ; Ja, Ende
367:     xor ah, ah
368:     int 16h
369:     call AsmInitText
370:     ; Textmodus wieder einschalten
371:     mov ax, 4C00h
372:     ; Zurück zu DOS
373:     int 21h
374: END GfxStart

```

Demoprogramm für die Befehle Plot und Test

Schneller als die Polizei erlaubt

Wer schnelle Bildschirmgrafiken programmieren möchte, kommt an effektiven Algorithmen zum Zeichnen von Linien aller Art nicht vorbei, da sie Grundelemente von Bildschirmfenstern, Dialogboxen und Knöpfen sind. Der folgende Beitrag zeigt, wie Sie mit einfachen Mitteln den Grafikaufbau um den Faktor 10 beschleunigen.



Einer der positivsten Gesichtspunkte von Windows 3.1 gegenüber Windows 3.0 ist zweifelsohne der erheblich beschleunigte Bildschirmaufbau. Das einzige Geheimnis hinter diesem Phänomen besteht darin, daß die Entwickler die Routinen zum Zeichnen senkrechter und waagerechter Linien rapide beschleunigten, indem sie einen völlig neuen Algorithmus entwickelten.

Wenn man sich eine grafische Benutzeroberfläche wie Windows ansieht, stellt man fest, daß über 90 Prozent der dargestellten Elemente (Fenster, Dialogboxen, Menüboxen, Knöpfe, Schiebeleisten und so weiter) ausschließlich aus senkrechten und waagerechten Linien bestehen. Es ist daher kein Wunder, daß sich die Beschleunigung beim Linienzeichnen-Algorithmus sofort drastisch bemerkbar macht. Wir wollen uns das Geschwindigkeitsgeheimnis der Windows-Entwickler (das allerdings schon lange vor Windows bekannt war, nur bei Version 3.0 nicht zur Anwendung kam) einmal näher ansehen.

Ein beliebiger Punkt (x,y) auf einer Linie mit den Startkoordinaten (x_1,y_1) und den Endkoordinaten (x_2,y_2) läßt sich mathematisch durch den Ausdruck

$$y = y_1 + Dy/Dx * (x-x_1)$$

mit

$$Dy = y_2 - y_1$$

und

$$Dx = x_2 - x_1$$

beschreiben, wobei der erste Ausdruck die Steigung angibt (Bild 1). Ein Problem der Computergrafik besteht darin, daß, anders als auf dem Papier, mit diskreten Werten gerechnet werden muß. Angenommen, der Quotient Dy/Dx würde den Wert 3,5 und x den Wert 3 annehmen, so ergäbe sich für y theoretisch der Wert 10,5. Die Computergrafik läßt jedoch nur die Koordinaten 10 oder 11 zu. Bild 2 verdeutlicht die

Schwierigkeit. Während auf dem Papier eine Linie kontinuierlich gezeichnet ist, bildet sich auf dem Bildschirm, durch die diskreten Werte bedingt, eine Art Treppmuster, wobei man die abbildbaren Werte möglichst nah an die tatsächlichen Ergebnisse annähern muß. Für diese Aufgabenstellung existieren zahlreiche Algorithmen. Einer der bekannten Vertreter darunter ist der Bresenham-Algorithmus, auf den wir am Ende des Beitrags nochmals eingehen werden.

Wenn Sie sich die zuvor erwähnte Formel ansehen, erkennen Sie zwei Sonderfälle: Bei einer waagerechten Linie ist $y_2=y_1$ und die Steigung beträgt 0, so daß für alle x der y -Wert immer y_1 ($=y_2$) annimmt. Bei einer senkrechten Linie ($x_1=x_2$) wird die Steigung unendlich groß. Wenn Sie sich den ersten Kursteil der Grafikprogrammierung bezüglich des Aufbaus des Video-RAMs nochmals in Erinnerung rufen, werden Sie zustimmen, daß man für diese beiden mathematischen Sonderfälle, die »zufällig« auch noch am häufigsten anzutreffen sind, effektivere Algorithmen als diejenigen, die nach der allgemeinen Formel arbeiten, finden kann. Lassen Sie uns mit den waagerechten Linien beginnen. Als Beispiel soll eine Linie von $x=6$

nach $x=4$ gezogen werden, wobei die y -Koordinate beliebig sein soll. Stellt man sich die Linie im Bytemuster des Video-RAMs vor (Bild 3), fällt auf, daß praktisch nur drei verschiedene Inhalte der Grafikbytes vorkommen: ein Bitmuster am linken Rand, eines am rechten und dazwischen nur Bitmasken mit dem Inhalt 255, da alle Bits dieser Bytes gesetzt sind. Der Trick besteht nun darin, statt wie bei einer »gewöhnlichen« Linie, bei der jeder Punkt einzeln gezeichnet und deshalb die Adresse im Video-RAM sowie die Bitmaske jedesmal neu berechnet wird, gleich mehrere Punkte auf einmal zu behandeln.

Für den linken beziehungsweise rechten Rand existieren jeweils nur acht Bitmasken, die davon abhängen, wieviele Bits eines Bytes im Video-RAM durch den Rand

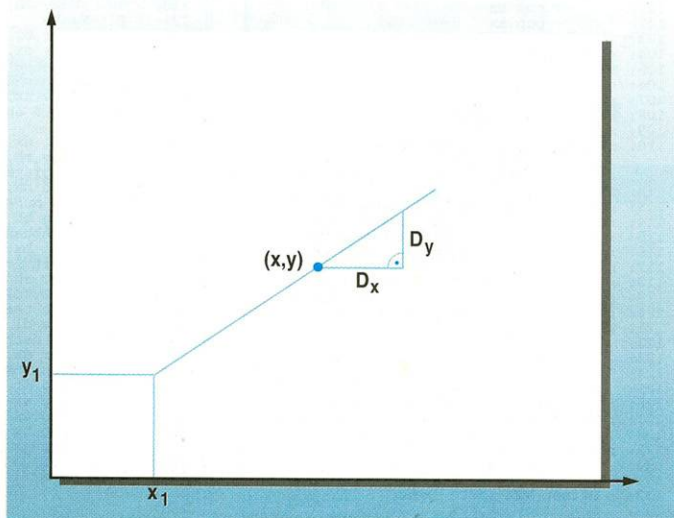


Bild 1. Zusammenhang zwischen Punkten im Koordinatensystem und der Steigung. Dx und Dy bestimmen dabei die Entfernung zum Zeitpunkt.

belegt sind (Bild 4). In Abhängigkeit der x-Koordinate vom Linienstart- und -endpunkt braucht man daher nur jeweils den zutreffenden Wert aus einer Tabelle auszulesen und in das Bitmask-Register des Grafik-Controllers zu schreiben.

Bei den Bytewerten zwischen den Rändern ist die Sache noch einfacher: Da hier alle Bits gesetzt sind, ist immer nur der Wert 255 ins Bitmask-

Register einzutragen. Das Verfahren funktioniert in der Praxis daher wie folgt: Man berechnet mit dem im ersten Kursteil vorgestellten Programmmodul *calcx* die Adressen im Video-RAM, die zu dem Linienstart- und -endpunkt korrespondieren. *calcx* liefert dabei auch eine Bitmaske für diese Punkte zurück. Die Aufgabe besteht nun darin, diese Bitmaske in einen Index für die Tabellen mit den Bitmasken für den linken und rechten Rand umzuwandeln. Eine einfache Lösung stellt folgende Befehlssequenz dar, welche voraussetzt, daß sich die Bitmaske im Register DH befindet:

```
xor bx,bx
loop:
shr dh, 1
inc bl
jnc loop
```

Der Index steht anschließend im Register CL zur Verfügung. Mit der Anweisungsfolge

```
mov ah, [BYTE StartTab+bx]
beziehungsweise
```

```
mov ah, [BYTE EndTab+bx]
```

läßt sich anschließend einfach die Bitmaske der Ränder aus einer Tabelle auslesen. Ein Sonderfall tritt bei kurzen Linien auf, die insgesamt nur ein Byte im Video-RAM belegen. Ein Beispiel dafür wäre eine Linie zwischen den x-Koordinaten 9 und 13, da sich die x-Koordinaten 8 bis 15 im Video-RAM an derselben Adresse befinden. In dieser Situation fallen zunächst einmal alle Zwischenwerte mit der Bitmaske 255 weg. Des weiteren muß man jedoch auch noch die Rand-Bitmasken modifizieren, da – wie in Bild 4 ersichtlich – immer der gesamte Bereich bis zum rechten beziehungsweise linken Rand des Bytes abgedeckt ist. Wenn man aber beide Rand-Bitmasken durch eine logische AND-Verknüpfung miteinander verbindet, bleiben nur die Bits stehen, die in beiden Bitmasken vorkommen (Bild 5). In diesem Fall ist nur ein einziges Byte im Video-RAM zu modifizieren.

Dieses Verfahren kann gegenüber den herkömmlichen Techniken einen Geschwin-

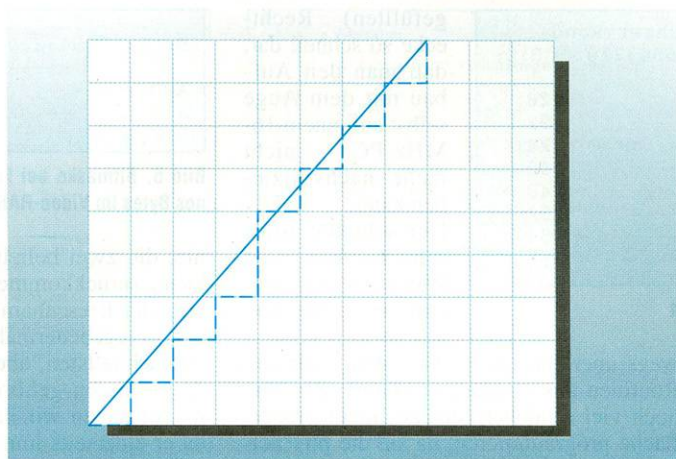


Bild 2. Eine auf Papier gezeichnete Linie erscheint kontinuierlich, während die Ganzzahlkoordinaten auf dem Bildschirm eine Art Treppemuster bewirken.

digkeitszuwachs von über 7000 Prozent erbringen, es ist der Standardprogrammierung also 70fach überlegen. Selbst gegenüber den schnellsten bekannten Grafikroutinen von Hochsprachen-Compilern wie Turbo Pascal wird noch eine Steigerung von 1000 Prozent erzielt.

Nehmen Sie als Beispiel eine Linie von $x=0$ bis $x=639$. Beim herkömmlichen Verfahren muß das Calcx-Modul, das den zweitaufwendigsten Anteil an der Plot-Routine beinhaltet, 640mal aufgerufen

jetzt aber tatsächlich waagerechte Linien zum Einsatz kommen (die y-Koordinaten bei $x=0$ und $x=639$ stimmen überein). Das einzige, was man jetzt noch erkennen kann, ist ein hektisches Bildschirmflackern, nach einer halben Sekunde ist alles vorbei. Anders ausgedrückt: In einer halben Sekunde wurden

$$15 \cdot 480 = 7200$$

Linien gezeichnet oder

$$15 \cdot 480 \cdot 640 = 4608000$$

Grafikpunkte gesetzt. Das entspricht pro Sekunde annähernd zehn Millionen Punkten in 16 Farben!

Kommen wir nun zu den vertikalen Linien. Da die x-Koordinate konstant bleibt, muß nur eine einzige Bitmaske berechnet werden, die für alle Zeilen gültig ist. Man braucht daher nur die Startadresse im Video-RAM zu ermitteln und kann dann jeweils entsprechend oft 80 zu dieser Adresse addieren (jede Zeile mit 640 Punkten enthält

$$640 / 8 = 80$$

Byte), wie es durch die Differenz der y-Koordinaten ($y_2 - y_1$) bestimmt wird. Das Listing zeichnet wiederum 15 Bildschirme, füllt sie jedoch diesmal mit vertikalen Linien (von $y=0$ bis $y=479$).

Zunächst wählen wir die x-Koordinate des Endpunktes bei $y=479$ immer um eine Einheit größer als die bei der x-Koordinate des Startpunktes ($y=0$), um die Routine

x	111111	11112222	22222233	33333333
44444444				
	01234567	89012345	67890123	45678901
01234567				
Adress-Offset	?	?+1	?+2	?+3
?+5				?+4
Bits	76543210	76543210	76543210	76543210

Bild 3. Waagerechte Linie als Bitmuster im Video-RAM

werden. Des weiteren sind auch das Bitmask-Register und die Latch-Register 640fach neu zu laden. Bei der eben vorgestellten Methode wird *calcx* jedoch nur zweimal aufgerufen. Das Bitmask-Register lädt der Algorithmus genau dreimal neu (einmal für den linken, einmal für den rechten Rand sowie einmal die Bitmaske 255 für die Zwischenräume) und die Latch-Register nur

$$640 / 8 = 80$$

mal. Das Ergebnis veranschaulicht das Programm »line.asm« (Listing).

Anfangs füllen wir den Bildschirm mit 480 nur fast waagerechten Linien (die y-Koordinate des Endpunktes bei $x=639$ ist immer um 1 größer als die y-Koordinate am Startpunkt $x=0$), damit die Prozedur zum Zeichnen einer allgemeinen Linie verwendbar bleibt. Nach der Füllung eines Bildschirms wiederholt sich der Vorgang mit einer anderen Farbe, insgesamt 15 Durchgänge. Auf einem 486er PC mit 33 MHz und Trident-VGA-Karte benötigen die Bildschirmfüllungen 34,5 Sekunden. Nach einem Tastendruck wird die gleiche Prozedur nochmals durchgeführt, wobei

allgemeingültig zu halten. Wie erwartet benötigt das Modul genauso lange wie bei den waagerechten Linien, da hier wie dort punktweise operiert wurde. Unsere optimierte Routine nimmt mit zwölf Sekunden allerdings erheblich mehr Zeit in Anspruch als die Variante für waagerechten Linien und ist damit »nur« 300 Prozent schneller bezüglich des Standardgrafikaufbaus einer konventionell programmierten Routine.

Die Erklärung der Geschwindigkeitsverringerung bei vertikalen Linien gegenüber den waagerechten liegt im Aufbau des Video-RAMs begründet. Da bei senkrechten Linien pro Byte im Video-RAM jeweils nur ein Bit gesetzt werden kann, sind schon allein aus diesem Grund achtmal mehr Schreibzugriffe erforderlich (bei der waagerechten Linie konnten wir bei einem Zugriff bis zu acht Punkte gleichzeitig setzen).

Wenn man die Prozeduren für waagerechte und senkrechte Linien programmiert hat, ist es auch nicht mehr schwer, Funktionen für Rechtecke beziehungsweise ausgefüllte Rechtecke zu schreiben. Bei den ausgefüll-

Bits	linker Rand	Bitmaske	rechter Rand	Bit-
maske	76543210		76543210	
-----	255	-	128	
-----	127	--	192	
-----	63	---	224	
-----	31	----	240	
-----	15	-----	248	
-----	7	-----	252	
-----	3	-----	254	
-----	1	-----	255	

Bild 4. Bitmasken für den linken und rechten Linienrand

ten Rechtecken wäre theoretisch eine Entscheidung bezüglich des Füllens mit senkrechten oder waagerechten Linien erforderlich. Da die waagerechten Linien bis zu 25mal schneller zu erzeugen sind, hat sich die Auswahl damit erledigt.

Ein nicht gefülltes Rechteck ist noch einfacher zu beschreiben, da es lediglich aus zwei senkrechten und zwei waagerechten Linien besteht. »line.asm« stellt die (aus-

wegs unerklärlich. Im Gegenteil: Mit den Routinen aus »line.asm« können Sie eine noch viel schnellere eigene Benutzeroberfläche programmieren, da sie die physikalisch maximal realisierbare Geschwindigkeit beim Grafikaufbau aus Ihrem PC herausholen.

Lassen Sie uns abschließend die Sonderfälle waagerechte und senkrechte Linie verlassen und zu einer gewöhnlichen Li-

gefüllten) Rechtecke so schnell dar, daß man den Aufbau mit dem Auge selbst auf einem 16-MHz-PC nicht mehr nachvollziehen kann.

Der schnelle Bildschirmaufbau von Windows 3.1. gegenüber Version 3.0 ist also keines-

Bits	76543210	linker Rand
	-----	rechter Rand
AND	-----	Gesamtlinie
=	--	

Bild 5. Bitmaske bei kurzen Linien innerhalb eines Bytes im Video-RAM

nie, die zwei beliebige Punkte verbinden kann, zurückkommen. Anfangs wurde bereits der Bresenham-Algorithmus erwähnt, der zugegebenermaßen zwar nicht zu den allerschnellsten, aber dafür kompaktesten Algorithmen gehört. Für das Programmmodul haben wir ihn bewußt ausgewählt, da er einerseits immer noch schnell genug ist, Linien für das menschliche Auge unsichtbar zu zeichnen (Listing), und andererseits schräge Linien im Vergleich zu waagerechten und senkrechten Linien nur sehr selten auftreten.

(Frank Riemenschneider/bj)

Listing: line.asm

```

1: ;Programm: line.asm
2: ;Funktion: Demo-Programm für die
3: ;Grafikbefehle
4: ; LINE, BAR und BOX
5: ;Sprache : Assembler
6: ;Autor : Frank Riemenschneider
7: ;(c) 1993 DMV Daten- und Medienverlag
8: ;
9:
10: IDEAL
11: ; IDEAL-Modus einschalten
12: MODEL SMALL
13: ; Speichermodell SMALL wählen
14: STACK
15: ; db 512 dup (?)
16: DATASEG
17: MaxX = 639
18: ; Maximale X-Koordinate
19: MaxY = 479
20: ; Maximale Y-Koordinate
21: ; bei EGA auf 349 setzen !!!
22: MinX = 0
23: ; Minimale X-Koordinate
24: MinY = 0
25: ; Minimale Y-Koordinate
26: MaxColor = 15
27: ; Größtmöglicher Farbwert
28: ; (1 für Hercules, 15 für EGA)
29: MinColor = 0
30: ; Kleinstmöglicher Farbwert
31: xStep = 1
32: ; Schrittweite der X-Koordinate
33: yStep = 1
34: ; Schrittweite der Y-Koordinate
35: cStep = 1
36: ; Schrittweite des Farbwertes
37: xx dw 100
38: ; X-Koordinate
39: yy dw 100
40: ; Y-Koordinate
41: Color db 1
42: ; Farbe
43: xInc dw xStep
44: ; Inkrement der X-Koordinate
45: yInc dw yStep
46: ; Inkrement der Y-Koordinate
47: cInc db cStep
48: ; Inkrement des Farbwertes
49: CODESEG
50: ;
51: ; ASMINITGRAPHIC:
52: ; Schaltet die VGA-Karte in d. Grafik
53: ; modus und löscht Bildschirm
54: ;
55: ; Eingabe: keine
56: ; Ausgabe: keine
57: PROC AsmInitGraphic
58: ; push ax
59: ; Für alle Fälle alle Register retten
60: ; push bx
61: ; push cx
62: ; push dx
63: ; push si
64: ; push di
65: ; xor ah, ah
66: ; Funktionsnummer laden
67: ; mov al, 12h
68: ; Grafikmodus 640 x 480 Punkte in 16
69: ; int 10h
70: ; Farben setzen
71: ; mov dx, 3CEh
72: ; Port-Adresse des VGA-Grafik-
73: ; Controllers laden

```

```

74: ; mov ax, 0205h
75: ; Read-Modus 0 und Write-Modus 2
76: ; out dx, ax
77: ; im Modusregister setzen
78: ; mov ax, 0003h
79: ; Verknüpfungsmodus = Ersetzen im
80: ; Function-Select-Register setzen
81: ; out dx, ax
82: ; pop di
83: ; Alle Registerinhalte wieder
84: ; herstellen
85: ; pop ds
86: ; pop dx
87: ; pop cx
88: ; pop bx
89: ; pop ax
90: ; ret
91: ; Back home
92: ENDP
93: ;
94: ; ASMINITTEXT:
95: ; Schaltet die VGA-Karte in den Text-
96: ; modus
97: ; Eingabe: keine
98: ; Ausgabe: keine
99: PROC AsmInitText
100: ; push ax
101: ; Für alle Fälle alle Register retten
102: ; push bx
103: ; push cx
104: ; push dx
105: ; push ds
106: ; push di
107: ; mov dx, 3CEh
108: ; Port-Adresse des VGA-Grafik-
109: ; controllers laden
110: ; mov ax, 0005h
111: ; Mode- und Function-Select-Register
112: ; mit ihren
113: ; Standardwerten laden
114: ; out dx, ax
115: ; mov ax, 0003h
116: ; out dx, ax
117: ; xor ah, ah
118: ; Funktionsnummer laden
119: ; mov al, 3
120: ; Textmodus 80 x 25 Zeichen in 16
121: ; Farben setzen
122: ; int 10h
123: ; pop di
124: ; Alle Registerinhalte wieder
125: ; herstellen
126: ; pop ds
127: ; pop dx
128: ; pop cx
129: ; pop bx
130: ; pop ax
131: ; ret
132: ; Back home
133: ENDP
134: ;
135: ; CALCXY:
136: ; Berechnet Segment und Offset d Byte
137: ; im Video-RAM, sowie d Bit-Maske für
138: ; das passende Bit bei gegebener X-
139: ; und Y-Koordinate
140: ;
141: ; Eingabe: AX = X-Koordinate (0-639)
142: ; BX = Y-Koordinate (0-479)
143: ; Ausgabe: ES=Segmentadresse des Byte
144: ; im Video-RAM
145: ; DI = Offsetadresse des Byte im
146: ; Video-RAM
147: ; DL = Bit-Maske des Byte im
148: ; Video-RAM

```

```

149: PROC CalcXY
150: ; push bx
151: ; push cx
152: ; push si
153: ; mov cx, ax
154: ; Koordinaten nochmal für spätere
155: ; Verwendung in der Berechnung retten
156: ; mov ax, 80
157: ; Pro Zeile 80 Bytes
158: ; mul bx
159: ; mov dx, 0A000h
160: ; Offset VGA-Karte laden
161: ; mov es, dx
162: ; Segmentadresse laden
163: ; mov dx, ax
164: ; mov ax, cx
165: ; shr ax, 1
166: ; shr ax, 1
167: ; shr ax, 1
168: ; x/8
169: ; add dx, ax
170: ; Spaltenoffset addieren
171: ; mov di, dx
172: ; DI mit Offset laden
173: ;
174: ; Berechnung der Bit-Maske
175: ; and cl, 7
176: ; x MOD 8
177: ; mov dl, 80h
178: ; Maske für Bit 0
179: ; shr dl, cl
180: ; gewünschtes Bit durch Rotation
181: ; der ursprünglichen Maske erzeugen
182: ; pop si
183: ; pop cx
184: ; pop bx
185: ; ret
186: ; Back home
187: ENDP CalcXY
188: ;
189: ; VGAPLOT:
190: ; Setzt einen Bildpunkt im Video-RAM
191: ; der VGA-Karte
192: ;
193: ; Eingabe: AX=X-Koordinate des Pixels
194: ; BX=Y-Koordinate des Pixels
195: ; DI=Farbe Low-Byte
196: ; Ausgabe: keine
197: PROC AsmPlot
198: ; push ax
199: ; push bx
200: ; push cx
201: ; push dx
202: ; push si
203: ; push di
204: ; mov cx, di
205: ; Farbe retten
206: ; call CalcXY
207: ; Offset- und Segmentadresse, sowie
208: ; die Bit-Maske aus den X- und
209: ; Y-Koordinaten berechnen
210: ; mov ah, dl
211: ; Bit-Maske
212: ; mov dx, 3CEh
213: ; Port-Adresse des VGA-Grafik-
214: ; Controllers laden
215: ; mov al, 08h
216: ; Bit-Maske im Mask-Register setzen
217: ; out dx, ax
218: ; mov ch, [es:di]

```

Demoprogramm für die Grafikbefehle Line, Bar und Box


```

219: ; Latch-Register laden
220: mov [es:di], cl
221: ; Farbe setzen
222: pop di
223: pop si
224: pop dx
225: pop cx
226: pop bx
227: pop ax
228: ret
229: ; Back home
230: ENDP
231: ;
232: ; VGAHLINe:
233: ; Zieht eine horizontale Linie im
234: ; Grafik-Bildschirm der VGA-Karte
235: ;
236: ; Eingabe: AX=X-Koordinate (0-639)
237: ; des Startpunktes
238: ; BX=Y-Koordinate (0-479) des
239: ; Startpunktes
240: ; CX=X-Koordinate (0-639) des
241: ; Endpunktes
242: ; DX=Y-Koordinate (0-479) des
243: ; Endpunktes
244: ; DI=Farbe Low-Byte
245: ; Ausgabe: keine
246: DATASEG
247: StartTab db 1,3,7,15,31,63,127,255
248: EndTab db 255,254,252,248,240,224,
192,128
249: CODESEG
250: PROC VGAHLINe
251: push ax
252: ; X- und Y-Koordinaten für spätere
253: ; Befehle retten
254: push bx
255: push cx
256: push dx
257: push si
258: push di
259: ;
260: ; Farbe wird später noch gebraucht
261: ;
262: ; Adressen holen
263: push cx
264: ; Endkoordinaten und Karte retten
265: push dx
266: push di
267: call CalcXY
268: ; Anfangsadresse aus den Koordinaten
269: ; berechnen
270: mov si, di
271: ; Anfangsadressenoffset retten
272: pop di
273: ; Endkoordinaten u. Karte wied. laden
274: pop bx
275: pop ax
276: mov cx, dx
277: ; Anfangs-Bitmaske retten
278: call CalcXY
279: ; Endadresse aus Koordinaten berechnen
280: mov dh, cl
281: ; Anfangs-Bitmaske zurückschreiben
282: ;
283: ; Anfangs- und Endbitmuster bestimmen
284: ;
285: ; XOR CX, CX
286: ; Zähler initialisieren
287: VGABitStart:
288: shr dh, 1
289: ; Anfangs-Bitmaske rotieren
290: inc ch
291: ; Zähler erhöhen
292: jnc VGABitStart
293: ; Solange rotieren, bis das gesetzte
294: ; Bit im Carry ist
295: dec ch
296: ; Zähler an Bitnummer anpassen
297: VGABitEnd:
298: shr dl, 1
299: ; End-Bitmaske rotieren
300: inc cl
301: ; Zähler erhöhen
302: jnc VGABitEnd
303: ; Solange rotieren, bis das gesetzte
304: ; Bit im Carry ist
305: dec cl
306: ; Zähler an Bitnummer anpassen
307: xor bx, bx
308: mov bl, ch
309: ; Indexregister laden
310: mov ah, [BYTE StartTab+bx]
311: ; Start-Bitmuster laden
312: mov bl, cl
313: ; Indexregister laden
314: mov al, [BYTE EndTab+bx]
315: ; End-Bitmuster laden
316: pop cx
317: ; Farbe laden
318: mov dx, 3CEh
319: ; Port-Adresse des VGA-Grafik-
320: ; controllers laden
321: cmp si, di
322: ; Anfangs- und Endadresse gleich?
323: jne VGADifferent
324: ; Nein -> Zwischenbytes setzen
325: ;
326: ; Anfangs- und Endadresse sind gleich
327: and ah, al
328: ; Anfangs- u. End-Bitmuster AND-verkn.
329: mov al, 08h
330: ; Bit-Maske i Bit-Mask-Register setzen
331: out dx, ax
332: mov bl, [es:di]
333: ; Latch-Register laden
334: mov [es:di], cl
335: ; Linie setzen
336: jmp VGAHLINeExit

```

```

336: ; Beenden
337: ;
338: ; Anfangs- und Endadresse s. versch.
339: VGADifferent:
340: mov ch, al
341: ; End-Bitmuster retten
342: mov al, 08h
343: ; Bit-Maske im Mask-Register setzen
344: out dx, ax
345: mov bl, [es:si]
346: ; Latch-Register laden
347: mov [es:si], cl
348: ; Farbe im Video-RAM setzen
349: inc si
350: ; Zeiger erhöhen
351: cmp si, di
352: ; Endadresse erreicht?
353: jne VGAHLINeLab1
354: ; Nein -> weiter hinten
355: mov ah, ch
356: ; End-Bitmuster laden
357: out dx, ax
358: ; Bit-Maske im Mask-Register setzen
359: mov bl, [es:si]
360: ; Latch-Register laden
361: mov [es:si], cl
362: ; Farbe im Video-RAM setzen
363: jmp VGAHLINeExit
364: ; Beenden
365: VGAHLINeLab1:
366: mov ah, 0FFh
367: ; Zwischen-Bitmuster laden
368: ; und im Bit-Mask-Register setzen
369: out dx, ax
370: VGAHLINeLab2:
371: mov bl, [es:si]
372: ; Latch-Register laden
373: mov [es:si], cl
374: ; Farbe im Video-RAM setzen
375: inc si
376: ; Zeiger erhöhen
377: cmp si, di
378: ; Endadresse erreicht?
379: jne VGAHLINeLab2
380: ; Nein -> weitere Bytes löschen
381: mov ah, ch
382: ; End-Bitmuster laden,
383: out dx, ax
384: ; im Bit-Mask-Register
385: mov bl, [es:si]
386: ; Latch-Register laden
387: mov [es:si], cl
388: ; Farbe im Video-RAM setzen
389: VGAHLINeExit:
390: pop di
391: pop si
392: pop dx
393: pop cx
394: pop bx
395: ; X- und Y-Koord. wieder restaurieren
396: pop ax
397: ret
398: ; Back home
399: ENDP VGAHLINe
400: ;
401: ; VGAVLINe:
402: ; Zieht eine vertikale Linie im
403: ; Grafik-Bildschirm der VGA-Karte
404: ;
405: ; Eingabe: AX=X-Koordinate (0-639)
406: ; des Startpunktes
407: ; BX=Y-Koordinate (0-479)
408: ; des Startpunktes
409: ; CX=X-Koordinate (0-639)
410: ; des Endpunktes
411: ; DX=Y-Koordinate (0-479)
412: ; des Endpunktes
413: ; Ausgabe: keine
414: CODESEG
415: PROC VGAVLINe
416: push ax
417: ; X- und Y-Koordinaten für spätere
418: ; Befehle retten
419: push bx
420: push cx
421: push dx
422: push si
423: push di
424: mov bp, 0A000h
425: mov es, bp
426: ; Segmentadresse laden
427: shr ax, 1
428: ; x/8 (Es ist schneller den Befehl 3x
429: ; abzuarbeiten als die Rotation in
430: ; einem Befehl)
431: shr ax, 1
432: shr ax, 1
433: ; Berechnung der Bit-Maske
434: and cl, 7
435: ; x MOD 8
436: mov ch, 80h
437: ; Maske für Bit 0
438: shr ch, cl
439: ; Maske f. gewünschtes Bit durch
440: ; Rotat.
441: push dx
442: ; Endkoordinate retten
443: push ax
444: ; X-Koordinate retten
445: mov dx, 3CEh
446: ; Port-Adresse des VGA-Grafik-
447: ; controllers laden
448: mov ah, ch
449: ; Bit-Maske im Mask-Register setzen
450: mov al, 08h
451: out dx, ax
452: mov cx, di

```

```

453: ; Farbe in CL laden
454: mov ax, 80
455: ; Zeilenanfang berechnen
456: mul bx
457: mov di, ax
458: pop ax
459: pop dx
460: add di, ax
461: ; Spaltenoffset addieren
462: VGAVLINeLoop:
463: mov ch, [es:di]
464: ; Latch-Register laden
465: mov [es:di], cl
466: ; Punkt setzen
467: add di, 80
468: ; Nächste Zeile
469: inc bx
470: ; Y-Koordinate erhöhen
471: cmp bx, dx
472: ; Endpunkt erreicht?
473: jbe VGAVLINeLoop
474: ; Weitere Punkte setzen
475: VGAVLINeExit:
476: pop di
477: pop si
478: pop dx
479: pop cx
480: pop bx
481: ; X- und Y-Koordinaten wieder restaur.
482: pop ax
483: ret
484: ; Back home
485: ENDP VGAVLINe
486: ;
487: ; ASMLINe:
488: ; Zieht Linie im Grafik-Bildschirm
489: ;
490: ; Eingabe: AX=X-Koordinate (0-639)
491: ; des Startpunktes
492: ; BX=Y-Koordinate (0-479)
493: ; des Startpunktes
494: ; CX=X-Koordinate (0-639)
495: ; des Endpunktes
496: ; DX=Y-Koordinate (0-479)
497: ; des Endpunktes
498: ; DI=Farbe Low-Byte
499: ; Ausgabe: keine
500: DATASEG
501: DeltaX dw ?
502: ; Steigung in X-Richtung
503: DeltaY dw ?
504: ; Steigung in Y-Richtung
505: SignY dw ?
506: ; Vorzeichen Y-Richtung
507: Incl dw ?
508: ; Inkrement 1
509: Inc2 dw ?
510: ; Inkrement 2
511: CODESEG
512: PROC AsmLine
513: push ax
514: push bx
515: push cx
516: push dx
517: push si
518: push di
519: push bp
520: cmp bx, dx
521: ; Horizontale Linie?
522: jne LineLabA
523: jmp HLine
524: LineLabA:
525: cmp ax, cx
526: ; Vertikale Linie?
527: jne LineLabB
528: jmp VLine
529: LineLabB:
530: mov [DeltaX], cx
531: ; Steigung in X-Richtung berechnen
532: mov bp, 1
533: ; positives Vorzeichen annehmen
534: sub [DeltaX], ax
535: jg LineLab9
536: ; Steigung negativ?
537: neg [DeltaX]
538: ; Ja -> DeltaX komplementieren
539: mov bp, -1
540: ; negatives Vorzeichen in X-Richtung
541: LineLab9:
542: mov [DeltaY], dx
543: ; Steigung in Y-Richtung berechnen
544: mov [SignY], 1
545: ; positives Vorzeichen annehmen
546: sub [DeltaY], bx
547: jg LineLab1
548: ; Steigung negativ?
549: neg [DeltaY]
550: ; Ja -> DeltaY komplementieren
551: mov [SignY], -1
552: ; negatives Vorzeichen in Y-Richtung
553: LineLab1:
554: push ax
555: ; Koordinate retten, um Register zum
556: ; Arbeiten freizubekommen
557: mov ax, [DeltaX]
558: cmp ax, [DeltaY]
559: ; Größe der Steigungen in X- und
560: ; Y-Richtung vergleichen
561: pushf
562: ; Ergebnis merken
563: jge LineLab2
564: ; Ja -> Steigungen nicht tauschen

```

Demoprogramm für die Grafikbefehle Line, Bar und Box


```

565:      mov ax, [DeltaX]
566: ; Steigungen vertauschen
567:      xchg [DeltaY], ax
568:      mov [DeltaX], ax
569: LineLabl2:
570:      mov ax, [DeltaY]
571: ; Steigung Y-Richtung laden,
572:      shl ax, 1
573: ; mit 2 multiplizieren,
574:      sub ax, [DeltaX]
575: ; davon Steigung X-Richtung abziehen
576:      mov si, ax
577: ; Ergebnis ist der Startwert
578:      mov ax, [DeltaX]
579: ; Inkrement 2 berechnen
580:      shl ax, 1
581: ; als Doppeltes d. Steigung X-Richtung
582:      mov [Inc2], ax
583:      mov ax, [DeltaY]
584: ; Inkrement 1 berechnen
585:      shl ax, 1
586: ; als Doppeltes d. Steigung Y-Richtung
587:      mov [Inc1], ax
588:      popf
589: ; Flags zurückholen
590:      jnge gr45grad
591: ; Algorithmus für Steigungen > 45 Grad
592: ;
593: ; Bresenham-Algorithmus
594: ; für Steigungen <= 45 Grad
595:      pop ax
596: ; X-Koordinate Startpunkt wiederherst.
597:      call AsmPlot
598: ; Punkt setzen
599: LineWhileLoop:
600:      cmp ax, cx
601: ; X-Koordinate Endpunkt erreicht?
602:      jne LineLabl3
603: ; Nein -> weiter plotten
604:      cmp bx, dx
605: ; Y-Koordinate Endpunkt erreicht?
606:      jne LineLabl3
607: ; Nein -> weiter plotten
608:      jmp LineExit
609: ; Ja -> Beenden
610: LineLabl3:
611:      cmp si, 0
612: ; Startwert >= 0?
613:      jl LineLabl6
614: ; Nein -> weiter hinten
615:      add bx, [SignY]
616: ; Y-Koordinate erhöhen
617:      sub si, [Inc2]
618: ; Startwert um Inkrement 2 erniedrigen
619: LineLabl6:
620:      add ax, bp
621: ; X-Koordinate erhöhen
622:      add si, [Inc1]
623: ; Startwert um Inkrement 1 erhöhen
624:      call AsmPlot
625: ; Punkt setzen
626:      jmp LineWhileLoop
627: ; Zurück zum Schleifenanfang
628: ;
629: ; Bresenham-Algorithmus für Steigung > 45 Grad
630: ; n > 45 Grad
631: gr45grad:
632:      pop ax
633: ; X-Koordinate Startpunkt wiederherst.
634:      call AsmPlot
635: ; Punkt setzen
636: _LineWhileLoop:
637:      cmp ax, cx
638: ; X-Koordinate Endpunkt erreicht?
639:      jne _LineLabl3
640: ; Nein -> weiter plotten
641:      cmp bx, dx
642: ; Y-Koordinate Endpunkt erreicht?
643:      jne _LineLabl3
644: ; Nein -> weiter plotten
645:      jmp LineExit
646: ; Ja -> Beenden
647: _LineLabl3:
648:      cmp si, 0
649: ; Startwert >= 0?
650:      jl _LineLabl6
651: ; Nein -> weiter hinten
652:      add ax, bp
653: ; X-Koordinate erhöhen
654:      sub si, [Inc2]
655: ; Startwert um Inkrement 2 erniedrigen
656: _LineLabl6:
657:      add bx, [SignY]
658: ; Y-Koordinate erhöhen
659:      add si, [Inc1]
660: ; Startwert um Inkrement 1 erhöhen
661:      call AsmPlot
662: ; Punkt setzen
663:      jmp _LineWhileLoop
664: ; Zurück zum Schleifenanfang
665: ;
666: ; Vertikale Linie ziehen
667: VLine:
668:      cmp bx, dx
669: ; YAnf > YEnde?
670:      jle VLineLabl
671:      xchg bx, dx
672: VLineLabl:
673:      call VGAVLine
674: ; VGA: Vertikale Linie ziehen
675:      jmp LineExit
676: ;
677: ; Horizontale Linie ziehen
678: HLine:
679:      cmp ax, cx
680: ; XAnf > XEnde?
681:      jle HLineLabl

```

```

682:      xchg ax, cx
683: HLineLabl:
684:      call VGAVLine
685: ; VGA: Horizontale Linie ziehen
686: LineExit:
687:      pop bp
688:      pop di
689:      pop si
690:      pop dx
691:      pop cx
692:      pop bx
693:      pop ax
694:      ret
695: ; Back home
696: ENDP
697: ;
698: ; ASMBBOX:
699: ; Zeichnet Rechteck i Grafikbildschirm
700: ;
701: ;
702: ; Eingabe: AX=X-Koordinate (0-639)
703: ; der oberen
704: ; linken Ecke
705: ; BX=Y-Koordinate (0-479) der ob
706: ; oberen linken Ecke
707: ;
708: ; CX=X-Koordinate (0-639) der
709: ; unteren rechten Ecke
710: ; DX=Y-Koordinate (0-479) der
711: ; unteren rechten Ecke
712: ; DI=Farbe Low-Byte
713: ; Ausgabe: keine
714: CODESEG
715: PROC AsmBox
716:      push ax
717:      push bx
718:      push cx
719:      push dx
720:      push di
721:      push dx
722:      mov dx, bx
723:      call AsmLine
724: ; obere Linie zeichnen
725:      pop dx
726:      push ax
727:      mov ax, cx
728:      call AsmLine
729: ; rechte Linie zeichnen
730:      pop ax
731:      push bx
732:      mov bx, dx
733:      call AsmLine
734: ; untere Linie zeichnen
735:      pop bx
736:      mov cx, ax
737:      call AsmLine
738: ; linke Linie zeichnen
739: BoxExit:
740:      pop di
741:      pop dx
742:      pop cx
743:      pop bx
744:      pop ax
745:      ret
746: ENDP
747: ;
748: ; BAR.ASM:
749: ; Zeichnet ein gefülltes Rechteck
750: ; im Grafik-Bildschirm
751: ;
752: ; Eingabe: AX=X-Koordinate (0-639)
753: ; der oberen
754: ; linken Ecke
755: ; BX=Y-Koordinate (0-479) der
756: ; oberen
757: ; linken Ecke
758: ; CX=X-Koordinate (0-639) der
759: ; unteren rechten Ecke
760: ; DX=Y-Koordinate (0-479) der
761: ; unteren rechten Ecke
762: ; DI=Farbe Low-Byte
763: ; Ausgabe: keine
764: CODESEG
765: PROC AsmBar
766:      push ax
767: ; Register retten, die nicht schnell
768: ; genug in den nächsten Stack kommen
769:      push bx
770:      push cx
771:      push dx
772:      push di
773:      mov bp, dx
774:      mov dx, bx
775: BarLoop:
776:      call AsmLine
777: ; oberste Linie zeichnen
778:      inc bx
779: ; Start- und Endkoordinaten der Linie
780: ; erhöhen
781:      inc dx
782:      cmp bx, bp
783: ; Ende erreicht?
784:      jbe BarLoop
785: ; Nächste Linie zeichnen
786: BarExit:
787:      pop di
788:      pop dx
789:      pop cx
790:      pop bx
791:      pop ax
792:      ret
793: ENDP
794: ;
795: ; ----- P R O G R A M M S T A R T -----
796: GfxStart:
797:      mov ax, @data
798:      mov ds, ax
799:      call AsmInitGraphic

```

```

800: ; Grafikmodus einschalten
801: ;
802: ; "Herkömmliche" waagerechte Linie
803: n
804: ; mit Bresenham-Algorithmus zeichnen
805:      mov si, 1
806: ; Zähler setzen
807: _GfxLoop4:
808:      mov di, si
809: ; Anfangsfarbe setzen
810:      xor ax, ax
811: ; In oberer, linker Ecke beginnen
812:      xor bx, bx
813:      mov cx, MaxX
814:      mov dx, 1
815: ; YEnde > YAnfang !!!
816: GfxLoop4:
817:      call AsmLine
818: ; Linie zeichnen
819:      inc bx
820:      inc dx
821:      cmp dx, MaxY
822: ; Y > MaxY?
823:      jbe GfxLoop4
824:      inc si
825:      cmp si, 16
826: ; 10 Durchläufe?
827:      jne _GfxLoop4
828:      xor ah, ah
829: ; Tastatur abfragen
830:      int 16h
831:      call AsmInitGraphic
832: ; Bildschirm löschen
833: ;
834: ; ----- Schnelle waagerechte
835: ; Linie zeichnen -----
836:      mov si, 1
837: ; Zähler setzen
838: _GfxLoop3:
839:      mov di, si
840: ; Anfangsfarbe setzen
841:      xor ax, ax
842: ; In oberer, linker Ecke beginnen
843:      xor bx, bx
844:      mov cx, MaxX
845:      xor dx, dx
846: ; YEnde == YAnfang !!!
847: GfxLoop3:
848:      call AsmLine
849: ; Linie zeichnen
850:      inc bx
851:      inc dx
852:      cmp dx, MaxY
853: ; Y > MaxY?
854:      jbe GfxLoop3
855:      inc si
856:      cmp si, 16
857: ; 15 Durchläufe?
858:      jne _GfxLoop3
859:      xor ah, ah
860:      int 16h
861:      call AsmInitGraphic
862: ; Bildschirm löschen
863: ;
864: ; "Herkömmliche" senkrechte Linien
865: mit
866: ; Bresenham-Algorithmus zeichnen
867:      mov si, 1
868: ; Zähler setzen
869: _GfxLoop2:
870:      mov di, si
871: ; Anfangsfarbe setzen
872:      xor ax, ax
873: ; In oberer, linker Ecke beginnen
874:      xor bx, bx
875:      mov cx, 1
876: ; XEnde > XAnfang !!!
877:      mov dx, MaxY
878: GfxLoop2:
879:      call AsmLine
880: ; Linie zeichnen
881:      inc ax
882:      inc cx
883:      cmp cx, MaxX
884: ; X > MaxX?
885:      jbe GfxLoop2
886:      inc si
887:      cmp si, 16
888: ; 10 Durchläufe?
889:      jne _GfxLoop2
890:      xor ah, ah
891: ; Tastatur abfragen
892:      int 16h
893:      call AsmInitGraphic
894: ; Bildschirm löschen
895: ;
896: ; ----- Schnelle senkrechte
897: ; Linie zeichnen -----
898:      mov si, 1
899: ; Zähler setzen
900: _GfxLoop1:
901:      mov di, si
902: ; Anfangsfarbe setzen
903:      xor ax, ax
904: ; In oberer, linker Ecke beginnen
905:      xor bx, bx
906:      xor cx, cx
907: ; XEnde == XAnfang !!!
908:      mov dx, MaxY
909: GfxLoop1:
910:      call AsmLine
911: ; Linie zeichnen
912:      inc ax

```

Demoprogramm für die Grafikbefehle Line, Bar und Box


```

913:      inc cx
914:      cmp cx, MaxX
915: ; X > MaxX ?
916:      jbe GfxLoop1
917:      inc si
918:      cmp si, 16
919: ; 15 Durchläufe ?
920:      jne _GfxLoop1
921:      xor ah, ah
922:      int 16h
923:      call AsmInitGraphic
924: ; Bildschirm löschen
925: ;
926: ; Schräge Linien zeichnen -
927:      mov ax, [xx]
928: ; X- und Y-Koordinaten laden
929:      mov bx, [yy]
930: _GfxLoop:
931:      cmp ax, MaxX
932: ; X > MaxX ?
933:      jle DemoLab1
934:      mov ax, MaxX
935:      mov [xInc], -xStep
936:      mov cl, [cInc]
937:      add [Color], cl
938: ; Farbe=Farbe + Farbkrement
939: DemoLab1:
940:      cmp ax, MinX
941: ; X < MinX ?
942:      jge DemoLab2
943:      mov ax, MinX
944:      mov [xInc], xStep
945:      mov cl, [cInc]
946:      add [Color], cl
947: ; Farbe=Farbe + Farbkrement
948: DemoLab2:
949:      cmp bx, MaxY
950: ; Y > MaxY ?
951:      jle DemoLab3
952:      mov bx, MaxY
953:      mov [yInc], -yStep
954:      mov cl, [cInc]
955:      add [Color], cl
956: ; Farbe=Farbe + Farbkrement
957: DemoLab3:
958:      cmp bx, MinY
959: ; Y < MinY ?
960:      jge DemoLab4
961:      mov bx, MinY

```

```

962:      mov [yInc], yStep
963:      mov cl, [cInc]
964:      add [Color], cl
965: ; Farbe=Farbe + Farbkrement
966: DemoLab4:
967:      mov cl, [Color]
968:      cmp cl, MaxColor
969: ; Farbe > MaxColor ?
970:      jl DemoLab5
971:      mov cl, MaxColor
972:      mov [cInc], -cStep
973: DemoLab5:
974:      cmp cl, MinColor
975: ; Farbe < MinColor ?
976:      jg DemoLab6
977:      mov cl, MinColor
978:      mov [cInc], cStep
979: DemoLab6:
980:      mov [Color], cl
981: ; Farbe setzen
982:      mov di, cx
983:      push ax
984:      push bx
985:      call AsmPlot
986:      mov ah, 1
987: ; Tastatur abfragen
988:      int 16h
989:      jnz GfxExit
990: ; Taste gedrückt -> Programm beenden
991:      pop bx
992:      pop ax
993:      add ax, [xInc]
994: ; X- und Y-Koordinate erhöhen
995:      add bx, [yInc]
996:      jmp _GfxLoop
997: GfxExit:
998:      xor ah, ah
999: ; Tastatur abfragen
1000:      int 16h
1001:      call AsmInitGraphic
1002: ; Bildschirm löschen
1003: ;
1004: ; ----- Ausgefüllte Rechtecke
1005: ; zeichnen -----
1006:      mov ax, 160
1007:      mov bx, 80
1008:      mov cx, 480
1009:      mov dx, 400
1010:      mov di, 1

```

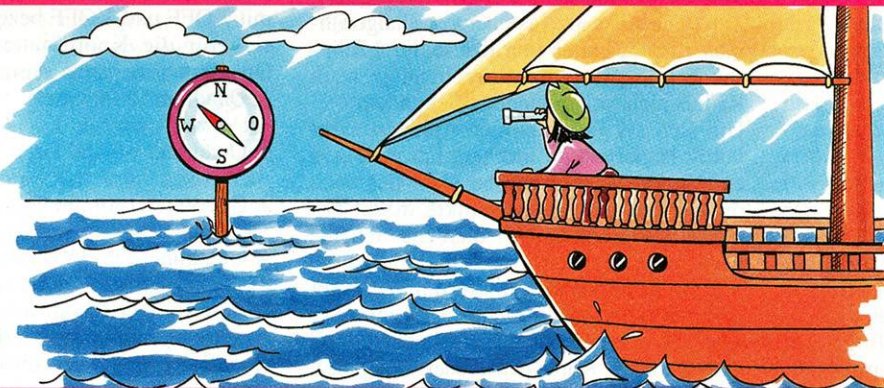
```

1011: _GfxLoop:
1012:      call AsmBar
1013: ; Ausgefülltes Rechteck zeichnen
1014:      add ax, 10
1015:      add bx, 10
1016:      sub cx, 10
1017:      sub dx, 10
1018:      inc di
1019:      cmp di, 15
1020:      jne _GfxLoop
1021:      xor ah, ah
1022: ; Tastatur abfragen
1023:      int 16h
1024:      call AsmInitGraphic
1025: ; Bildschirm löschen
1026: ;
1027: ; ----- Rechtecke zeichnen -----
1028:      mov ax, 310
1029:      mov bx, 230
1030:      mov cx, 330
1031:      mov dx, 250
1032:      mov di, 1
1033: GfxLoop:
1034:      call AsmBox
1035: ; Rechteck zeichnen
1036:      sub ax, 10
1037:      sub bx, 10
1038:      add cx, 10
1039:      add dx, 10
1040:      inc di
1041:      cmp di, 15
1042:      jne GfxLoop
1043:      xor ah, ah
1044:      int 16h
1045:      call AsmInitText
1046: ; Textmodus wieder einschalten
1047:      mov ax, 4C00h
1048: ; Zurück zu DOS
1049:      int 21h
1050: END GfxStart

```

Demoprogramm für die Grafikbefehle Line, Bar und Box

WENN IM DATEN-MEER KEIN LAND IN SICHT IST...



Die Bestell-Hotline:
08121/769-102
oder fix faxen:
08121/769-103

... WEIST IHNEN COLUMBUS 2.5 DEN RICHTIGEN KURS.

Überlassen Sie die Navigation auf Ihrem PC nicht dem Zufall, sondern Columbus 2.5 für Windows. Damit haben Sie Ihre Daten jederzeit im Griff.

Columbus 2.5 für Windows spricht sogar mit Ihnen – und das natürlich in Deutsch! Er sagt Ihnen alles über Uhrzeit, Systemressourcen und ohne daß Sie dafür Ihre Arbeit unterbrechen müssen.

► **Minimieren Sie Ihren Festplattenbedarf:** Statt selten benötigte Programme von Ihrer Festplatte zu entfernen, weil das neue Programm mal wieder so viel Platz braucht, lassen Sie es von Columbus einfach komprimieren.

► **Schluß mit der zeitraubenden Daten-suche:** Columbus findet Ihre Daten blitzschnell und ohne schwierige Parameter-Definition, egal ob Sie nach Dateinamen, -inhalten oder virtuellen Gruppen suchen.

► **Überlassen Sie Columbus 2.5 Ihre Datenorganisation:** Ergänzen Sie die nichtssagenden 8-Buchstaben Datei-Namen durch aussagekräftige Kommentare, organisieren Sie sich Ihre Dateien aus verschiedensten Anwendungen nach Projekten.

► **Schützen Sie Ihre Daten vor unerlaubtem Zugriff:**

- Zugriffsschutz durch Passwort
- Reißwolf zur absolut sicheren Vernichtung

Columbus 2.5 für Windows DM 79,-

Euromaster-Superpaket
Engl./Dt. und Frz./Dt.

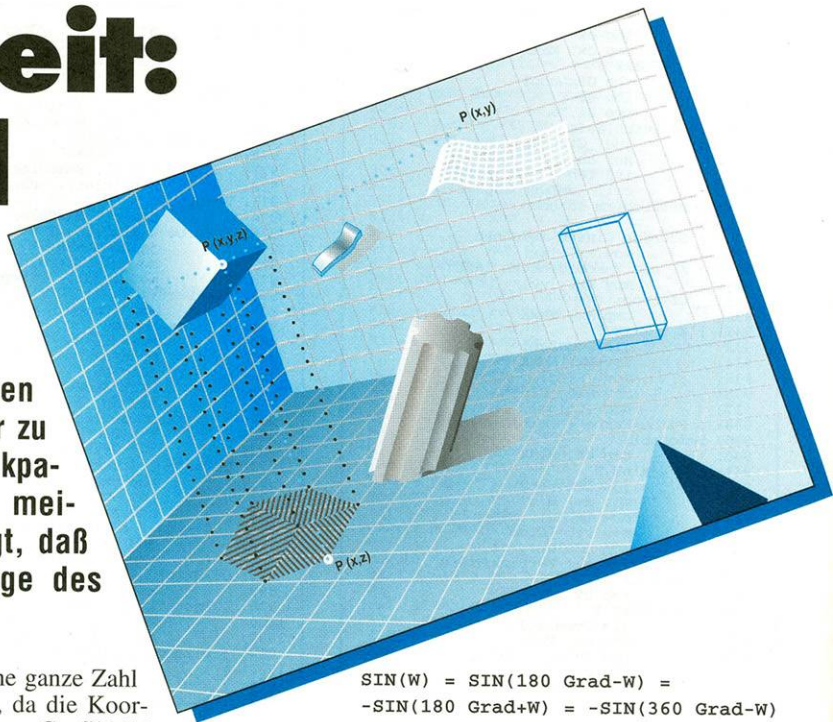
DM 89,-



IMMER VOLLES PROGRAMM!

Schwerarbeit: Kreise und Ellipsen

Die algorithmische Umsetzung von Kreisen und Ellipsen gehört für den Programmierer zu den unangenehmsten Aufgaben jedes Grafikpaketes. Entsprechend langsam arbeiten die meisten Unterprogramme. Dieser Beitrag zeigt, daß Geschwindigkeit auch hier nur eine Frage des Know-how ist.



Der grundsätzliche zeitliche Engpaß bei der Zeichnung von Ellipsen und Kreisen wird durch deren mathematische Beschreibung deutlich. Wenn wir das Zylinderkoordinatensystem heranziehen, den Mittelpunkt durch die Koordinaten XM und YM, den Radius in x-Richtung durch XR und den Radius in y-Richtung durch YR beschreiben sowie schließlich noch einen Winkel W einführen, kann man eine Ellipse als die Punktmenge definieren, welche die folgenden Gleichungen erfüllt (Bild 1):

$X = XM + XR * \cos(W)$
für den Wertebereich
 $0 \text{ Grad} \leq W < 360 \text{ Grad}$
und

$Y = YM + YR * \sin(W)$
bei gleichem Wertebereich.

Falls XR und YR identisch sind, liegt der Sonderfall eines Kreises vor, so daß sich mit dem Programmmodul sowohl Ellipsen als auch Kreise zeichnen lassen.

Dies ist allerdings schon die einzig erfreuliche Tatsache. Geringere Begeisterung bereiten in erster Linie die transzendenten Funktionen Sinus und Cosinus, da sie bis auf fünf Ausnahmen (bei 0, 90, 180, 270 und 360 Grad) keine ganzen Zahlen als Ergebnis liefern, mit denen der Prozessor weiterrechnen könnte. Man ist daher gezwungen, eine Fließkommabibliothek zu programmieren und sich Sinus- wie auch Cosinus-Funktion durch Potenzreihenentwicklungen anzunähern. Die Berechnung eines Wertes kann dabei durchaus mehrere Millisekunden in Anspruch nehmen, eine für Computerverhältnisse indiskutabel große Zeitspanne. Die Krönung besteht jedoch darin, daß der Algorithmus das Ergebnis der Berechnung

$YR * \sin(W)$
beziehungsweise
 $XR * \cos(W)$

wieder in eine ganze Zahl runden muß, da die Koordinaten des Grafikbildschirms nur aus Integerwerten bestehen. Man geht also von einer ganzen Zahl (dem Winkel W) aus, wandelt diese ins Fließkommaformat, um den Sinus beziehungsweise Cosinus berechnen zu können, multipliziert das Ergebnis mit dem Radius (der zuvor natürlich auch ins Fließkommaformat umzuwandeln ist) und rundet schließlich das Ergebnis wieder in eine ganze Zahl. Die Ausgaben solcher Grafikalgorithmen zum Zeichnen von Kreisen und Ellipsen benötigen dementsprechend je nach Radius nicht selten mehrere Sekunden.

Um sich als Programmierer jedoch nicht völlig lächerlich zu machen, hat man sich ein Verfahren überlegt, bei dem nur noch relativ wenige Ellipsenpunkte direkt berechnet werden, indem man die Schrittweite des Winkels W vergrößert und die einzeln errechneten Punkte durch Linien verbindet. Da das Ziehen einer Linie wesentlich schneller vonstatten geht (siehe Teil 2 dieses Kurses in dieser Ausgabe), spart man tatsächlich enorm Zeit ein. Der Nachteil dieser Technik liegt darin, daß Kreise je nach Schrittweite nicht mehr wie Kreise, sondern wie Bienenwaben aussehen (Bild 2). Eine andere Alternative, Zeit zu sparen, besteht in der Nutzung bestimmter mathematischer Gesetzmäßigkeiten der Sinus- und Cosinus-Funktionen. Es gilt nämlich:

$$\begin{aligned} \sin(W) &= \sin(180 \text{ Grad} - W) = \\ -\sin(180 \text{ Grad} + W) &= -\sin(360 \text{ Grad} - W) \\ \cos(W) &= \cos(360 \text{ Grad} - W) = \\ -\cos(180 \text{ Grad} - W) &= -\cos(180 \text{ Grad} + W) \end{aligned}$$

Durch diese Tatsache läßt sich mit einem berechneten Wertepaar jeweils in jedem Quadrat ein Punkt setzen, da sie sich nur im Vorzeichen unterscheiden. Wenn wir die Terme

$XR * \cos(W)$
und

$YR * \sin(W)$
mit XOFF und YOFF bezeichnen, ergeben sich für die Koordinaten der einzelnen Quadranten folgende Zusammenhänge:

1. Quadrant (0-90 Grad):

$$X = XM + XOFF : Y = YM - YOFF$$

2. Quadrant (90-180 Grad):

$$X = XM + XOFF : Y = YM + YOFF$$

3. Quadrant (180-270 Grad):

$$X = XM - XOFF : Y = YM + YOFF$$

4. Quadrant (270-360 Grad):

$$X = XM - XOFF : Y = YM - YOFF$$

Durch diese Methode ist die Rechenzeit um zirka den Faktor 4 verringierbar. Allerdings kann man nach diesem Schema nur

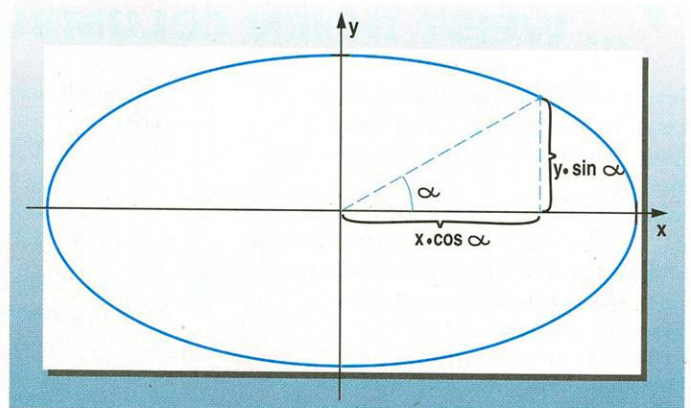


Bild 1. Zusammenhang zwischen Winkel, Sinus- und Cosinus-Funktion bei Ellipsen. Sind XR und YR identisch, liegt der Sonderfall »Kreis« vor.

noch ganze Kreise oder Ellipsen zeichnen, Halb- oder Viertelkreisen ist mit einer Prozedur, die nach diesem Verfahren arbeitet, der Weg versperrt. Außerdem reicht uns die Geschwindigkeitserhöhung gegenüber dem Standardverfahren noch bei weitem nicht aus.

Daß es aber auch genau und schnell geht, zeigt die Ellipsen-Routine aus dem Turbo-Pascal-Grafikpaket, die in Assembler programmiert ist und die schnellsten uns bekannten Ellipsen und Kreise zeichnet. Sie verwendet grundsätzlich offenbar das gleiche Verfahren wie dasjenige, das wir Ihnen an dieser Stelle vorstellen möchten. Wahrscheinlich haben die Turbo-Pascal-Programmierer das entsprechende Modul allerdings noch nicht maximal optimiert, da unsere Routine immerhin weitere rund 20 Prozent schneller arbeitet.

Die Idee unserer Verfahrensweise besteht darin, die Sinus- und Cosinus-Werte in einer Integertabelle abzulegen. Dies ist in dieser Form unmittelbar nicht realisierbar, da die Ergebnisse nur im Bereich zwischen -1 und +1 liegen. Wenn man sie jedoch mit einer hinreichend großen Zahl multipliziert, steht der Umwandlung ins Integerformat nichts mehr im Weg. Sofern ein maximaler Radius von 255 Pixeln ausreicht (in y-Richtung kommt in der Praxis sogar nur ein maximaler Radius von 240 Pixeln vor, da der Bildschirm 480 Pixel aufweist), kann man statt mit dem Datentyp Integer mit dem Datentyp Byte rechnen, was die Geschwindigkeit weiter erhöht, da die entsprechenden Prozessorbefehle weniger Taktzyklen zur Ausführung benötigen. Für die Ermittlung eines Tabellenwertes eines Winkels W läßt sich daher folgende Rechnung benutzen:

```
Wert = INT(COS(W)*255)
```

Damit ist der Offset XOFF mit der Formel

```
XOFF = XM + XR*Wert/255
```

berechenbar. Der Trick besteht nun darin, daß der Ausdruck

```
XR*Wert
```

so groß wird, daß die Division durch 255 mit Schiebeoperationen des Prozessors durchgeführt werden kann. Fatal wäre hingegen zuerst die Rechnung

```
Wert/255
```

da man hiermit nur Werte zwischen 0 und 1 produzieren würde. Betrachten Sie folgendes Beispiel, bei dem der Offset für den Winkel W 30 Grad und der Radius XR

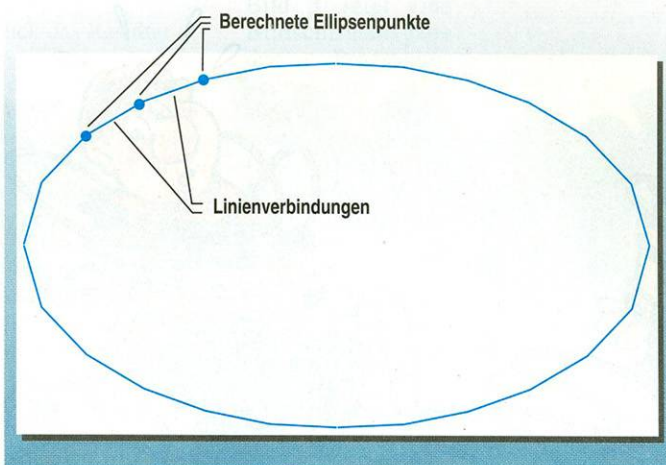


Bild 2. Verbindet man ermittelte Punkte durch Linien miteinander, um Rechenzeit einzusparen, erscheint die Ellipse mit zunehmender Schrittweite wie eine Bienenwabe.

100 Einheiten betragen soll. Der zugehörige Tabellenwert ergibt sich wie folgt:

```
Wert = INT(COS(30 Grad)*255)
      = INT(0.866025403*255)
      = INT(220.8364)
      = 220
```

Aus dem Produkt

```
XR*Wert
resultiert
100*220=22000
```

und die Ganzzahldivision 22000/255 führt zu 86, was dem Offset in x-Richtung entspricht. Hätte man zuerst die Ganzzahldivision

```
Wert/255=220/255=0
```

durchgeführt, wäre als Offset der falsche Wert

```
100*0=0
```

berechnet worden.

Die Frage, die an dieser Stelle fast zwangsläufig auftaucht, beschäftigt sich mit der Rechengenauigkeit. Theoretisch müßte die Tabelle unendlich viele Werte enthalten, da ja auch unbegrenzt viele Winkel W mit entsprechend vielen Cosinus- und Sinus-Werten existieren. In der Praxis benötigt man jedoch nur eine Genauigkeit von jeweils einem Bildpunkt in x- und y-Richtung, das heißt, die Forderung an die Tabellenwerte muß lauten, daß zwei benachbarte Tabellenwerte nicht mehr als um den Wert 1 differieren dürfen. Hiermit ist auch bei maximalem Radius 255 gewährleistet, daß sich zwei aufeinanderfolgende Offsets

```
XR*Wert/255
```

nicht um mehr als eine Zeile beziehungsweise Spalte unterscheiden. Dieses Faktum ist bei der Grafikprogrammierung sehr wichtig, da der Kreis und die Ellipse durch eine geschlossene Linie umgeben sein müssen. Wäre dies nicht der Fall, könnte man sie zum Beispiel nicht mit Farbe füllen, da die Füll-Routine keine durchgehende Begrenzung finden könnte. Bei Radien kleiner als 255 Einheiten wird das Problem immer irrelevant, da der Faktor

XR/255

immer kleiner als 1 ist.

Durch Ausprobieren findet man heraus, daß die maximale Schrittweite für den Winkel W, die unsere Forderung gerade noch erfüllt,

```
90/395=0.227848
```

beträgt. Sofern der Programmierer also die Tabelle mit dem Winkel 0 Grad beginnend anlegt und für jeden Eintrag einen Winkelzuwachs von 90/395 Grad berücksichtigt, erhält er im Bereich zwischen 0 und 90 Grad genau 395 Ergebnisse. Werte für weitere Winkel sind nicht erforderlich, da sie sich ja wie zuvor gesehen für alle übrigen drei Quadranten durch Vertauschen der Vorzeichen ergeben.

Bislang war immer nur von Offsets in x-Richtung die Rede. Für die Berechnung der Versatzwerte in y-Richtung läßt sich die gleiche Tabelle benutzen, wenn man die Werte von hinten ausliest, da gilt:

```
SIN(W) = COS(90 Grad-W)
```

Die Tabellenwerte sind zum Beispiel auf einem programmierbaren Taschenrechner wie folgt ermittelbar:

```
FOR I = 0 TO 395
  WERT(I) = 255*INT(COS(90/395*I))
NEXT I
```

Wer nun die Bildschirmkoordinaten für einen bestimmten Winkel W berechnen möchte, muß diesen zunächst in den Tabellenindex umsetzen. Falls W kleiner oder gleich 90 Grad ist, reicht hierzu die einfache Formel

```
Index I = (W/90)*395
```

aus. Bei größeren Winkeln wird zusätzlich der Quadrant nach dem Schema

```
Quadrant = INT(W/90) + 1
```

und anschließend der Index nach der Formel

```
Index I = 395*(W-Quadrant*90)/90
```

ermittelt, wobei sich W im Bereich von 0 bis 359 Grad bewegen darf. Der Quadrant spielt nur insofern eine Rolle, daß er festlegt, ob die Offsets zu den Mittelpunktkoordinaten zu addieren oder zu subtrahieren sind. Hat man den Index I berechnet, erhält man die Koordinaten des Bildpunktes:

```
X = XM + (-) XR*Wert(I)/255
```

```
Y = YM + (-) YR*Wert(395-I)/255
```

Beim Ansehen dieser Formeln könnte der Eindruck entstehen, daß der Algorithmus selbst Radien größer als 255 Einheiten gestattet. Sicherlich läßt sich der Ausdruck

```
XR*Wert(I)/255
```

auch mit XR=300 oder XR=1000 berechnen. Da der Faktor XR/255 damit aber größer als 1 ausfällt, liegen zwei benachbarte Punkte der Ellipse um mehr als einen Bildpunkt auseinander, wenn die Tabelleneinträge um genau 1 differieren. Betrachten wir dazu ein kleines Beispiel. Nehmen Sie an, zwei Tabellenwerte für die Indizes 3 und 4 lauten

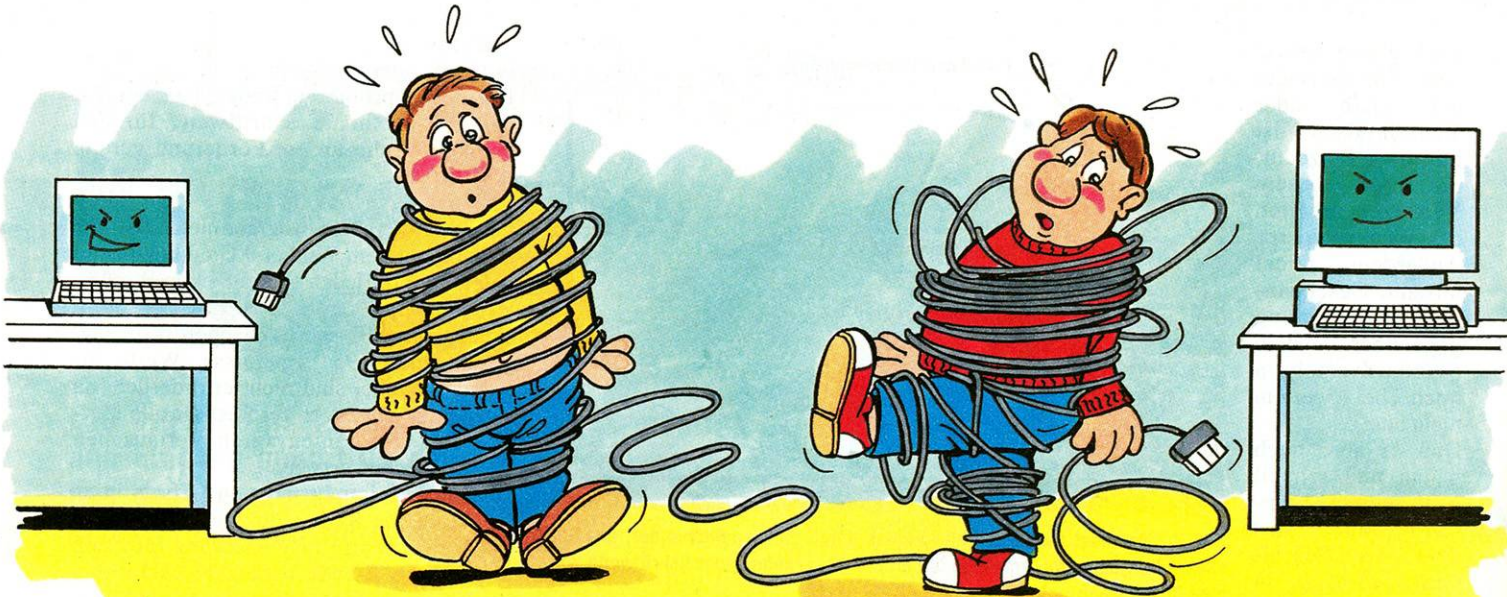
```
Wert(3)=100
```

```
und
```

```
Wert(4)=101
```

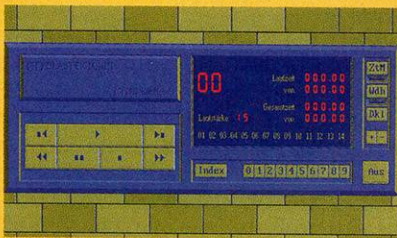
Bei einem Radius XR=400 würde damit der erste Ausdruck das Ergebnis

LASSEN SIE SICH NICHT VON IHREM RECHNER EINWICKELN!



29,-

CD BLASTER MK-III

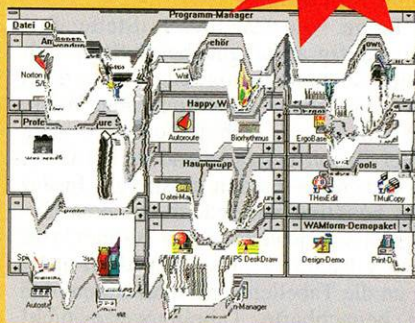


Bringen Sie Ihr CD-ROM-Laufwerk auf Trab! Mit dem Audio-CD-Abspielprogramm für Ihren Multimedia-PC. Die realistische 3D-Oberfläche besitzt alle Grundfunktionen eines echten CD-Players!

DARK FADER

Bei jeder PC-Pause leidet die Bildröhre. Der Bildschirm-Schoner "Dark Fader" macht Schluß damit. Und läßt mit 29 lustigen Film-Ideen keine Langeweile aufkommen. Sie selbst führen Regie und können einzelne Sequenzen zu ganzen Serien zusammenschalten. (für Windows)

49,-



TEXTCOMPILER

Verwandeln Sie Ihre Texte in ausführbare Programme mit eleganter Textpräsentation.
DM 29,-

TERMINPLANER

Gut geplant ist halb gewonnen! Managen Sie Ihre Zeit mit dem Terminplaner, und Sie werden keine Termine mehr platzen lassen.
DM 29,-

TIPPSY

Schneller an Computer oder Schreibmaschine! Tippen im Zehnfiingersystem spielend gelernt.
DM 39,-

Die Bestell-Hotline:
08121/769-102
oder fix faxen:
08121/769-103

ICON-DESIGNER

Verschönern Sie Ihre Windows-Oberfläche. Wie? Mit der Maus fangen Sie ein Motiv von der Oberfläche ein und bearbeiten es im Icon Designer nach. Ebenso simpel wie originell! (für Windows)
DM 49,-

MULTI-COPY

Das Duplizieren Ihrer Disketten macht Multi-Copy kinderleicht. Einfach Original-Disketten in den Speicher lesen und kopieren – so oft Sie wollen.
DM 29,-

MENUE

Sie haben den lästigen und zeitraubenden Programm-Start satt? Dann gestalten Sie Ihr eigenes Menuesystem mit bequem zu startenden Programmen.
DM 29,-



IMMER VOLLES PROGRAMM!

Bitte gewünschtes Programm ankreuzen.

DMV Software, Postfach 1146, 85580 Poing

Meine Adresse:

105041

Datum

Unterschrift (Bei Minderjährigen der gesetzliche Vertreter.)

400*100/255=156

und der zweite Ausdruck das Resultat

400*101/255=158

liefern, so daß sich die x-Koordinaten zweier unmittelbar benachbarter Ellipsenpunkte um zwei Bildpunkte unterscheiden. Der Versuch, diese Ellipse mit Farbe zu füllen, scheitert, da die Füll-Routine durch die Lücke zwischen 156 und 158 durchlaufen und den gesamten Bildschirm einfärben würde.

Lassen Sie uns nochmals auf kleinere Radien als 255 Einheiten zu sprechen kommen. Hier wird der Faktor $XR/255$ beständig geringer (und immer kleiner als 1), je kleiner XR ausfällt. Wenn sich zwei benachbarte Tabellenwerte daher um maximal eine Einheit unterscheiden (die Cosinus- und Sinus-Funktion verlaufen nicht linear, so daß an den Stellen mit geringer Steigung zwei benachbarte Tabellenwerte identisch sind), kann es vorkommen, daß für zwei aufeinanderfolgende Indizes I die gleichen Offsets berechnet werden. Nehmen wir wieder das vorherige Beispiel, diesmal jedoch mit dem Radius 50. Damit würde der erste Ausdruck das Ergebnis

50*100/255=19

und der zweite Ausdruck ebenfalls das Resultat

50*101/255=19

erzielen. Es wäre also reine Zeitverschwendung, Bildpunkte doppelt oder noch öfter zu setzen, so daß sich unsere Routine jeweils nach dem Zeichnen eines Bildpunktes dessen Koordinaten merkt und nach Berechnung der Offsets des nächsten Tabellenwertes diese damit vergleicht. Stimmen sie überein, wird das Plotmodul übersprungen. Die Programmsequenz arbeitet somit um so schneller, je kleiner die Radien sind, da die Anzahl der tatsächlich gesetzten Bildpunkte immer stärker abnimmt.

Bild 3 zeigt eine Bildschirmausgabe des Programms »curve.asm« (Listing), das selbst auf einer langsamen Lowcost-VGA-Grafikkarte mit 46 Ellipsen insgesamt mehr als 52 600 Bildpunkte in 0,56 Sekunden zeichnet.

Pro Sekunde erlaubt das Programm damit das Setzen von fast 100 000 Bildpunkten. 99 Prozent aller Grafikroutinen müssen bereits an diesem Punkt passen. Die Verbleibenden kapitulieren spätestens dann, wenn es um die Zeichnung von beliebigen Ellipsenausschnitten geht. Fast alle uns bekannten Routinen (mit Ausnahme von Borlands C und Pascal) arbeiten mit flexiblen Ellipsen- und Kreisausschnitten erheblich langsamer, da das Verfahren mit den vier gleichzeitig berechneten Punkten (für jeden Quadranten einen) nicht mehr funktioniert. Probieren Sie als Extrembeispiel eine Ellipse von 0 bis 359 Grad mit Standardroutinen aus, die fast genauso viele Punkte wie eine Vollellipse aufweist, jedoch nicht mehr nach deren optimiertem Algorithmus berechnet werden kann. Wenn Sie »curve.asm« abändern, indem Sie die Befehle

```
mov ax, 360
```

gegen

```
mov ax, 359
```

austauschen, dürften Sie überrascht feststellen, daß das Programm die Grafiken fast genauso schnell aufbaut wie zuvor.

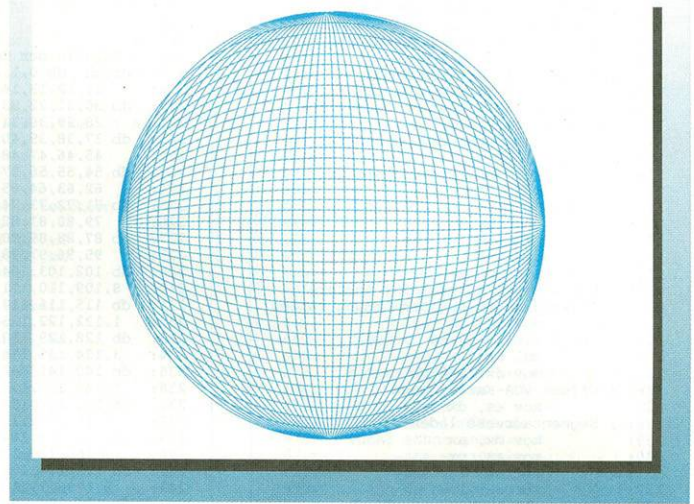


Bild 3. Bildschirmausgabe des Programms »curve.asm«

Für diesen Fall liegen nämlich zwei Offset-Tabellen im Speicher vor (für XOFF und YOFF), die anschließend von den einzelnen Routinen zum Zeichnen eines beliebigen Quadrantenausschnitts nur noch wiederholt auszulesen sind. Die meisten Grafikfunktionen dieser Art berechnen dagegen zum Beispiel bei zweieinhalb zu zeichnenden Quadranten die Koordinaten der Punkte zwei- beziehungsweise dreimal, was einen erheblichen Zeitaufwand mit sich bringt.

Um das relativ lange Listing zu kürzen, wurden für verschiedene, häufig benutzte Befehlssequenzen Makros definiert. Der Quelltext wird mit

TASM DEMO.ASM

assembliert und mit

TLINK DEMO.OBJ

in eine Exe-Datei überführt.

(Frank Riemenschneider/bj)

Listing: curve.asm

```
1: ;Programm: curve.asm
2: ;Funktion: Demo-Programm für den
3: ; Grafikbefehl
4: ; CURVE (Kreise+Ellipsen)
5: ;Sprache : Assembler
6: ;Autor : Frank Riemenschneider
7: ;(c) 1993 DMV Daten- und Medienverlag
8: ;
9: IDEAL
10: ; IDEAL-Modus einschalten
11: MODEL SMALL
12: ; Speichermodell SMALL wählen
13: STACK
14: db 512 dup (?)
15: CODESEG
16: ;
17: ; ASMINITGRAPHIC:
18: ; Schaltet die VGA-Karte in den
19: ; Grafikmodus und löscht Bildschirm
20: ;
21: ; Eingabe: keine
22: ; Ausgabe: keine
23: PROC AsmInitGraphic
24: push ax
25: ; Für alle Fälle alle Register retten
26: push bx
27: push cx
28: push dx
29: push ds
30: push di
31: xor ah, ah
32: ; Funktionsnummer laden
33: mov al, 12h
34: ; Grafikmodus 640 x 480 Punkte in 16
```

```
35: int 10h
36: ; Farben setzen
37: mov dx, 3CEh
38: ; Port-Adresse des VGA-Grafik
39: ; controllers laden
40: mov ax, 0205h
41: ; Read-Modus 0 und Write-Modus 2
42: out dx, ax
43: ; im Modusregister setzen
44: mov ax, 0003h
45: ; Verknüpfungsmodus = Ersetzen im
46: ; Function-Select-Register setzen
47: out dx, ax
48: pop di
49: ; Alle Registerinhalte wieder
50: ; herstellen
51: pop ds
52: pop dx
53: pop cx
54: pop bx
55: pop ax
56: ret
57: ; Back home
58: ENDP
59: ;
60: ; ASMINITTEXT:
61: ; Schaltet VGA-Karte i. d. Textmodus
62: ;
63: ; Eingabe: keine
64: ; Ausgabe: keine
65: PROC AsmInitText
66: push ax
67: ; Für alle Fälle alle Register retten
68: push bx
69: push cx
70: push dx
```

```
71: push ds
72: push di
73: mov dx, 3CEh
74: ; Port-Adresse des VGA-Grafik-
75: ; Controllers laden
76: mov ax, 0005h
77: ; Mode- und Function-Select-Register
78: ; mit ihren
79: ; Standardwerten laden
80: out dx, ax
81: mov ax, 0003h
82: out dx, ax
83: xor ah, ah
84: ; Funktionsnummer laden
85: mov al, 3
86: ; Textmodus 80 x 25 Zeichen in 16
87: ; Farben setzen
88: int 10h
89: pop di
90: ; Alle Registerinhalte wieder
91: ; herstellen
92: pop ds
93: pop dx
94: pop cx
95: pop bx
96: pop ax
97: ret
98: ; Back home
99: ENDP
100: ;
101: ; CALCXY:
```

Demoprogramm für den Grafikbefehl Curve
(Kreise und Ellipsen)

GRAFIK-PROGRAMMIERUNG

Kreise und Ellipsen zeichnen

```

102: ; Berechnet Segment und Offset des
103: ; Byte im Video-RAM, sowie die
104: ; Bit-Maske für das passende Bit bei
105: ; gegebener X- und Y-Koordinate
106: ;
107: ; Eingabe: AX = X-Koordinate (0-639)
108: ;          BX = Y-Koordinate (0-479)
109: ; Ausgabe: ES = Segmentadresse des
110: ;           Bytes im Video-RAM
111: ;           DI = Offsetadresse des Bytes im
112: ;           Video-RAM
113: ;           DL = Bit-Maske des Bytes im
114: ;           Video-RAM
115: PROC CalcXY
116:     push bx
117:     push cx
118:     push si
119:     mov cx, ax
120: ; Koordinaten nochmal für spätere
121: ; Verwendung in der Berechnung retten
122:     mov ax, 80
123: ; Pro Zeile 80 Bytes
124:     mul bx
125:     mov dx, 0A000h
126: ; Offset VGA-Karte laden
127:     mov es, dx
128: ; Segmentadresse laden
129:     mov dx, ax
130:     mov ax, cx
131:     shr ax, 1
132:     shr ax, 1
133:     shr ax, 1
134: ; x/8
135:     add dx, ax
136: ; Spaltenoffset addieren
137:     mov di, dx
138: ; DI mit Offset laden
139: ;
140: ; Berechnung der Bit-Maske
141:     and cl, 7
142: ; x MOD 8
143:     mov dl, 80h
144: ; Maske für Bit 0
145:     shr dl, cl
146: ; Maske für gew. Bit durch Rotation
147: ; der ursprünglichen Maske erzeugen
148:     pop si
149:     pop cx
150:     pop bx
151:     ret
152: ; Back home
153: ENDP CalcXY
154: ;
155: ; VGAPLOT:
156: ; Setzt einen Bildpunkt im Video-RAM
157: ; der VGA-Karte
158: ;
159: ; Eingabe: AX=X-Koordinate des Pixels
160: ;          BX=Y-Koordinate des Pixels
161: ;          DI=Farbe Low-Byte
162: ; Ausgabe: keine
163: PROC AsmPlot
164:     push ax
165:     push bx
166:     push cx
167:     push dx
168:     push si
169:     push di
170:     mov cx, di
171: ; Farbe retten
172:     call CalcXY
173: ; Offset- und Segmentadresse, sowie
174: ; die Bit-Maske aus den X- und
175: ; Y-Koordinaten berechnen
176:     mov ah, dl
177: ; Bit-Maske
178:     mov dx, 3CEh
179: ; Port-Adresse des VGA-Grafik-
180: ; Controllers laden
181:     mov al, 08h
182: ; Bit-Maske im Mask-Register setzen
183:     out dx, ax
184:     mov ch, [es:di]
185: ; Latch-Register laden
186:     mov [es:di], cl
187: ; Farbe setzen
188:     pop di
189:     pop si
190:     pop dx
191:     pop cx
192:     pop bx
193:     pop ax
194:     ret
195: ; Back home
196: ENDP
197: ;
198: ; ASMCURVE:
199: ; Zeichnet einen Ellipsenbogen
200: ;
201: ; Eingabe: AX=X-Koordinate Ellipsen-
202: ;          Mittelpunkt (0-639)
203: ;          BX=Y-Koordinate Ellipsen-
204: ;          Mittelpunkt (0-479)
205: ;          CX=Radius in X-Richtung (0-255)
206: ;          DX=Radius in Y-Richtung (0-255)
207: ;          ES=Anfangswinkel in Grad(0-359)
208: ;          SI=Endwinkel in Grad (0-360)
209: ;          DI=Farbe Low-Byte
210: ; Ausgabe: keine
211: DATASEG

```

```

212: ;
213: ;
214: ;
215: ;
216: ; Tabelle der Cosinus-Werte
217: costab db 0,1,2,3,4,5,6,7,8,9,10,
218:         11,12,13,14,15,16,17,18,19
219:         20,21,22,23,24,25,26,27,
220:         28,29,30,31,32,33,34,35,36
221:         37,38,39,40,41,42,43,44,
222:         45,46,47,48,49,50,51,52,53
223:         54,55,56,57,58,59,60,61,
224:         62,63,64,65,66,67,68,69,70
225:         71,72,73,74,75,76,77,78,
226:         79,80,81,82,83,84,85,86,86
227:         87,88,89,90,91,92,93,94,
228:         95,96,97,98,99,100,101,102
229:         102,103,104,105,106,107,10
230:         8,109,110,111,112,113,114,114
231:         115,116,117,118,119,120,12
232:         1,122,122,123,124,125,126,127
233:         128,129,130,130,131,132,13
234:         3,134,135,136,136,137,138,139
235:         140,141,142,142,143,144,14
236:         5,146,147,147,148,149,150,151
237:         152,152,153,154,155,156,15
238:         6,157,158,159,160,160,161,162
239:         163,163,164,165,166,167,16
240:         7,168,169,170,170,171,172,173
241:         173,174,175,176,176,177,17
242:         8,179,179,180,181,181,182,183
243:         184,184,185,186,186,187,18
244:         8,188,189,190,190,191,192,192
245:         193,194,194,195,196,196,19
246:         7,198,198,199,200,200,201,201
247:         202,203,203,204,204,205,20
248:         6,206,207,207,208,209,209,210
249:         210,211,212,212,213,213,21
250:         4,214,215,215,216,216,217,218
251:         218,219,219,220,220,221,22
252:         1,222,222,223,223,224,224,225
253:         225,226,226,227,227,227,22
254:         8,228,229,229,230,230,231,231
255:         231,232,232,233,233,234,23
256:         4,234,235,235,236,236,237
257:         237,237,238,238,238,239,23
258:         9,240,240,240,241,241,241,242
259:         242,242,243,243,243,243,24
260:         4,244,244,245,245,245,245,246
261:         246,246,247,247,247,247,24
262:         8,248,248,248,248,249,249,249
263:         249,250,250,250,250,250,25
264:         1,251,251,251,251,251,252,252
265:         252,252,252,252,253,253,25
266:         3,253,253,253,253,254,254,25
267:         4,254,254,254,255,255,255,255
268:         4,254,254,254,255,255,255,255
269:         5,255,255,255,255,255,255,25
270:         5,255,255,255,255,255,255
271: ArcStartX dw 0
272: ArcStartY dw 0
273: ArcEndX dw 0
274: ArcEndY dw 0
275: x dw ?
276: ; X-Koordinate
277: y dw ?
278: ; Y-Koordinate
279: xr dw ?
280: ; X-Radius
281: yr dw ?
282: ; Y-Radius
283: xoff dw ?
284: ; X-Offset
285: yoff dw ?
286: ; Y-Offset
287: zaehl1 dw ?
288: ; Zähler 1
289: zaehl2 dw ?
290: ; Zähler 2
291: xm dw ?
292: ; X-Koordinate Mittelpunkt
293: anqua dw ?
294: ; Anfangsquadrant
295: endqua dw ?
296: ; Endquadrant
297: anpunkt dw ?
298: ; Anfangspunkt
299: endpunkt dw ?
300: ; Endpunkt
301: offtab1 db 400 dup (?)
302: ; Offset-Tabellen
303: offtab2 db 400 dup (?)
304: CODESEG
305: ;
306: ; Koordinate des Start- und Endpunktes
307: ; des Ellipsenbogens berechnen
308: MACRO CalcPoints StartQuad,EndQuad
309: IFIDNI <StartQuad>,<1>
310:     mov bx,394
311:     sub bx,[anpunkt]
312: ENDIF
313: IFIDNI <StartQuad>,<2>
314:     mov bx,[anpunkt]
315: ENDIF
316: IFIDNI <StartQuad>,<3>
317:     mov bx,394
318:     sub bx,[anpunkt]
319: ENDIF
320: IFIDNI <StartQuad>,<4>
321:     mov bx,[anpunkt]

```

```

322: ENDIF
323:     xor ch,ch
324:     mov dh,ch
325:     mov cl,[offtab1+bx]
326:     mov dl,[offtab2+bx]
327: IFIDNI <StartQuad>,<1>
328:     add cx,[xm]
329:     mov [WORD ArcStartX],cx
330:     mov cx,[y]
331:     sub cx,dx
332:     mov [WORD ArcStartY],cx
333: ENDIF
334: IFIDNI <StartQuad>,<2>
335:     add cx,[xm]
336:     mov [WORD ArcStartX],cx
337:     add dx,[y]
338:     mov [WORD ArcStartY],dx
339: ENDIF
340: IFIDNI <StartQuad>,<3>
341:     mov ax,[xm]
342:     sub ax,cx
343:     mov [WORD ArcStartX],ax
344:     add dx,[y]
345:     mov [WORD ArcStartY],dx
346: ENDIF
347: IFIDNI <StartQuad>,<4>
348:     mov ax,[xm]
349:     sub ax,cx
350:     mov [WORD ArcStartX],ax
351:     mov cx,[y]
352:     sub cx,dx
353:     mov [WORD ArcStartY],cx
354: ENDIF
355: IFIDNI <EndQuad>,<1>
356:     mov bx,394
357:     sub bx,[endpunkt]
358: ENDIF
359: IFIDNI <EndQuad>,<2>
360:     mov bx,[endpunkt]
361: ENDIF
362: IFIDNI <EndQuad>,<3>
363:     mov bx,394
364:     sub bx,[endpunkt]
365: ENDIF
366: IFIDNI <EndQuad>,<4>
367:     mov bx,[endpunkt]
368: ENDIF
369:     xor ch,ch
370:     mov dh,ch
371:     mov cl,[offtab1+bx]
372:     mov dl,[offtab2+bx]
373: IFIDNI <EndQuad>,<1>
374:     add cx,[xm]
375:     mov [WORD ArcEndX],cx
376:     mov cx,[y]
377:     sub cx,dx
378:     mov [WORD ArcEndY],cx
379: ENDIF
380: IFIDNI <EndQuad>,<2>
381:     add cx,[xm]
382:     mov [WORD ArcEndX],cx
383:     add dx,[y]
384:     mov [WORD ArcEndY],dx
385: ENDIF
386: IFIDNI <EndQuad>,<3>
387:     mov ax,[xm]
388:     sub ax,cx
389:     mov [WORD ArcEndX],ax
390:     add dx,[y]
391:     mov [WORD ArcEndY],dx
392: ENDIF
393: IFIDNI <EndQuad>,<4>
394:     mov ax,[xm]
395:     sub ax,cx
396:     mov [WORD ArcEndX],ax
397:     mov cx,[y]
398:     sub cx,dx
399:     mov [WORD ArcEndY],cx
400: ENDIF
401: ENDM
402: ;
403: ; Quadranten zeichnen
404: MACRO QuadLoop Quad,Index1,Index2
405:     LOCAL _LoopQ,_Noplot
406:     xor ch,ch
407:     mov dh,ch
408:     mov bx,[zaehl+Index1]
409:     mov cl,[offtab1+bx]
410: ; X-Offset holen
411:     mov dl,[offtab2+bx]
412: ; Y-Offset holen
413:     jmp _LoopQ
414: _LoopQ&Quad:
415:     mov bx,[zaehl+Index1]
416:     mov cl,[offtab1+bx]
417: ; X-Offset holen
418:     mov dl,[offtab2+bx]
419: ; Y-Offset holen
420:     cmp cl,[offtab1+bx-1]
421: ; Bildpunkt schon gesetzt ?
422:     jne _LoopQ
423:     cmp dl,[offtab2+bx-1]
424:     je _nplot
425: ; Ja,überspringen !
426: _LoopQ:
427:     mov ax,[xm]

```

Demoprogramm für den Grafikbefehl Curve
(Kreise und Ellipsen)


```

428: ; X-Koordinate Mittelpunkt holen
429: IFIDNI <Quad>, <1>
430: add ax, cx
431: ; x=xm + xoff
432: mov bx, [y]
433: ; Y-Koordinate holen
434: sub bx, dx
435: ; y=y - yoff
436: ENDIF
437: IFIDNI <Quad>, <2>
438: add ax, cx
439: ; x=xm + xoff
440: mov bx, [y]
441: add bx, dx
442: ; y=y + yoff
443: ENDIF
444: IFIDNI <Quad>, <3>
445: sub ax, cx
446: ; x=xm - xoff
447: mov bx, [y]
448: add bx, dx
449: ; y=y + yoff
450: ENDIF
451: IFIDNI <Quad>, <4>
452: sub ax, cx
453: ; x=xm - xoff
454: mov bx, [y]
455: sub bx, dx
456: ; y=y - yoff
457: ENDIF
458: call AsmPlot
459: ; Punkt setzen
460: _noplot:
461: mov ax, [zaehl&Index1]
462: cmp ax, [zaehl&Index2]
463: ; alle Punkte gesetzt?
464: je ExitLoopQ&Quad
465: ; Ja -!> Beenden
466: inc ax
467: ; Zähler erhöhen
468: mov [zaehl&Index1], ax
469: jmp LoopQ&Quad
470: ExitLoopQ&Quad:
471: ret
472: ENDM
473: ;
474: ; Koordinaten holen
475: PROC koor NEAR
476: mov [xm], ax
477: ; X-Koordinate Mittelpunkt
478: mov [y], bx
479: ; Y-Koordinate Mittelpunkt
480: mov [xr], cx
481: ; Radius X-Richtung
482: mov [yr], dx
483: ; Radius Y-Richtung
484: mov ax, es
485: ; Anfangswinkel in AX
486: call rechne
487: ; Anfangswinkel in Anfangsquadranten
488: ; und Anfangspunkt umrechnen
489: mov [anqua], ax
490: ; Anfangsquadrant
491: mov [anpunkt], bx
492: ; Anfangspunkt
493: mov ax, si
494: ; Endwinkel in AX
495: call rechne
496: ; Endwinkel in Endquadranten und
497: ; Endpunkt umrechnen
498: mov [endqua], ax
499: ; Endquadrant
500: mov [endpunkt], bx
501: ; Endpunkt
502: ret
503: ; Umrechnung: Winkel -> Quadrant und
504: ; Punkt rechne: push ax
505: ; wird später noch gebraucht
506: mov cx, 90
507: div cl
508: ; Winkel / 90
509: xor ah, ah
510: ; Divisionsrest löschen
511: ; => INT(Winkel / 90)
512: inc al
513: ; Quadrant=INT(Winkel / 90) + 1
514: mov dx, ax
515: ; Quadrant retten
516: dec al
517: mul cl
518: ; (Quadrant - 1) * 90
519: pop bx
520: ; Winkel holen
521: push dx
522: sub bx, ax
523: ; Teilwinkel=Winkel-(Quadrant-1)*90,
524: ;
525: ; 0 <= Teilwinkel < 90.
526: mov ax, bx
527: mov cx, 395
528: mul cx
529: ; Teilwinkel * 395, Ergebnis in DX:AX
530: mov cx, 90
531: div cx
532: ; Punkt=(Teilwinkel * 395) / 90.
533: mov bx, ax
534: ; Werte in richtigen Register bringen
535: pop ax
536: ret
537: ENDP

```

```

538: ;
539: ; Berechnung von AL * BX / 256
540: PROC Mult NEAR
541: xchg ax, bx
542: mul bl
543: ; AL * BX
544: REPT 8
545: shr ax, 1
546: ; AX 8 mal rechts schieben => AX/256
547: ENDM
548: ret
549: ENDP
550: ;
551: ; Ersten Quadranten zeichnen
552: PROC erster NEAR
553: mov [zaehl1], ax
554: mov [zaehl2], bx
555: mov bx, 394
556: ; Zähler setzen
557: sub bx, [zaehl1]
558: ; und invertieren
559: mov [zaehl1], bx
560: mov bx, 394
561: sub bx, [zaehl2]
562: mov [zaehl2], bx
563: QuadLoop 1, 2, 1
564: ENDP
565: ;
566: ; Zweiten Quadranten zeichnen
567: PROC zweiter NEAR
568: mov [zaehl1], ax
569: ; Zähler setzen
570: mov [zaehl2], bx
571: QuadLoop 2, 1, 2
572: ENDP
573: ;
574: ; Dritten Quadranten zeichnen
575: PROC dritter NEAR
576: mov [zaehl1], ax
577: ; Zähler setzen
578: mov [zaehl2], bx
579: mov ax, 394
580: sub ax, [zaehl1]
581: ; und invertieren
582: mov [zaehl1], ax
583: mov bx, 394
584: sub bx, [zaehl2]
585: mov [zaehl2], bx
586: QuadLoop 3, 2, 1
587: ENDP
588: ;
589: ; Vierten Quadranten zeichnen
590: PROC vierter NEAR
591: mov [zaehl1], ax
592: ; Zähler setzen
593: mov [zaehl2], bx
594: QuadLoop 4, 1, 2
595: ENDP
596: PROC AsmCurve
597: ;
598: ; Register retten
599: push ax
600: push bx
601: push cx
602: push dx
603: push es
604: push si
605: push di
606: call koor
607: ; Koordinaten zuweisen
608: cmp [anqua], 1
609: ; Start im 1. Quadranten?
610: jne ausschnitt
611: ; mit erstem Punkt
612: cmp [anpunkt], 0
613: ; Nur Ellipsenabschnitt zeichnen?
614: jne ausschnitt
615: cmp [endqua], 5
616: ; Ende wieder im 1. Quadranten?
617: jne ausschnitt
618: ; nur Ellipsenabschnitt
619: cmp [endpunkt], 0
620: ; Komplette Ellipse zeichnen?
621: jne ausschnitt
622: ; Nein -> Ellipsenabschnitt zeichnen
623: je ganz
624: ; Ja -> komplette Ellipse zeichnen
625: ausschnitt:
626: jmp aussch
627: ; Sprung zum Zeichnen e. Ellipsen-
628: ; abschnitts
629: ;
630: ; Komplette Ellipse zeichnen
631: ganz:
632: mov bx, -1
633: mov [xoff], bx
634: ; Variablen initialisieren
635: mov [yoff], bx
636: mov bx, [xm]
637: ; X-Koordinate Mittelpunkt in x merken
638: push ax
639: mov ax, [y]
640: ; Anfangs- und Endpunktskoordinaten
641: ; berechnen
642: ; und speichern
643: sub ax, [yr]
644: mov [WORD ArcStartX], bx
645: mov [WORD ArcEndX], bx
646: mov [WORD ArcStartY], ax
647: mov [WORD ArcEndY], ax

```

```

648: pop ax
649: mov [x], bx
650: mov [zaehl2], 0
651: ; Beide Zähler setzen
652: mov [zaehl1], 394
653: CurveLoop:
654: mov bx, [zaehl1]
655: xor ah, ah
656: mov al, [BYTE costab+bx]
657: ; Cosinus-Wert in AX
658: mov bx, [xr]
659: ; X-Radius in BX
660: call mult
661: ; AL*xr/256 = x-Offset berechnen
662: push ax
663: mov bx, [zaehl2]
664: ; Zähler2
665: xor ah, ah
666: mov al, [BYTE costab+bx]
667: ; Sinus-Wert holen
668: mov bx, [yr]
669: ; Y-Radius
670: call mult
671: ; AL*yr/256 = y-Offset berechnen
672: mov bx, ax
673: pop ax
674: cmp [xoff], ax
675: ; Punkt schon gesetzt?
676: jne setzpunkt
677: cmp [yoff], bx
678: je weiter
679: ; Ja, überspringen !!!
680: setzpunkt:
681: mov [xoff], ax
682: mov [yoff], bx
683: mov ax, [xm]
684: ; X-Koordinate Mittelpunkt in AX
685: add ax, [xoff]
686: ; x=xm + xoff
687: mov bx, [y]
688: ; Y-Koordinate in BX
689: add bx, [yoff]
690: ; y=y + yoff
691: mov cx, bx
692: ; y + yoff in CX retten
693: call AsmPlot
694: ; 1. Punkt setzen
695: mov bx, [y]
696: ; Y-Koordinate in BX
697: sub bx, [yoff]
698: ; y=y - yoff
699: mov dx, bx
700: ; y - yoff in DX retten
701: call AsmPlot
702: ; 2. Punkt setzen
703: mov ax, [xm]
704: ; X-Koordinate Mittelpunkt laden
705: sub ax, [xoff]
706: ; x=xm - xoff
707: mov bx, cx
708: ; y + yoff laden
709: call AsmPlot
710: ; 3. Punkt setzen
711: mov bx, dx
712: ; y - yoff laden
713: call AsmPlot
714: ; 4. Punkt setzen
715: weiter:
716: inc [zaehl2]
717: ; Zähler 1 erhöhen
718: dec [zaehl1]
719: ; Zähler 2 erniedrigen
720: jne CurveLoop
721: ; nächsten 4 Punkte berechnen
722: jmp CurveExit1
723: ; Beide Zähler gleich -> Beenden
724: ;
725: ; Ellipsenabschnitt zeichnen
726: aussch:
727: mov [zaehl1], 395
728: ; Beide Zähler laden
729: mov [zaehl2], 0
730: ausloop:
731: mov bx, [zaehl1]
732: ; Tabelle der Offsets in X- und
733: ; Y-Richtung aufbauen
734: xor ah, ah
735: mov al, [BYTE costab+bx]
736: mov bx, [xr]
737: call mult
738: mov bx, [zaehl2]
739: mov [offset1+bx], al
740: xor ah, ah
741: mov al, [BYTE costab+bx]
742: mov bx, [yr]
743: call mult
744: mov bx, [zaehl2]
745: mov [offset2+bx], al
746: inc [zaehl2]
747: dec [zaehl1]
748: jne ausloop
749: mov ax, [xm]
750: mov [x], ax
751: mov bx, [anqua]
752: ; Anfangsquadranten laden
753: shl bx, 1

```

Demoprogramm für den Grafikbefehl Curve
(Kreise und Ellipsen)


```

754:      shl bx,1
755: ; anqua * 4
756:      add bx,[endqua]
757: ; Endquadranten dazu addieren
758:      sub bx,5
759: ; 0 <= Indexwert <= 15
760:      shl bx,1
761: ; Indexwert * 8
762:      shl bx,1
763:      shl bx,1
764:      mov ax,OFFSET jumtab
765:      add ax,bx
766:      jmp ax
767: jumtab:
768:      call sle1
769:      jmp CurveExit1
770: ; Start und Ende im 1. Quadranten
771:      nop
772:      nop
773:      call sle2
774:      jmp CurveExit1
775: ; Start im 1.,Ende im 2. Quadranten
776:      nop
777:      nop
778:      call sle3
779:      jmp CurveExit1
780: ; Start im 1.,Ende im 3. Quadranten
781:      nop
782:      nop
783:      call sle4
784:      jmp CurveExit1
785: ; Start im 1.,Ende im 4. Quadranten
786:      nop
787:      nop
788:      call sle1
789:      jmp CurveExit1
790: ; Start im 2.,Ende im 1. Quadranten
791:      nop
792:      nop
793:      call sle2
794:      jmp CurveExit1
795: ; Start im 2.,Ende im 2. Quadranten
796:      nop
797:      nop
798:      call sle3
799:      jmp CurveExit1
800: ; Start im 2.,Ende im 3. Quadranten
801:      nop
802:      nop
803:      call sle4
804:      jmp CurveExit1
805: ; Start im 2.,Ende im 4. Quadranten
806:      nop
807:      nop
808:      call sle1
809:      jmp CurveExit1
810: ; Start im 3.,Ende im 1. Quadranten
811:      nop
812:      nop
813:      call sle2
814:      jmp CurveExit1
815: ; Start im 3.,Ende im 2. Quadranten
816:      nop
817:      nop
818:      call sle3
819:      jmp CurveExit1
820: ; Start im 3.,Ende im 3. Quadranten
821:      nop
822:      nop
823:      call sle4
824:      jmp CurveExit1
825: ; Start im 3.,Ende im 4. Quadranten
826:      nop
827:      nop
828:      call sle1
829:      jmp CurveExit1
830: ; Start im 4.,Ende im 1. Quadranten
831:      nop
832:      nop
833:      call sle2
834:      jmp CurveExit1
835: ; Start im 4.,Ende im 2. Quadranten
836:      nop
837:      nop
838:      call sle3
839:      jmp CurveExit1
840: ; Start im 4.,Ende im 3. Quadranten
841:      nop
842:      nop
843:      call sle4
844:      jmp CurveExit1
845: ; Start im 4.,Ende im 4. Quadranten
846: ;
847: ; Start im 1.,Ende im 1. Quadranten
848: sle1:
849:      CalcPoints 1,1
850:      mov ax,[anpunkt]
851:      mov bx,[endpunkt]
852:      cmp ax,bx
853:      jle sle11
854:      mov bx,394
855:      call erster
856:      xor ax,ax
857:      jmp sle2a
858: sle11: mov bx,[endpunkt]
859:      jmp erster
860: ;
861: ; Start im 1.,Ende im 2. Quadranten
862: sle2:
863:      CalcPoints 1,2

```

```

864:      mov ax,[anpunkt]
865: sle2b: mov bx,394
866:      call erster
867:      xor ax,ax
868: sle2a: mov bx,[endpunkt]
869:      jmp zweiter
870: ;
871: ; Start im 1.,Ende im 3. Quadranten
872: sle3:
873:      CalcPoints 1,3
874:      mov ax,[anpunkt]
875: sle3c: mov bx,394
876:      call erster
877:      xor ax,ax
878: sle3a: mov bx,394
879:      call zweiter
880:      xor ax,ax
881: sle3b: mov bx,[endpunkt]
882:      jmp dritter
883: ;
884: ; Start im 1.,Ende im 4. Quadranten
885: sle4:
886:      CalcPoints 1,4
887:      mov ax,[anpunkt]
888: sle4c: mov bx,394
889:      call erster
890:      xor ax,ax
891: sle4a: mov bx,394
892:      call zweiter
893:      xor ax,ax
894: sle4b: mov bx,394
895:      call dritter
896:      xor ax,ax
897: sle4d: mov bx,[endpunkt]
898:      jmp vierter
899: ;
900: ; Start im 2.,Ende im 1. Quadranten
901: sle1:
902:      CalcPoints 2,1
903:      mov ax,[anpunkt]
904: sle1a: mov bx,394
905:      call zweiter
906:      xor ax,ax
907: sle1b: mov bx,394
908:      call dritter
909:      xor ax,ax
910: sle1c: mov bx,394
911:      call vierter
912:      xor ax,ax
913:      jmp sle11
914: ;
915: ; Start im 2.,Ende im 2. Quadranten
916: sle2:
917:      CalcPoints 2,2
918:      mov ax,[anpunkt]
919:      mov bx,[endpunkt]
920:      cmp ax,bx
921:      jle sle22
922:      mov bx,394
923:      call zweiter
924:      xor ax,ax
925:      jmp sle2a
926: sle22: jmp sle2a
927: ;
928: ; Start im 2.,Ende im 3. Quadranten
929: sle3:
930:      CalcPoints 2,3
931:      mov ax,[anpunkt]
932:      jmp sle3a
933: ;
934: ; Start im 2.,Ende im 4. Quadranten
935: sle4:
936:      CalcPoints 2,4
937:      mov ax,[anpunkt]
938:      jmp sle4a
939: ;
940: ; Start im 3.,Ende im 1. Quadranten
941: sle1:
942:      CalcPoints 3,1
943:      mov ax,[anpunkt]
944:      jmp sle1b
945: ;
946: ; Start im 3.,Ende im 2. Quadranten
947: sle2:
948:      CalcPoints 3,2
949:      mov ax,[anpunkt]
950: sle2a: mov bx,394
951:      call dritter
952:      xor ax,ax
953: sle2b: mov bx,394
954:      call vierter
955:      xor ax,ax
956:      jmp sle2b
957: ;
958: ; Start im 3.,Ende im 3. Quadranten
959: sle3:
960:      CalcPoints 3,3
961:      mov ax,[anpunkt]
962:      mov bx,[endpunkt]
963:      cmp ax,bx
964:      jle sle33
965:      mov bx,394
966:      call dritter
967:      xor ax,ax
968:      jmp sle3a
969: sle33: jmp sle3b
970: ;
971: ; Start im 3.,Ende im 4. Quadranten
972: sle4:
973:      CalcPoints 3,4

```

```

974:      mov ax,[anpunkt]
975:      jmp sle4b
976: ;
977: ; Start im 4.,Ende im 1. Quadranten
978: sle1:
979:      CalcPoints 4,1
980:      mov ax,[anpunkt]
981:      jmp sle1c
982: ;
983: ; Start im 4.,Ende im 2. Quadranten
984: sle2:
985:      CalcPoints 4,2
986:      mov ax,[anpunkt]
987:      jmp sle2b
988: ;
989: ; Start im 4.,Ende im 3. Quadranten
990: sle3:
991:      CalcPoints 4,3
992:      mov ax,[anpunkt]
993: sle3a: mov bx,394
994:      call vierter
995:      xor ax,ax
996:      jmp sle3c
997: ;
998: ; Start im 4.,Ende im 4. Quadranten
999: sle4:
1000:      CalcPoints 4,4
1001:      mov ax,[anpunkt]
1002:      mov bx,[endpunkt]
1003:      cmp ax,bx
1004:      jle sle44
1005:      mov bx,394
1006:      call vierter
1007:      xor ax,ax
1008:      jmp sle4c
1009: sle44: jmp sle4d
1010: CurveExit1:
1011:      pop di
1012: ; Alle Registerinhalte wieder
1013: ; laden
1014:      pop si
1015:      pop es
1016:      pop dx
1017:      pop cx
1018:      pop bx
1019:      pop ax
1020:      ret
1021: ENDP
1022: GfxStart:
1023:      mov ax,@data
1024:      mov ds,ax
1025:      call AsmInitGraphic
1026: ; Grafikmodus einschalten
1027:      mov di,15
1028:      xor ax,ax
1029: ; Kreise zeichnen
1030:      mov es,ax
1031:      mov si,360
1032:      mov ax,320
1033:      mov bx,240
1034:      mov cx,10
1035:      mov dx,10
1036: _loop:
1037:      call AsmCurve
1038:      inc di
1039:      inc cx
1040:      inc dx
1041:      cmp dx,230
1042:      jne _loop
1043:      xor ah,ah
1044:      int 16h
1045:      call AsmInitGraphic
1046:      mov di,15
1047:      xor ax,ax
1048: ; Ellipsen zeichnen
1049:      mov es,ax
1050:      mov si,360
1051:      mov ax,320
1052:      mov bx,240
1053:      mov dx,225
1054:      mov cx,225
1055: _loop1:
1056:      call AsmCurve
1057:      inc di
1058:      sub cx,10
1059:      jnb _loop1
1060:      mov cx,225
1061: _loop2:
1062:      call AsmCurve
1063:      inc di
1064:      sub dx,10
1065:      jnb _loop2
1066: ;
1067:      xor ah,ah
1068:      int 16h
1069:      call AsmInitText
1070: ; Textmodus wieder einschalten
1071:      mov ax,4C00h
1072: ; Zurück zu DOS
1073:      int 21h
1074: END GfxStart

```

Demoprogramm für den Grafikbefehl Curve
(Kreise und Ellipsen)

Geh aufs Ganze - Flächen füllen in Rekordzeit

Das Füllen einer beliebigen umrandeten Fläche gehört zu den schwierigsten Aufgaben eines Grafikprogrammierers. Die im folgenden Beitrag vorgestellte Assembler-Routine zum Füllen beliebiger Flächen stellt nebenbei auf PCs mit VGA-Karten noch einen neuen Geschwindigkeitsrekord auf.

Die grundsätzliche Aufgabe beim Ausfüllen einer durch eine Umrandung begrenzten Fläche besteht darin, sich vom Ausgangspunkt in alle vier Richtungen bis zur Umrandung »vorzutasten« und jeden bearbeiteten Punkt mit der Füllfarbe zu belegen. In der Praxis führt die Umsetzung eines solchen punktorientierten Verfahrens schon bei kleineren Flächen zu indiskutablen Ausführungszeiten. Das folgende Verfahren arbeitet hingegen linienorientiert, wobei insbesondere die extrem hohe Zeichengeschwindigkeit einer waagerechten Linie gegenüber dem wesentlich langsameren Setzen einzelner Grafikpunkte ausgenutzt wird. Trotzdem würde man alleine mit dieser Methode auch noch keine zufriedenstellenden Ergebnisse erzielen, da die permanent erforderliche Farbabfrage der einzelnen Grafikpunkte erhebliche Verzögerungen mit sich bringt. Es wurden daher spezielle Routinen programmiert, die, ausgehend von einem Startpunkt innerhalb der Umrandung, sehr schnell den Rand in alle

vier Richtungen finden. Zunächst wollen wir jedoch die grundsätzliche Arbeitsweise des Algorithmus anhand einer Beispielfläche (Bild 1) aufzeigen. Als allererstes wird überprüft, ob der Startpunkt vielleicht schon gesetzt ist, woraufhin die Routine die Bearbeitung abbricht. Im anderen Fall muß so lange um eine Zeile nach oben gewandert werden, bis man einen gesetzten Punkt oder den Bildschirmrand erreicht. Sodann verzweigt der Algorithmus wieder um eine Zeile nach unten und tätigt gleichzeitig einen Schritt nach rechts. Dieses Vorgehen wiederholt sich so lange, bis man nicht nur oben gegen eine Begrenzung stößt, sondern auch rechts. Der rechte, obere Eckpunkt der Figur ist also erreicht. Den Weg dorthin zeigt Bild 1, in der Assembler-Routine befinden wir uns am Label La2. Das Modul merkt sich diesen Eckpunkt, da wir ihn später noch benötigen. Als nächstes wird jetzt so lange nach links gewandert, bis auch hier gegen eine Begrenzung gestoßen wird. Nachdem diese Koordina-

ten ebenfalls gespeichert wurden, zeichnet man eine Linie zwischen diesen beiden äußeren Punkten. Oberhalb der Linie existieren allerdings noch freie Punkte, die bislang nicht erfaßt werden konnten. Deshalb beschäftigt sich der nächste Schritt ab Label L3 damit, eine Zeile nach oben zu wandern und vom linken Endpunkt der Linie bis zum rechten Startpunkt zu prüfen, ob nicht gesetzte Punkte auftreten. Da die Analyse gerade die Zeile mit der Begrenzung der Figur untersucht, kann ein fehlender Punkt nur bedeuten, daß das Muster nach oben weiterläuft oder die Begrenzung ein Loch enthält, was allerdings sehr bitter wäre. Entdeckt das Programm also einen fehlenden Punkt, so ruft sich die Routine mit dieser Position als Startpunkt rekursiv selbst auf, das heißt, der gesamte Algorithmus wird nun auf das freie Feld oberhalb der gezeichneten Linie angewendet. Dafür sind natürlich zunächst alle Variablen zu sichern, damit die Füllroutine nach Beendigung der Rekursion mit den vorherigen Werten weiterrechnen kann und sie nicht

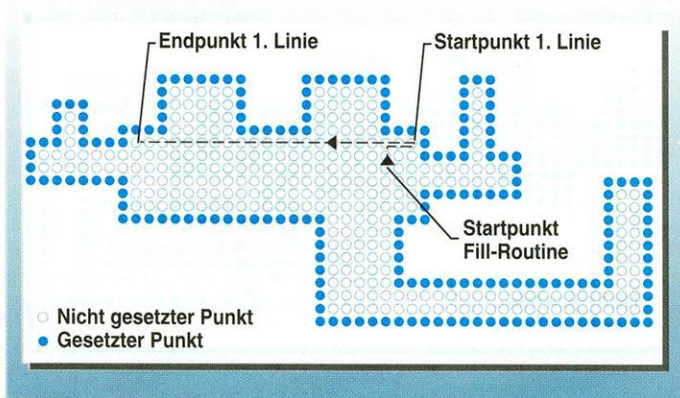


Bild 1. Der algorithmische Weg zum Startpunkt der Flächenfüllung

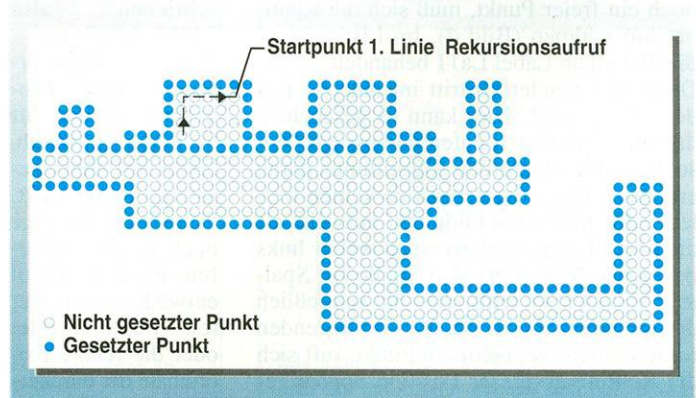


Bild 2. Füllung oberhalb der Startlinie und Erreichen der rechten Begrenzung

zwischenzeitlich zerstört werden.

Nachdem das Feld gefüllt ist, folgt die Suche nach weiteren Leerpunkten, bis man die rechte Begrenzung der Figur erreicht. In unserem Fall benötigen wir dementsprechend einen weiteren Rekursionsaufruf. Beim Erreichen der rechten Begrenzung stellt sich die Figur gemäß Bild 2 dar.

Nun wird die Figur zeilenweise nach unten abgearbeitet (ab Label L5). Wiederum ist die linke sowie rechte Begrenzung zu suchen und eine Linie zwischen beiden zu ziehen. Man muß jedoch aufpassen: Auch nach unten können sich noch nicht gefüllte Bereiche anschließen. Doch wie erkennt man diese Fälle?

Nun, die erste Situation taucht auf der linken Seite der Figur auf: Die linke Koordinate der zu zeichnenden Linie ist offenbar kleiner als die der darüberliegenden. Daraus läßt sich schließen, daß noch ein Flächenteil nach oben anschließt. Man prüft also in der darüberliegenden Zeile (die weiter rechts beginnende), ob noch ein freier Punkt links von diesem Anfangspunkt existiert, und zwar so lange, bis die linke Koordinate der darunter liegenden Linie erreicht ist. In unserem Beispiel findet sich ein freier Punkt, und die Programmsequenz wendet sich rekursiv auf diesen freien Flächenteil an (Bild 3). Der zugehörige Programmteil befindet sich ab Sprungmarke La9.

Die zweite Variante tritt ähnlich auf der rechten Seite auf: Hier ist die größere x-Koordinate der unteren Linie ein Indiz dafür, daß es noch verborgene Flächenteile oberhalb der unteren Linie geben könnte. Also wird in Höhe der oberen Linie so weit nach rechts gewandert, bis die rechte Koordinate der unteren, längeren Linie erreicht ist. Befindet sich in diesem Teil noch ein freier Punkt, muß sich die Routine ihm widmen (Bild 4). Im Listing wird dieser Fall ab Label La11 behandelt.

Der letzte Sonderfall tritt im unteren Teil des Musters auf. Hier kann es geschehen, daß ein Zipfel nach unten herausragt. Diese Ausnahme kann der Algorithmus daran erkennen, daß die untere Linie keinen weiter rechts liegenden Endpunkt aufweist als die obere Linie, sondern einen weiter links liegenden. Man prüft also Spalte für Spalte, ob ein Punkt fehlt, dies bis schließlich der rechte Endpunkt der darüberliegenden Linie erreicht ist. Fehlt ein Punkt, ruft sich die Routine ab dieser Position wieder rekursiv auf (Bild 5). Ab Label La10 steht der zugehörige Programmteil.

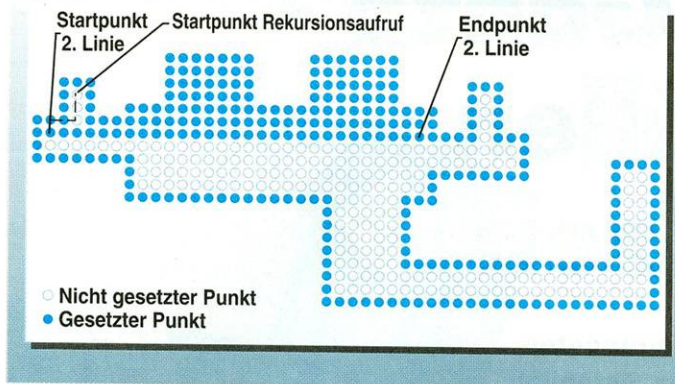


Bild 3. Die Füllung von Zipfeln auf der linken Seite erfolgt wiederum durch rekursiven Programmaufruf.

Was aber passiert, wenn wie in unserem Beispiel der untere Zipfel selbst noch einen weiteren nach oben aufweist? Nun, das Modul wird dann ganz einfach nochmals rekursiv aufgerufen, diesmal jedoch eine Ebene

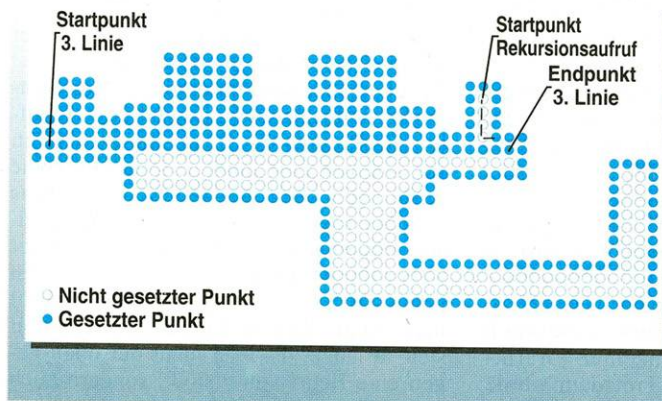


Bild 4. Noch nicht gefärbte Positionen oberhalb der dritten Linie vorhanden, also Füllung von Zipfeln auf der rechten Seite

tiefer. Die zulässige Verschachtelungstiefe legt nur die Größe des Stacks fest. Bei einer maximalen Verschachtelungstiefe von 10, die kaum eine Figur überbieten dürfte, werden jedesmal nur elf Wörter sowie Rücksprungsadresse auf den Stack geschrieben, also $24 \cdot 10 = 240$ Byte.

Auf diese Weise arbeitet das Programm Zeile für Zeile nach unten ab, bis es auf einen gesetzten Punkt trifft. Dann wird so lange nach rechts gelaufen, bis das Modul entweder einen leeren Punkt findet oder die rechte Koordinate der darüberliegenden Zeile erreicht wurde. In

letzterem Fall ist die Gesamtfüllung abgeschlossen.

Wie schon zuvor erwähnt, liefert dieser Algorithmus aber noch keine zufriedenstellenden Ergebnisse, wenn man zur Farbüberprüfung die im ersten Kursteil vorgestellte Routine *AsmTestPixel* verwendet. Der zeitliche Engpaß dieser Programmsequenz besteht darin, daß sie für jeden Punkt die Adresse im Video-RAM und die Bitmaske berechnen, einen umständlichen, bitweisen Vergleich (siehe Beschreibung des Read Modus 0 des Grafik-Controllers im ersten Kursteil) durchführen und weiterhin bei nebeneinanderliegenden Punkten ein Grafikbyte mehrfach auslesen muß, obwohl dies de facto nicht nötig wäre.

Die für die Performance außerordentlich wichtigen Routinen *SearchRight* und *SearchLeft*, die sehr häufig zum Einsatz kommen, um die linke beziehungsweise rechte Begrenzung der Figur in einer bestimmten Zeile zu ermitteln, arbeiten daher nach einem wesentlich effektiveren Verfahren.

Sie bedienen sich des bislang noch nicht beschriebenen Read-Modus 1 des Grafik-Controllers, der es gestattet, für ein ganzes Byte des Video-RAMs einen Vergleich der Farbe mit einem vorher definierten Vergleichswert durchzuführen.

Nach dem Laden der Latch-Register sind jeweils die vier korrespondierenden Bits zu einer Gruppe zusammengefaßt, so daß bei acht

Bits für alle Bytes des Video-RAMs acht Vierergruppen entstehen. Jede dieser acht Vierergruppen wird daraufhin mit einem Vergleichswert, der zuvor in das Color-Compare-Register des Grafik-Controllers

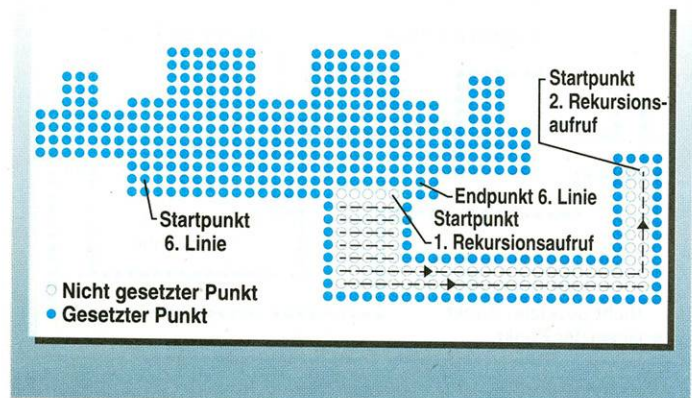


Bild 5. Prinzip beim Füllen von Zipfeln im unteren Bereich

geschrieben wurde, verglichen. Als Ergebnis des Vergleichs trifft ein Ergebnisbyte bei der CPU ein, in dem alle Bits gesetzt sind, deren korrespondierende Vierergruppe mit dem Wert aus dem Color-Compare-Register übereinstimmt. Mit dieser Information läßt sich sehr leicht feststellen, welche Bildpunkte einer bestimmten Farbe entsprechen (Bild 6). Eine weitere Geschwindigkeitssteigerung läßt sich dadurch erreichen, daß sofort mit dem nächsten Byte fortgefahren und auf die bitweise Überprüfung verzichtet wird, wenn im Vergleichsbyte kein Bit gesetzt ist.

»fill.asm« (Listing) setzt in Form eines Demoprogramms die beschriebene Flächenfüllroutine um. Zunächst wird dabei durch mehrfachen Aufruf der Linienroutine eine relativ komplexe Figur aufgebaut und diese nach einem Tastendruck mit Farbe gefüllt. Das vorgestellte Modul arbeitet zirka 20 Prozent schneller als die Füllroutine von Borland, die in den C-

und Pascal-Compilern zum Einsatz kommt und in bezug auf die Geschwindigkeit bislang als Maß aller Dinge galt. Eine weitere Steigerung der Effizienz ist nur noch ab 386-Prozessorarchitekturen denkbar, bei denen sich die vergleichsweise langsamen Zugriffe auf Variablen im Arbeitsspeicher vermeiden lassen, indem man hierfür die erweiterten Prozessorregister benutzt.

(Frank Riemenschneider/bj)

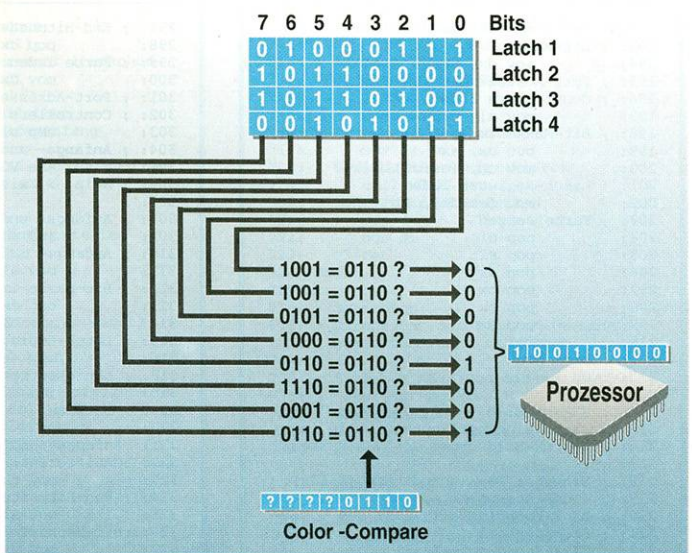


Bild 6. Funktion des Read-Modus 1 des Grafik-Controllers

Listing: fill.asm

```
1: ;Programm: fill.asm
2: ;Funktion: Demo-Programm für den
3: ; Grafikbefehl
4: ; FILL (Fläche füllen)
5: ;Sprache : Assembler
6: ;Autor : Frank Riemenschneider
7: ;(c) 1993 DMV Daten- und Medienverlag
8: ;
9: IDEAL
10: ; IDEAL-Modus einschalten
11: MODEL SMALL
12: ; Speichermodell SMALL wählen
13: STACK
14: db 1024 dup (?)
15: DATASEG
16: linetab dw 30,30,200,30,200,30,200,60
17: ,200,60,230
18: ; Tabelle mit
19: ; Koordinaten für
20: ; Linien, welche die
21: ; Umrandung bilden
22: dw 60,230,60,230,20,230,20,260,20
23: dw 260,20,260,180,260,180,400,180
24: dw 400,180,400,100,400,100,320,100
25: dw 320,100,320,30,320,30,470,30,470
26: dw 30,470,160,470,160,630,160,630,160
27: dw 630,440,630,440,100,440,100,440,
100
28: dw 360,100,360,20,360,20,360,20,290,
20
29: dw 290,100,290,100,290,100,270,100,
270
30: dw 470,230,470,230,30,230,30,230,30,
30
31: CODESEG
32: ;
33: ; ASMINITGRAPHIC:
34: ; Schaltet die VGA-Karte in den
35: ; Grafikmodus und löscht Bildschirm
36: ;
37: ; Eingabe: keine
38: ; Ausgabe: keine
39: PROC AsmInitGraphic
40: push ax
41: ; Für alle Fälle alle Register retten
42: push bx
43: push cx
44: push dx
45: push ds
46: push di
47: xor ah, ah
48: ; Funktionsnummer laden
49: mov al, 12h
50: ; Grafikmodus 640 x 480 Punkte in 16
51: int 10h
52: ; Farben setzen
53: mov dx, 3CEh
54: ; Port-Adresse des VGA-Grafik-
55: ; Controllers laden
56: mov ax, 0205h
57: ; Read-Modus 0 und Write-Modus 2
58: ; im Modusregister setzen
59: out dx, ax
60: mov ax, 0003h
61: ; Verknüpfungsmodus = Ersetzen im
62: ; Function-Select-Register setzen
```

```
63: out dx, ax
64: pop di
65: ; Alle Registerinhalte wieder
66: ; herstellen
67: pop ds
68: pop dx
69: pop cx
70: pop bx
71: pop ax
72: ret
73: ; Back home
74: ENDP
75: ;
76: ; ASMINITTEXT:
77: ; Schaltet die VGA-Karte in den Text-
78: ; modus
79: ; Eingabe: keine
80: ; Ausgabe: keine
81: PROC AsmInitText
82: push ax
83: ; Für alle Fälle alle Register retten
84: push bx
85: push cx
86: push dx
87: push ds
88: push di
89: mov dx, 3CEh
90: ; Port-Adresse des VGA-Grafik-
91: ; Controllers laden
92: mov ax, 0005h
93: ; Mode- und Function-Select-Register
94: ; mit ihren
95: ; Standardwerten laden
96: out dx, ax
97: mov ax, 0003h
98: out dx, ax
99: xor ah, ah
100: ; Funktionsnummer laden
101: mov al, 3
102: ; Textmodus 80 x 25 Zeichen in 16
103: ; Farben setzen
104: int 10h
105: pop di
106: ; Alle Registerinhalte wieder
107: ; herstellen
108: pop ds
109: pop dx
110: pop cx
111: pop bx
112: pop ax
113: ret
114: ; Back home
115: ENDP
116: ;
117: ; CALCXY:
118: ; Berechnet Segment und Offset des
119: ; Byte im Video-RAM, sowie die Bit-
120: ; Maske für das passende Bit bei
121: ; gegebener X- und Y-Koordinate
122: ;
123: ; Eingabe: AX = X-Koordinate (0-639)
124: ; BX = Y-Koordinate (0-479)
125: ; Ausgabe: ES = Segmentadresse des
126: ; Bytes im Video-RAM
127: ; DI = Offsetadresse des Bytes im
128: ; Video-RAM
129: ; DL = Bit-Maske des Bytes im
130: ; Video-RAM
```

```
131: PROC CalcXY
132: push ax
133: push bx
134: push cx
135: mov cx, ax
136: ; Koordinaten nochmal für spätere
137: ; Verwendung in der Berechnung retten
138: mov ax, 80
139: ; Pro Zeile 80 Bytes
140: mul bx
141: mov dx, 0A000h
142: ; Offset VGA-Karte laden
143: mov es, dx
144: ; Segmentadresse laden
145: mov dx, ax
146: mov ax, cx
147: shr ax, 1
148: shr ax, 1
149: shr ax, 1
150: ; x/8
151: add dx, ax
152: ; Spaltenoffset addieren
153: mov di, dx
154: ; DI mit Offset laden
155: ;
156: ; Berechnung der Bit-Maske
157: and cl, 7
158: ; x MOD 8
159: mov dl, 80h
160: ; Maske für Bit 0
161: shr dl, cl
162: ; Maske für gew. Bit durch Rotation
163: ; der ursprünglichen Maske erzeugen
164: pop cx
165: pop bx
166: pop ax
167: ret
168: ; Back home
169: ENDP CalcXY
170: ;
171: ; VGAPLOT:
172: ; Setzt einen Bildpunkt im Video-RAM
173: ; der VGA-Karte
174: ;
175: ; Eingabe: AX=X-Koordinate des Pixels
176: ; BX=Y-Koordinate des Pixels
177: ; DI=Farbe Low-Byte
178: ; Ausgabe: keine
179: PROC AsmPlot
180: push ax
181: push bx
182: push cx
183: push dx
184: push si
185: push di
186: mov cx, di
187: ; Farbe retten
188: call CalcXY
189: ; Offset- und Segmentadresse, sowie
190: ; die Bit-Maske aus den X- und
191: ; Y-Koordinaten berechnen
```

Das Demoprogramm »fill.asm« veranschaulicht ein optimiertes Prinzip zum Füllen von Flächen.


```

192:      mov ah, dl
193: ; Bit-Maske
194:      mov dx, 3CEh
195: ; Port-Adresse des VGA-Grafik-
196: ; Controllers laden
197:      mov al, 08h
198: ; Bit-Maske im Mask-Register setzen
199:      out dx, ax
200:      mov ch, [es:di]
201: ; Latch-Register laden
202:      mov [es:di], cl
203: ; Farbe setzen
204:      pop di
205:      pop si
206:      pop dx
207:      pop cx
208:      pop bx
209:      pop ax
210:      ret
211: ; Back home
212: ENDP
213: ;
214: ; VGAHLINe:
215: ; Zieht eine horizontale Linie im
216: ; Grafik-Bildschirm der VGA-Karte
217: ;
218: ; Eingabe: AX=X-Koordinate (0-639)
219: ; des Startpunktes
220: ; BX=Y-Koordinate (0-479) des
221: ; Startpunktes
222: ; CX=X-Koordinate (0-639) des
223: ; Endpunktes
224: ; DX=Y-Koordinate (0-479) des
225: ; Endpunktes
226: ; DI=Farbe Low-Byte
227: ; Ausgabe: keine
228: DATA SEG
229: StartTab db 1,3,7,15,31,63,127,255
230: EndTab db 255,254,252,248,240,224,
231:         192,128
232: CODE SEG
233: PROC VGAHLINe
234:      push ax
235: ; X- und Y-Koordinaten für spätere
236: ; Befehle retten
237:      push bx
238:      push cx
239:      push dx
240:      push si
241:      push di
242:      push di
243: ; Farbe wird später noch gebraucht
244: ;
245: ; Adressen holen
246:      push cx
247: ; Endkoordinaten und Karte retten
248:      push dx
249:      push di
250:      call CalcXY
251: ; Anfangsadresse aus den Koordinaten
252: ; berechnen
253:      mov si, di
254: ; Anfangsadressenoffset retten
255:      pop di
256: ; Endkoordinaten u. Karte wieder laden
257:      pop bx
258:      pop ax
259:      mov cx, dx
260: ; Anfangs-Bitmaske retten
261:      call CalcXY
262: ; Endadresse aus d. Koord. berechnen
263:      mov dh, cl
264: ; Anfangs-Bitmaske zurückschreiben
265: ;
266: ; Anfangs- und Endbitmuster bestimmen
267:      xor cx, cx
268: ; Zähler initialisieren
269: VGABitStart:
270:      shr dh, 1
271: ; Anfangs-Bitmaske rotieren
272:      inc ch
273: ; Zähler erhöhen
274:      jnc VGABitStart
275: ; Solange rotieren, bis gesetzte Bit im
276: ; Carry ist
277:      dec ch
278: ; Zähler an Bitnummer anpassen
279: VGABitEnd:
280:      shr dl, 1
281: ; End-Bitmaske rotieren
282:      inc cl
283: ; Zähler erhöhen
284:      jnc VGABitEnd
285: ; Solange rotieren, bis das gesetzte
286: ; Bit im Carry ist
287:      dec cl
288: ; Zähler an Bitnummer anpassen
289:      xor bx, bx
290:      mov bl, ch
291: ; Indexregister laden
292:      mov ah, [BYTE StartTab+bx]
293: ; Start-Bitmuster laden
294:      mov bl, cl
295: ; Indexregister laden
296:      mov al, [BYTE EndTab+bx]

```

```

297: ; End-Bitmuster laden
298:      pop cx
299: ; Farbe laden
300:      mov dx, 3CEh
301: ; Port-Adresse des VGA-Grafik-
302: ; Controllers laden
303:      cmp si, di
304: ; Anfangs- und Endadresse gleich?
305:      jne VGADifferent
306: ; Nein -> Zwischenbytes setzen
307: ;
308: ; Anfangs- und Endadresse sind gleich
309:      and ah, al
310: ; Anfangs- u. End-Bitmuster AND-verkn.
311:      mov al, 08h
312: ; Bit-Maske im Mask-Register setzen
313:      out dx, ax
314:      mov bl, [es:di]
315: ; Latch-Register laden
316:      mov [es:di], cl
317: ; Linie setzen
318:      jmp VGAHLINeExit
319: ; Beenden
320: ;
321: ; Anfangs- und Endadresse sind versch.
322: VGADifferent:
323:      mov ch, al
324: ; End-Bitmuster retten
325:      mov al, 08h
326: ; Bit-Maske im Mask-Register setzen
327:      out dx, ax
328:      mov bl, [es:si]
329: ; Latch-Register laden
330:      mov [es:si], cl
331: ; Farbe im Video-RAM setzen
332:      inc si
333: ; Zeiger erhöhen
334:      cmp si, di
335: ; Endadresse erreicht?
336:      jne VGAHLINeLab1
337: ; Nein -> weiter hinten
338:      mov ah, ch
339: ; End-Bitmuster laden
340:      out dx, ax
341: ; Bit-Maske im Mask-Register setzen
342:      mov bl, [es:si]
343: ; Latch-Register laden
344:      mov [es:si], cl
345: ; Farbe im Video-RAM setzen
346:      jmp VGAHLINeExit
347: ; Beenden
348: VGAHLINeLab1:
349:      mov ah, 0FFh
350: ; Zwischen-Bitmuster laden
351: ; und im Bit-Mask-Register setzen
352:      out dx, ax
353: VGAHLINeLab2:
354:      mov bl, [es:si]
355: ; Latch-Register laden
356:      mov [es:si], cl
357: ; Farbe im Video-RAM setzen
358:      inc si
359: ; Zeiger erhöhen
360:      cmp si, di
361: ; Endadresse erreicht?
362:      jne VGAHLINeLab2
363: ; Nein -> weitere Bytes löschen
364:      mov ah, ch
365: ; End-Bitmuster laden,
366:      out dx, ax
367: ; im Bit-Mask-Register
368:      mov bl, [es:si]
369: ; Latch-Register laden
370:      mov [es:si], cl
371: ; Farbe im Video-RAM setzen
372: VGAHLINeExit:
373:      pop di
374:      pop si
375:      pop dx
376:      pop cx
377:      pop bx
378: ; X- und Y-Koordinaten restaurieren
379:      pop ax
380:      ret
381: ; Back home
382: ENDP VGAHLINe
383: ;
384: ; VGAHLINe:
385: ; Zieht eine vertikale Linie im
386: ; Grafik-Bildschirm der VGA-Karte
387: ;
388: ; Eingabe: AX=X-Koordinate (0-639)
389: ; des Startpunktes
390: ; BX=Y-Koordinate (0-479) des
391: ; Startpunktes
392: ; CX=X-Koordinate (0-639) des
393: ; Endpunktes
394: ; DX=Y-Koordinate (0-479) des
395: ; Endpunktes
396: ; Ausgabe: keine
397: CODE SEG
398: PROC VGAHLINe
399:      push ax
400: ; X- und Y-Koordinaten für spätere
401: ; Befehle retten

```

```

402:      push bx
403:      push cx
404:      push dx
405:      push si
406:      push di
407:      mov bp, 0A000h
408:      mov es, bp
409: ; Segmentadresse laden
410:      shr ax, 1
411: ; x/8 (Es ist schneller den Befehl
412: ; dreimal abzuarbeiten als die
413: ; Rotation in einem Befehl)
414:      shr ax, 1
415:      shr ax, 1
416: ; Berechnung der Bit-Maske
417:      and cl, 7
418: ; x MOD 8
419:      mov ch, 80h
420: ; Maske für Bit 0
421:      shr ch, cl
422: ; Maske für gewünschtes Bit durch
423: ; Rotation der urspr. Maske erzeugen
424:      push dx
425: ; Endkoordinate retten
426:      push ax
427: ; X-Koordinate retten
428:      mov dx, 3CEh
429: ; Port-Adresse d. VGA-Grafik-
430: ; Controllers laden
431:      mov ah, ch
432: ; Bit-Maske im Mask-Register setzen
433:      mov al, 08h
434:      out dx, ax
435:      mov cx, di
436: ; Farbe in CL laden
437: ; Offsetadresse berechnen
438:      mov ax, 80
439: ; Zeilenanfang berechnen
440:      mul bx
441:      mov di, ax
442:      pop ax
443:      pop dx
444:      add di, ax
445: ; Spaltenoffset addieren
446: VGAVLineLoop:
447:      mov ch, [es:di]
448: ; Latch-Register laden
449:      mov [es:di], cl
450: ; Punkt setzen
451:      add di, 80
452: ; Nächste Zeile
453:      inc bx
454: ; Y-Koordinate erhöhen
455:      cmp bx, dx
456: ; Endpunkt erreicht?
457:      jbe VGAVLineLoop
458: ; Weitere Punkte setzen
459: VGAVLineExit:
460:      pop di
461:      pop si
462:      pop dx
463:      pop cx
464:      pop bx
465: ; X- und Y-Koordinaten restaurieren
466:      pop ax
467:      ret
468: ; Back home
469: ENDP VGAVLine
470: ;
471: ; ASMLINE:
472: ; Zieht Linie im Grafik-Bildschirm
473: ;
474: ; Eingabe: AX=X-Koordinate (0-639)
475: ; des Startpunktes
476: ; BX=Y-Koordinate (0-479) des
477: ; Startpunktes
478: ; CX=X-Koordinate (0-639) des
479: ; Endpunktes
480: ; DX=Y-Koordinate (0-479) des
481: ; Endpunktes
482: ; DI=Farbe Low-Byte
483: ; Ausgabe: keine
484: DATA SEG
485:      DeltaX dw ?
486: ; Steigung in X-Richtung
487:      DeltaY dw ?
488: ; Steigung in Y-Richtung
489:      SignY dw ?
490: ; Vorzeichen Y-Richtung
491:      Incl dw ?
492: ; Inkrement 1
493:      Inc2 dw ?
494: ; Inkrement 2
495: CODE SEG
496: PROC AsmLine
497:      push ax
498:      push bx
499:      push cx

```

Das Demoprogramm »fill.asm« veranschaulicht ein optimiertes Prinzip zum Füllen von Flächen.


```

500:      push dx
501:      push si
502:      push di
503:      push bp
504:      cmp bx,dx
505: ; Horizontale Linie?
506:      jne LineLabA
507:      jmp HLine
508: LineLabA:
509:      cmp ax,cx
510: ; Vertikale Linie?
511:      jne LineLabB
512:      jmp VLine
513: LineLabB:
514:      mov [DeltaX],cx
515: ; Steigung in X-Richtung berechnen
516:      mov bp,1
517: ; Positives Vorzeichen annehmen
518:      sub [DeltaX],ax
519:      jg LineLab9
520: ; Steigung negativ?
521:      neg [DeltaX]
522: ; Ja -> DeltaX komplementieren
523:      mov bp,-1
524: ; Negatives Vorzeichen in X-Richtung
525: LineLab9:
526:      mov [DeltaY],dx
527: ; Steigung in Y-Richtung berechnen
528:      mov [SignY],1
529: ; Positives Vorzeichen annehmen
530:      sub [DeltaY],bx
531:      jg LineLab11
532: ; Steigung negativ?
533:      neg [DeltaY]
534: ; Ja -> DeltaY komplementieren
535:      mov [SignY],-1
536: ; Negatives Vorzeichen in Y-Richtung
537: LineLab11:
538:      push ax
539: ; Koordinate retten, um das Register
540: ; zum Arbeiten freizubekommen
541:      mov ax,[DeltaX]
542:      cmp ax,[DeltaY]
543: ; Größe der Steigungen in X- und
544: ; Y-Richtung vergleichen
545:      pushf
546: ; Ergebnis merken
547:      jge LineLab12
548: ; Ja -> Steigungen nicht tauschen
549:      mov ax,[DeltaX]
550: ; Steigungen vertauschen
551:      xchg [DeltaY],ax
552:      mov [DeltaX],ax
553: LineLab12:
554:      mov ax,[DeltaY]
555: ; Steigung Y-Richtung laden,
556:      shl ax,1
557: ; mit 2 multiplizieren,
558:      sub ax,[DeltaX]
559: ; davon Steigung X-Richtung abziehen
560:      mov si,ax
561: ; Ergebnis ist der Startwert
562:      mov ax,[DeltaX]
563: ; Inkrement 2 berechnen
564:      shl ax,1
565: ; als Doppeltes d. Steigung X-Richtung
566:      mov [Inc2],ax
567:      mov ax,[DeltaY]
568: ; Inkrement 1 berechnen
569:      shl ax,1
570: ; als Doppeltes d. Steigung Y-Richtung
571:      mov [Inc1],ax
572:      popf
573: ; Flags zurückholen
574:      jnge gr45grad
575: ; Algorithmus für Steigungen > 45 Grad
576: ;
577: ; Bresenham-Algorithmus
578: ; für Steigungen <= 45 Grad
579:      pop ax
580: ; X-Startpunkt wiederherstellen
581:      call AsmPlot
582: ; Punkt setzen
583: LineWhileLoop:
584:      cmp ax,cx
585: ; X-Koordinate Endpunkt erreicht?
586:      jne LineLab13
587: ; Nein -> weiter plotten
588:      cmp bx,dx
589: ; Y-Koordinate Endpunkt erreicht?
590:      jne LineLab13
591: ; Nein -> weiter plotten
592:      jmp LineExit
593: ; Ja -> Beenden
594: LineLab13:
595:      cmp si,0
596: ; Startwert >= 0?
597:      jl LineLab16
598: ; Nein -> weiter hinten
599:      add bx,[SignY]
600: ; Y-Koordinate erhöhen
601:      sub si,[Inc2]
602: ; Startwert um Inkrement 2 erniedrigen
603: LineLab16:
604:      add ax,bp

```

```

605: ; X-Koordinate erhöhen
606:      add si,[Inc1]
607: ; Startwert um Inkrement 1 erhöhen
608:      call AsmPlot
609: ; Punkt setzen
610:      jmp LineWhileLoop
611: ; Zurück zum Schleifenanfang
612: ;
613: ; Bresenham-Algor. f. Steigungen > 45
614: ; Grad
615: gr45grad:
616:      pop ax
617: ; X-Startpunkt wiederherstellen
618:      call AsmPlot
619: ; Punkt setzen
620: LineWhileLoop:
621:      cmp ax,cx
622: ; X-Koordinate Endpunkt erreicht?
623:      jne LineLab13
624: ; Nein -> weiter plotten
625:      cmp bx,dx
626: ; Y-Koordinate Endpunkt erreicht?
627:      jne LineLab13
628: ; Nein -> weiter plotten
629:      jmp LineExit
630: ; Ja -> Beenden
631: LineLab13:
632:      cmp si,0
633: ; Startwert >= 0?
634:      jl LineLab16
635: ; Nein -> weiter hinten
636:      add ax,bp
637: ; X-Koordinate erhöhen
638:      sub si,[Inc2]
639: ; Startwert um Inkrement 2 erniedrigen
640: LineLab16:
641:      add bx,[SignY]
642: ; Y-Koordinate erhöhen
643:      add si,[Inc1]
644: ; Startwert um Inkrement 1 erhöhen
645:      call AsmPlot
646: ; Punkt setzen
647:      jmp LineWhileLoop
648: ; Zurück zum Schleifenanfang
649: ;
650: ; Vertikale Linie ziehen
651: VLine:
652:      cmp bx,dx
653: ; YAnf > YEnde?
654:      jle VLineLab1
655:      xchg bx,dx
656: VLineLab1:
657:      call VGAVLine
658: ; VGA: Vertikale Linie ziehen
659:      jmp LineExit
660: ;
661: ; Horizontale Linie ziehen
662: HLine:
663:      cmp ax,cx
664: ; XAnf > XEnde?
665:      jle HLineLab1
666:      xchg ax,cx
667: HLineLab1:
668:      call VGAHLine
669: ; VGA: Horizontale Linie ziehen
670: LineExit:
671:      pop bp
672:      pop di
673: ; Alle Register vom Stack zurückholen
674: ;
675:      pop si
676:      pop dx
677:      pop cx
678:      pop bx
679:      pop ax
680:      ret
681: ; Back home
682: ENDP
683: DATA SEG
684: ;
685: ; Benötigte Variablen
686: speicher21 dw ?
687: speicher22 dw ?
688: ; Speicher für letzte Zeile (rechts)
689: speicher31 dw ?
690: speicher32 dw ?
691: ; Speicher für aktuelle Zeile (links)
692: speicher41 dw ?
693: speicher42 dw ?
694: ; Speicher für aktuelle Zeile (rechts)
695: tcolor dw ?
696: ; Umrandungsfarbe
697: fillcolor dw ?
698: ; Füllfarbe
699: CODE SEG
700: ;
701: ; ZieheLinie:
702: ; Zeichnet waagerechte Linie in
703: ; Füllfarbe
704: ; Eingabe: AX=X-Koordinate (0-639)
705: ; des Startpunktes
706: ; BX=Y-Koordinate (0-479) des
707: ; Startpunktes
708: ; CX=X-Koordinate (0-639) des
709: ; Endpunktes

```

```

710: ; DX=Y-Koordinate (0-479) des
711: ; Endpunktes
712: ; Ausgabe: Keine
713: PROC ZieheLinie NEAR
714:      push di
715:      push dx
716:      mov di,[FillColor]
717:      mov dx,bx
718: ; Horizontale Linie (Y1=Y2)
719:      call VGAHLine
720: ; Line-Prozedur aufrufen
721:      pop dx
722:      pop di
723:      ret
724: ENDP
725: ;
726: ; SearchTop:
727: ; Durchsucht den Bildschirm nach
728: ; gesetztem Punkt nach oben
729: ;
730: ; Eingabe: AX=X-Koordinate (0-639)
731: ; des Startpunktes
732: ; BX=Y-Koordinate (0-479) des
733: ; Startpunktes
734: ; DX=Y-Koordinate (0-479) des
735: ; Endpunktes
736: ; Ausgabe: AX=X-Koordinate
737: ; BX=Y-Koordinate
738: ; Zero-Flag: Nicht gesetzt, dann
739: ; gesetzter Punkt gefunden
740: ;
741: ; Gesetzt, also kein
742: ; gesetzter Punkt gefunden
743: ;
744: PROC SearchTop NEAR
745:      push di
746:      push bp
747:      push cx
748:      push dx
749:      push dx
750: ; Endkoordinate retten
751:      call CALCXY
752: ; Adr. des Startpunktes im Video-RAM
753: ; suchen
754:      pop bp
755: ; Endkoordinate holen
756:      mov cl,dl
757: ; Bit-Maske retten
758:      push ax
759: ; X-Koordinate retten
760:      mov dx,03CEh
761:      mov ax,0A05h
762: ; Read-Modus 1 setzen
763:      out dx,ax
764: TopLoop:
765:      mov ah,[BYTE Fillcolor]
766: ; Mit Füllfarbe vergleichen
767:      mov al,02h
768:      out dx,ax
769:      mov ch,[es:di]
770: ; Farbbyte holen und vergleichen
771:      mov ah,[BYTE tcolor]
772: ; Mit Randfarbe vergleichen
773:      out dx,ax
774:      or ch,[es:di]
775:      and ch,cl
776: ; Bit-Maske isolieren
777:      jne FoundTop
778: ; Bit->0, dann gesetzter Punkt gefunden
779:      cmp bx,bp
780: ; Endkoordinate erreicht ?
781:      je FoundTop
782: ; Ja, beenden
783:      dec bx
784: ; Zeile vermindern
785:      sub di,80
786: ; Adr. im Video-RAM, um 80 vermindern
787:      jmp TopLoop
788: ; Weitersuchen
789: FoundTop:
790:      mov dx,03CEh
791:      mov ax,0205h
792: ; Read-Modus 0 und Write-Modus 2
793:      out dx,ax
794: ; setzen
795:      pop ax
796:      pop dx
797:      pop cx
798:      pop bp
799:      pop di
800:      ret
801: ENDP
802: ;
803: ; SearchLeft:
804: ; Durchsucht den Bildschirm nach
805: ; gesetztem Punkt nach links
806: ;
807: ; Eingabe: AX=X-Koordinate (0-639)

```

Das Demoprogramm »fill.asm« veranschaulicht ein optimiertes Prinzip zum Füllen von Flächen.


```

808: ; des Startpunktes
809: ; BX=Y-Koordinate (0-479) des
810: ; Startpunktes
811: ; DX=X-Koordinate (0-639) des
812: ; Endpunktes
813: ; Ausgabe: AX=X-Koordinate
814: ; BX=Y-Koordinate
815: ; Zero-Flag : Nicht gesetzt, dann
816: ; gesetzter Punkt gefunden
817: ;
818: ; : Gesetzt,also kein
819: ; gesetzter Punkt gefunden
820: ;
821: PROC SearchLeft NEAR
822: push di
823: push bx
824: push bp
825: push cx
826: push dx
827: push ax
828: ; Startkoordinate retten
829: push dx
830: ; Endkoordinate retten
831: call CALCX
832: ;Adr. des Startpunktes im Video-RAM
833: ;suchen
834: pop bp
835: ; Endkoordinate holen
836: mov cl,dl
837: ; Bit-Maske retten
838: mov dx,03CEh
839: mov ax,0A05h
840: ; Read-Modus 1 setzen
841: out dx,ax
842: mov ah,[BYTE tcolor]
843: ; Mit Randfarbe vergleichen
844: mov al,02h
845: out dx,ax
846: pop ax
847: LeftLoop:
848: mov ch,[es:di]
849: ; Farbbyte holen und vergleichen
850: mov dh,ch
851: ; Ergebnis retten
852: cmp cl,1
853: ; Anfang eines Grafik-Byte ?
854: jne LeftLoop
855: or dh,dh
856: ; Ja,prüfen,ob Grafik-Byte=0
857: jne LeftLoop
858: ; Nein,weiter
859: push dx
860: mov dx,ax
861: ; Ja,Endpunkt darin enthalten ?
862: sub dx,bp
863: cmp dx,8
864: pop dx
865: jb LeftLoop
866: sub ax,8
867: ; Ja,nächstes Byte
868: dec di
869: jmp LeftLoop
870: LeftLoop1:
871: mov ch,dh
872: ; Vergleichsergebnis holen
873: and ch,cl
874: ; Bit-Maske isolieren
875: jne FoundLeft
876: ; Bit<>0,dann gesetzter Punkt gefunden
877: cmp ax,bp
878: ; Endpunkt erreicht,dann Ende
879: je FoundLeft
880: dec ax
881: ; Spalte vermindern
882: shl cl,1
883: ; Grafik-Byte "aufgebraucht" ?
884: jnc LeftLoop
885: dec di
886: ; Grafik-Byte vermindern
887: mov cl,1
888: ; Bit-Maske neu setzen
889: jmp LeftLoop
890: ; und auslesen
891: FoundLeft:
892: push ax
893: mov dx,03CEh
894: mov ax,0205h
895: ; Read-Modus 0 und Write-Modus 2
896: out dx,ax
897: ; setzen
898: pop ax
899: pop dx
900: pop cx
901: pop bp
902: pop bx
903: pop di
904: ret
905: ENDP
906: ;
907: ; SearchNotLeft:
908: ; Durchsucht den Bildschirm nach
909: ; NICHT gesetztem Punkt nach links
910: ;
911: ; Eingabe: AX=X-Koordinate (0-639)
912: ; des Startpunktes

```

```

913: ; BX=Y-Koordinate (0-479) des
914: ; Startpunktes
915: ; DX=X-Koordinate (0-639) des
916: ; Endpunktes
917: ; Ausgabe: AX=X-Koordinate
918: ; BX=Y-Koordinate
919: PROC SearchNotLeft NEAR
920: push di
921: push bx
922: push bp
923: push cx
924: push dx
925: push ax
926: ; Startkoordinate retten
927: push dx
928: ; Endkoordinate retten
929: call CALCX
930: ; Adresse des Startpunktes im Video-
931: ; RAM suchen
932: pop bp
933: ; Endkoordinate holen
934: mov cl,dl
935: ; Bit-Maske retten
936: mov dx,03CEh
937: mov ax,0A05h
938: ; Read-Modus 1 setzen
939: out dx,ax
940: pop ax
941: ; Startkoordinate holen
942: NotLeftLoop:
943: push ax
944: mov ah,[BYTE Fillcolor]
945: ; Mit Fillfarbe vergleichen
946: mov al,02h
947: out dx,ax
948: mov ch,[es:di]
949: ; Farbbyte holen und vergleichen
950: mov ah,[BYTE tcolor]
951: ; Mit Randfarbe vergleichen
952: out dx,ax
953: or ch,[es:di]
954: pop ax
955: mov dh,ch
956: ; Ergebnis retten
957: cmp cl,1
958: ; Anfang eines Grafik-Byte ?
959: jne NotLeftLoop
960: or dh,dh
961: ; Ja,prüfen,ob Grafik-Byte <> 0
962: je NotLeftLoop
963: ; Nein,weiter
964: push dx
965: mov dx,ax
966: ; Ja,Endpunkt darin enthalten ?
967: sub dx,bp
968: cmp dx,8
969: pop dx
970: jb NotLeftLoop
971: sub ax,8
972: ; Ja,nächstes Byte
973: dec di
974: jmp NotLeftLoop
975: NotLeftLoop1:
976: mov ch,dh
977: ; Vergleichsergebnis holen
978: and ch,cl
979: ; Bit-Maske isolieren
980: je FoundNotLeft
981: ; Bit=0,dann NICHT gesetzter Punkt
982: ; gefunden !
983: cmp ax,bp
984: ; Endpunkt erreicht,dann Ende
985: je FoundNotLeft
986: dec ax
987: ; Spalte vermindern
988: shl cl,1
989: ; Grafik-Byte "aufgebraucht" ?
990: jnc NotLeftLoop
991: dec di
992: ; Grafik-Byte vermindern
993: mov cl,1
994: ; Bit-Maske neu setzen
995: jmp NotLeftLoop
996: ; und auslesen
997: FoundNotLeft:
998: push ax
999: mov dx,03CEh
1000: mov ax,0205h
1001: ; Read-Modus 0 und Write-Modus 2
1002: out dx,ax
1003: ; setzen
1004: pop ax
1005: pop dx
1006: pop cx
1007: pop bp
1008: pop bx
1009: pop di
1010: ret
1011: ENDP
1012: ;
1013: ; SearchRight:
1014: ; Durchsucht den Bildschirm nach
1015: ; gesetztem Punkt nach rechts
1016: ;

```

```

1017: ; Eingabe: AX=X-Koordinate (0-639)
1018: ; des Startpunktes
1019: ; BX=Y-Koordinate (0-479) des
1020: ; Startpunktes
1021: ; DX=X-Koordinate (0-639) des
1022: ; Endpunktes
1023: ; Ausgabe: AX=X-Koordinate
1024: ; BX=Y-Koordinate
1025: PROC SearchRight NEAR
1026: push di
1027: push bx
1028: push bp
1029: push cx
1030: push dx
1031: push ax
1032: ; Startkoordinate retten
1033: push dx
1034: ; Endkoordinate retten
1035: call CALCX
1036: ; Adresse des Startpunktes im Video-
1037: ; RAM suchen
1038: pop bp
1039: ; Endkoordinate holen
1040: mov cl,dl
1041: ; Bit-Maske retten
1042: mov dx,03CEh
1043: mov ax,0A05h
1044: ; Read-Modus 1 setzen
1045: out dx,ax
1046: mov ah,[BYTE tcolor]
1047: ; Mit Randfarbe vergleichen
1048: mov al,02h
1049: out dx,ax
1050: pop ax
1051: RightLoop:
1052: mov ch,[es:di]
1053: ; Farbbyte holen und vergleichen
1054: mov dh,ch
1055: ; Ergebnis retten
1056: cmp cl,128
1057: ; Anfang eines Grafik-Byte ?
1058: jne RightLoop
1059: or dh,dh
1060: ; Ja,prüfen,ob Grafik-Byte == 0
1061: jne RightLoop
1062: ; Nein,weiter
1063: push dx
1064: mov dx,bp
1065: ; Ja,Endpunkt darin enthalten ?
1066: sub dx,ax
1067: cmp dx,8
1068: pop dx
1069: jb RightLoop
1070: add ax,8
1071: ; Ja,nächstes Byte
1072: inc di
1073: jmp RightLoop
1074: RightLoop1:
1075: mov ch,dh
1076: ; Vergleichsergebnis holen
1077: and ch,cl
1078: ; Bit-Maske isolieren
1079: jne FoundRight
1080: ; Bit=1,dann gesetzter Punkt gefunden
1081: cmp ax,bp
1082: ; Endpunkt erreicht,dann Ende
1083: je FoundRight
1084: inc ax
1085: ; Spalte erhöhen
1086: shr cl,1
1087: ; Grafik-Byte "aufgebraucht" ?
1088: jnc RightLoop
1089: inc di
1090: ; Grafik-Byte erhöhen
1091: mov cl,128
1092: ; Bit-Maske neu setzen
1093: jmp RightLoop
1094: ; und auslesen
1095: FoundRight:
1096: push ax
1097: mov dx,03CEh
1098: mov ax,0205h
1099: ; Read-Modus 0 und Write-Modus 2
1100: out dx,ax
1101: ; setzen
1102: pop ax
1103: pop dx
1104: pop cx
1105: pop bp
1106: pop bx
1107: pop di
1108: ret
1109: ENDP
1110: ;
1111: ; SearchNotRight:
1112: ; Durchsucht den Bildschirm nach
1113: ; NICHT gesetztem Punkt nach rechts
1114: ;

```

Das Demoprogramm »fill.asm« veranschaulicht ein optimiertes Prinzip zum Füllen von Flächen.


```

1115: ; Eingabe: AX=X-Koordinate (0-639)
1116: ; des Startpunktes
1117: ; BX=Y-Koordinate (0-479) des
1118: ; Startpunktes
1119: ; DX=X-Koordinate (0-639) des
1120: ; Endpunktes
1121: ; Ausgabe: AX=X-Koordinate
1122: ; BX=Y-Koordinate
1123: PROC SearchNotRight NEAR
1124:     push di
1125:     push bx
1126:     push bp
1127:     push cx
1128:     push dx
1129:     push ax
1130: ; Startkoordinate retten
1131:     push dx
1132: ; Endkoordinate retten
1133:     call CALCXy
1134: ; Adresse des Startpunktes im Video-
1135: ; RAM suchen
1136:     pop bp
1137: ; Endkoordinate holen
1138:     mov cl,dl
1139: ; Bit-Maske retten
1140:     mov dx,03CEh
1141:     mov ax,0A05h
1142: ; Read-Modus 1 setzen
1143:     out dx,ax
1144:     pop ax
1145: ; Startkoordinate holen
1146: NotRightLoop:
1147:     push ax
1148:     mov ah,[BYTE Fillcolor]
1149: ; Mit Füllfarbe vergleichen
1150:     mov al,02h
1151:     out dx,ax
1152:     mov ch,[es:di]
1153: ; Farbbyte holen und vergleichen
1154:     mov ah,[BYTE tcolor]
1155: ; Mit Randfarbe vergleichen
1156:     out dx,ax
1157:     or ch,[es:di]
1158:     pop ax
1159:     mov dh,ch
1160: ; Ergebnis retten
1161:     cmp cl,128
1162: ; Anfang eines Grafik-Byte ?
1163:     jne NotRightLoop1
1164:     or dh,dh
1165: ; Ja,prüfen,ob Grafik-Byte <> 0
1166:     je NotRightLoop1
1167: ; Nein,weiter
1168:     push dx
1169:     mov dx,bp
1170: ; Ja,Endpunkt darin enthalten ?
1171:     sub dx,ax
1172:     cmp dx,8
1173:     pop dx
1174:     jb NotRightLoop1
1175:     add ax,8
1176: ; Ja,nächstes Byte
1177:     inc di
1178:     jmp NotRightLoop
1179: NotRightLoop1:
1180:     mov ch,dh
1181: ; Vergleichsergebnis holen
1182:     and ch,cl
1183: ; Bit-Maske isolieren
1184:     je FoundNotRight
1185: ; Bit=0,dann NICHT gesetzter Punkt
1186: ; gefunden !
1187:     cmp ax,bp
1188: ; Endpunkt erreicht,dann Ende
1189:     je FoundNotRight
1190:     inc ax
1191: ; Spalte erhöhen
1192:     shr cl,1
1193: ; Grafik-Byte "aufgebraucht" ?
1194:     jnc NotRightLoop1
1195:     inc di
1196: ; Grafik-Byte erhöhen
1197:     mov cl,128
1198: ; Bit-Maske neu setzen
1199:     jmp NotRightLoop
1200: ; und auslesen
1201: FoundNotRight:
1202:     push ax
1203:     mov dx,03CEh
1204:     mov ax,0205h
1205: ; Read-Modus 0 und Write-Modus 2
1206:     out dx,ax
1207: ; setzen
1208:     pop ax
1209:     pop dx
1210:     pop cx
1211:     pop bp
1212:     pop bx
1213:     pop di
1214:     ret
1215: ENDP
1216: ;
1217: ; TestPunkt:
1218: ; Stellt fest,ob Punkt in Rand-
1219: ; oder Füllfarbe gesetzt ist

```

```

1220: ;
1221: ; Eingabe: AX=X-Koordinate (0-639)
1222: ; des Punktes
1223: ; BX=Y-Koordinate (0-479) des Punktes
1224: ; Ausgabe: Zero-Flag gesetzt=Punkt
1225: ; ist in Rand- oder Füllfarbe
1226: ; gesetzt
1227: PROC TestPunkt NEAR
1228:     push di
1229:     push cx
1230:     push dx
1231:     push ax
1232: ; X-Koordinate retten
1233:     call CALCXy
1234: ; Adresse des Startpunktes im Video-
1235: ; RAM suchen
1236:     mov cl,dl
1237: ; Bit-Maske retten
1238:     mov dx,03CEh
1239:     mov ax,0A05h
1240: ; Read-Modus 1 setzen
1241:     out dx,ax
1242:     mov ah,[BYTE Fillcolor]
1243: ; Mit Füllfarbe vergleichen
1244:     mov al,02h
1245:     out dx,ax
1246:     mov ch,[es:di]
1247: ; Farbbyte holen und vergleichen
1248:     mov ah,[BYTE tcolor]
1249: ; Mit Randfarbe vergleichen
1250:     out dx,ax
1251:     or ch,[es:di]
1252:     and ch,cl
1253: ; Bit-Maske isolieren
1254:     mov dx,03CEh
1255:     mov ax,0205h
1256: ; Read-Modus 0 und Write-Modus 2
1257: ; setzen
1258:     out dx,ax
1259:     pop ax
1260:     pop dx
1261:     pop cx
1262:     pop di
1263:     ret
1264: ENDP
1265: PROC AsmFuell
1266:     mov dx,ax
1267: ; aktuelle Spalte
1268:     mov bp,bx
1269: ; aktuelle Zeile
1270:     call TestPunkt
1271: ; Startpunkt gesetzt ?
1272:     jz L1
1273:     jmp L1
1274: ; Ja,dann Ende
1275: L1:
1276:     dec bp
1277: ; Zeile vermindern
1278:     mov ax,dx
1279:     mov bx,bp
1280:     push dx
1281:     xor dx,dx
1282: ; Bis z. ob. Bildschirmrand(Y=0)suchen
1283:     call SearchTop
1284: ; Nach oben wandern,bis Punkt gesetzt
1285:     pop dx
1286:     mov bp,bx
1287: ; Gefundene Zeile setzen
1288:     inc bp
1289: ; Einen Schritt nach rechts und
1290: ; nach unten machen
1291:     inc dx
1292:     mov ax,dx
1293:     mov bx,bp
1294: ; Dieser Punkt auch gesetzt ?
1295:     call TestPunkt
1296:     jz L1
1297: ; Nein,weiter nach oben wandern
1298:     mov ax,dx
1299: ; Ja,rechte,obere Ecke der Figur
1300:     dec ax
1301: ; erreicht
1302:     mov [speicher21],ax
1303: ; Diese Koordinaten merken
1304:     mov [speicher22],bp
1305: L2:
1306:     dec dx
1307: ; Solange nach links wandern,bis
1308:     mov ax,dx
1309: ; gesetzter Punkt erreicht wird
1310:     mov bx,bp
1311:     xor dx,dx
1312: ; Bis z. link. Rand (X=0) suchen
1313:     call SearchLeft
1314:     mov dx,ax
1315:     inc dx
1316: ; Koordinaten merken
1317:     mov ax,dx
1318:     mov si,ax
1319:     mov bx,bp
1320:     mov di,bx
1321: ; Linie von diesem zum rechten Punkt
1322:     mov cx,[speicher21]
1323: ; zeichnen
1324:     call ZieheLinie

```

```

1325:     dec bp
1326: ;Eine Zeile nach oben wandern
1327:     mov dx,si
1328: ;Begrenzung der Figur)
1329: L3:
1330:     cmp dx,[speicher21]
1331:     ja L5
1332:     mov ax,dx
1333:     mov bx,bp
1334:     mov dx,[speicher21]
1335:     call SearchNotRight
1336: ;Punkt auf Begrenzung suchen,der
1337: ; nicht gesetzt ist
1338:     mov dx,ax
1339:     cmp dx,[speicher21]
1340: ;Rechte Begrenzung erreicht,dann
1341: ;keine Lücke mehr nach oben
1342:     je L5
1343: La40:
1344:     push si
1345:     push di
1346:     mov ax,[speicher21]
1347:     push ax
1348:     mov ax,[speicher22]
1349:     push ax
1350:     mov ax,[speicher31]
1351:     push ax
1352:     mov ax,[speicher32]
1353:     push ax
1354:     mov ax,[speicher41]
1355:     push ax
1356:     mov ax,[speicher42]
1357:     push ax
1358:     mov ax,dx
1359:     push ax
1360:     mov bx,bp
1361:     push bx
1362:     call AsmFuell
1363: ;Fill-Routine aufrufen
1364:     pop bp
1365:     pop dx
1366:     pop ax
1367:     mov [speicher42],ax
1368:     pop ax
1369: ;Nach Füllung der Lücke Variablen
1370: ; wieder vom Stack holen
1371:     mov [speicher41],ax
1372:     pop ax
1373:     mov [speicher32],ax
1374:     pop ax
1375:     mov [speicher31],ax
1376:     pop ax
1377:     mov [speicher22],ax
1378:     pop ax
1379:     mov [speicher21],ax
1380:     pop di
1381:     pop si
1382:     jmp L3
1383: L5:
1384:     mov bx,di
1385: ;Linken Ausgangspunkt der Zeile
1386:     inc bx
1387: ; (X und Y) für weiteres Vorgehen
1388:     mov bp,bx
1389:     mov ax,si
1390:     mov dx,ax
1391:     mov [speicher32],bx
1392: ; Zeile merken
1393:     mov [speicher42],bx
1394:     call TestPunkt
1395: ; Punkt gesetzt ?
1396:     jz La4
1397: ; Nein
1398: L6:
1399:     inc dx
1400: ; Ja : Spalte solange erhöhen,bis
1401: ; nicht gesetzter Punkt gefunden
1402:     mov ax,dx
1403:     mov bx,bp
1404: ; oder Spalte > als rechte Begrenzung
1405:     mov dx,[speicher21]
1406:     inc dx
1407:     call SearchNotRight
1408: ; Punkt auf Begrenzung suchen,der
1409: ; nicht gesetzt ist
1410:     mov dx,ax
1411:     jmp La5
1412: La4:
1413:     dec dx
1414: ; Wenn Ursprungspunkt nicht gesetzt,
1415: ; Spalte solange vermindern,bis
1416: ; gesetzter Punkt erreicht wird
1417:     mov ax,dx
1418:     mov bx,bp
1419:     xor dx,dx

```

Das Demoprogramm »fill.asm« veranschaulicht ein optimiertes Prinzip zum Füllen von Flächen.


```

1420: ;Bis zum 1. Bildschirmrand suchen
1421:   call SearchLeft
1422:   mov dx,ax
1423:   inc dx
1424: La5:   mov [speicher31],dx
1425: ;Spalte merken als linker Eckpunkt
1426: ;
1427:   mov ax,[speicher21]
1428:   cmp dx,ax
1429: ;Falls größer als rechter Eckpunkt
1430: ; der darüberliegenden Linie, rechte,
1431: ; untere Ecke
1432: ;der Figur gefunden ,d.h. Fill-Ende
1433:   jna L8
1434:   jmp Lal
1435: L8:
1436:   inc dx
1437: ;wenn nicht größer: Rechten Eckpunkt
1438:
1439: ; der Linie
1440: ;suchen, indem Spalte erhöht wird
1441:   mov ax, dx
1442:   mov bx, bp
1443:   mov dx, 639
1444: ;Bis zum rechten Rand (X=639) suchen
1445:   call SearchRight
1446:   mov dx, ax
1447:   dec dx
1448:   mov ax, [speicher31]
1449: ;Linie zeichnen
1450:   mov bx, [speicher32]
1451:   mov cx, dx
1452:   call ZieheLinie
1453:   mov [speicher41], dx
1454: ;Spalte des rechten Eckpunktes der
1455: ; Linie merken
1456:   mov ax, si
1457: ;Beginnt Linie weiter links als die
1458: ;darüberliegende ???
1459:   dec ax
1460:   cmp ax, [speicher31]
1461:   ja La22
1462: ;Ja, Sonderfall 1
1463:   jmp La8
1464: La22:
1465:   mov dx, si
1466: ;Sonderfall 1: Figurzipfel auf linker
1467: ;Seite muß gefüllt werden
1468:   mov bp, di
1469: L9:
1470:   dec dx
1471:   mov ax, dx
1472: ;Freien Punkt suchen
1473:   mov bx, bp
1474:   mov dx, [speicher31]
1475:   call SearchNotLeft
1476:   mov dx, ax
1477:   cmp dx, [speicher31]
1478:   jz La8
1479:   push si
1480:   push di
1481:   mov ax, [speicher21]
1482:   push ax
1483:   mov ax, [speicher22]
1484:   push ax
1485:   mov ax, [speicher31]
1486:   push ax
1487:   mov ax, [speicher32]
1488:   push ax
1489:   mov ax, [speicher41]
1490:   push ax
1491:   mov ax, [speicher42]
1492:   push ax
1493:   mov ax, dx
1494:   push ax
1495:   mov bx, bp
1496:   push bx
1497:   call AsmFuell
1498: ;Fill-Routine aufrufen
1499:   pop bp
1500:   pop dx
1501:   pop ax
1502:   mov [speicher42], ax
1503:   pop ax
1504: ;Nach Füllung der Lücke Variablen wie
1505: ;der vom Stack holen
1506:   mov [speicher41], ax
1507:   pop ax
1508:   mov [speicher32], ax
1509:   pop ax
1510:   mov [speicher31], ax
1511:   pop ax
1512:   mov [speicher22], ax
1513:   pop ax
1514:   mov [speicher21], ax
1515:   pop di
1516:   pop si
1517: La81:
1518: ;Nur bis zum linken Ausgangspunkt
1519: ;der Linie prüfen
1520:   jmp L9
1521: La8:
1522:   mov ax, [speicher41]
1523: ;Endet Linie weiter rechts als die

```

```

1524: ;darüberliegende ???
1525:   inc ax
1526:   cmp [speicher21], ax
1527:   ja La21
1528: ;Ja, Sonderfall 2
1529:   jmp La9
1530: La21:
1531:   mov dx, [speicher41]
1532: ;Sonderfall 2: Figurzipfel auf rechter
1533: ;Seite muß gefüllt werden
1534:   mov bp, [speicher42]
1535: L10:
1536:   inc dx
1537:   mov ax, dx
1538: ;Freier Punkt gefunden, dann füllen
1539:   mov bx, bp
1540:   mov dx, [speicher21]
1541:   call SearchNotRight
1542:   mov dx, ax
1543:   cmp dx, [speicher21]
1544:   jz La9
1545:   push si
1546:   push di
1547:   mov ax, [speicher21]
1548:   push ax
1549:   mov ax, [speicher22]
1550:   push ax
1551:   mov ax, [speicher31]
1552:   push ax
1553:   mov ax, [speicher32]
1554:   push ax
1555:   mov ax, [speicher41]
1556:   push ax
1557:   mov ax, [speicher42]
1558:   push ax
1559:   mov ax, dx
1560:   push ax
1561:   mov bx, bp
1562:   push bx
1563:   call AsmFuell
1564: ;Fill-Routine aufrufen
1565:   pop bp
1566:   pop dx
1567:   pop ax
1568:   mov [speicher42], ax
1569:   pop ax
1570: ;Nach Füllung der Lücke Variablen wie
1571: ;der vom Stack holen
1572:   mov [speicher41], ax
1573:   pop ax
1574:   mov [speicher32], ax
1575:   pop ax
1576:   mov [speicher31], ax
1577:   pop ax
1578:   mov [speicher22], ax
1579:   pop ax
1580:   mov [speicher21], ax
1581:   pop di
1582:   pop si
1583: La91:
1584: ;Nur bis zum rechten Endpunkt der
1585: ;Linie prüfen
1586:   jmp L10
1587: La9:
1588:   mov ax, [speicher41]
1589: ;Endet Linie weiter links als die
1590: ;darüberliegende ???
1591:   dec ax
1592:   cmp ax, [speicher21]
1593:   ja La20
1594: ;Ja, Sonderfall 3
1595:   jmp La10
1596: La20:
1597:   mov dx, [speicher21]
1598: ;Sonderfall 3: Figurzipfel an unterer
1599: ;Seite muß gefüllt werden
1600:   mov bp, [speicher22]
1601: L11:
1602:   inc dx
1603:   mov ax, dx
1604: ;Freier Punkt gefunden, dann füllen
1605:   mov bx, bp
1606:   mov dx, [speicher41]
1607:   call SearchNotRight
1608:   mov dx, ax
1609:   cmp dx, [speicher41]
1610:   jz La10
1611:   push si
1612:   push di
1613:   mov ax, [speicher21]
1614:   push ax
1615:   mov ax, [speicher22]
1616:   push ax
1617:   mov ax, [speicher31]
1618:   push ax
1619:   mov ax, [speicher32]
1620:   push ax
1621:   mov ax, [speicher41]
1622:   push ax
1623:   mov ax, [speicher42]
1624:   push ax
1625:   mov ax, dx
1626:   push ax
1627:   mov bx, bp
1628:   push bx

```

```

1629:   call AsmFuell
1630: ;Fill-Routine aufrufen
1631:   pop bp
1632:   pop dx
1633:   pop ax
1634:   mov [speicher42], ax
1635:   pop ax
1636: ;Nach Füllung der Lücke Variablen wie
1637: ; der vom Stack holen
1638:   mov [speicher41], ax
1639:   pop ax
1640:   mov [speicher32], ax
1641:   pop ax
1642:   mov [speicher31], ax
1643:   pop ax
1644:   mov [speicher22], ax
1645:   pop ax
1646:   mov [speicher21], ax
1647:   pop di
1648:   pop si
1649: La101:
1650: ;Nur bis zum linken Ausgangspunkt der
1651: ;Linie prüfen
1652:   jmp L11
1653: La10:
1654:   mov si, [speicher31]
1655: ;Darüberliegende Linie=akt. Linie
1656:   mov di, [speicher32]
1657:   mov ax, [speicher41]
1658:   mov [speicher21], ax
1659:   mov ax, [speicher42]
1660:   mov [speicher22], ax
1661:   jmp L5
1662: ;nächste Linie bearbeiten
1663: Lal:
1664:   ret
1665: ENDP
1666: ;
1667: ; ASMFILL:
1668: ; Füllt eine beliebige Fläche aus
1669: ;
1670: ; Eingabe: AX=X-Koordinate
1671: ; Füllpunkt (0-639)
1672: ; BX=Y-Koordinate Füllpunkt (0-479)
1673: ;
1674: ; CX=Füllfarbe (Low-Byte)
1675: ; DX=Randfarbe (Low-Byte)
1676: ; Ausgabe: keine
1677: PROC AsmFill
1678:   mov fillcolor, cx
1679: ;Füllfarbe merken
1680:   mov tcolor, dx
1681: ;Randfarbe merken
1682:   call AsmFuell
1683:   ret
1684: ENDP
1685: GfxStart:
1686:   mov ax, @data
1687:   mov ds, ax
1688:   call AsmInitGraphic
1689: ; Grafikmodus einschalten
1690:   mov cx, 23
1691: ;Umrandung zum Flächenfüllen zeichnen
1692:   mov di, Offset linetab
1693: _loop4:
1694:   push cx
1695:   mov ax, ds:[di]
1696:   mov bx, ds:[di+2]
1697:   mov cx, ds:[di+4]
1698:   mov dx, ds:[di+6]
1699:   push di
1700:   mov di, 13
1701:   call AsmLine
1702:   pop di
1703:   add di, 8
1704:   pop cx
1705:   loop _loop4
1706:   xor ah, ah
1707:   int 16h
1708:   mov ax, 60
1709: ; Fläche füllen
1710:   mov bx, 100
1711:   mov cx, 15
1712:   mov dx, 13
1713:   call AsmFill
1714:   xor ah, ah
1715:   int 16h
1716:   call AsmInitText
1717: ; Textmodus wieder einschalten
1718:   mov ax, 4C00h
1719: ; Zurück zu DOS
1720:   int 21h
1721: END GfxStart

```

Das Demoprogramm »fill.asm« veranschaulicht ein optimiertes Prinzip zum Füllen von Flächen.

Flexibilität und Geschwindigkeit kontra VGA-BIOS

Wer auf dem Bildschirm Grafiken produziert, möchte diese im allgemeinen auch beschriften. Selbst bei dieser Aufgabe versagen die VGA-BIOS-Funktionen, so daß der Grafikprogrammierer wiederum selbst Hand anlegen muß.

Der einfachste Weg, einen Text auf dem Grafikbildschirm auszugeben, besteht im Aufruf der Funktion 13h des VGA-BIOS-Interrupts 10h, wie in Listing 1 dargestellt. Die Problematik der Interrupt-Routine ist in erster Linie in ihrer mangelnden Flexibilität zu sehen. Zwar lassen sich durch weitere VGA-BIOS-Aufrufe noch drei Zeichensätze auswählen (8x8, 8x14 oder 8x16 Bildpunkte pro Zeichen). Eine Ausgabe von vergrößerten Buchstaben, zum Beispiel für Überschriften, oder eine Kippung der Schrift, beispielsweise zur Beschriftung von vertikalen Achsen bei Chart-Grafiken, ist dagegen genausowenig unterstützt wie Kursivschrift oder das Unterstreichen der Zeichen. Das Paradoxe dabei ist, daß den Entwicklern die Programmierung dieser Leistungsmerkmale gerade im Grafikmodus keine Schwierigkeiten bereitet hätte.

Bevor man sich jedoch dem Schreiben einer eigenen Routine widmet, muß man sich zunächst im klaren darüber sein, wie ein Zeichen überhaupt aufgebaut ist. Jede moderne Grafikkarte inklusive EGA und VGA besitzt ein sogenanntes Charakter-ROM. Hierbei handelt es sich um einen Baustein, in dem die Grafikdaten jedes Zeichens in Byteform nach einem bestimmten, von Grafikkarte zu Grafikkarte verschiedenen Muster abgelegt sind. Will der Controller ein Zeichen im Textmodus auf dem Bildschirm ausgeben, sieht er gewissermaßen im ROM nach und liest die entsprechenden Daten.

Das gleiche Vorgehen müssen wir im Grafikmodus realisieren. Um an die Startadresse der Zeichensatztafel zu gelangen, kann man sich der Funktion 11h, Unterfunktion 30h, des VGA-BIOS-Interrupts 10h bedienen. Der Interrupt-Routine wird im Register BH ein Code für den ge-

wünschten Zeichensatz übergeben (2 für 8x14-, 3 für 8x8- und 6 für 8x16-Zeichensatz). Die Adresse der Zeichensatztafel des ausgewählten Zeichensatzes findet sich daraufhin im Registerpaar ES:BP. Die Interrupt-Routine verändert die Registerinhalte des Grafik-Controllers, so daß man diese Werte anschließend neu laden muß, um das Plotmodul (siehe erster Kursteil) aufrufen zu können.

Bit	76543210
Zeile 0	..*.*.*.
Zeile 1	.****.*.
Zeile 2	**..*.*.
Zeile 3	*****.
Zeile 4	**..*.*.
Zeile 5	**..*.*.
Zeile 6	**..*.*.
Zeile 7	

Bild 1. Aufbau des Zeichens A im 8x8-Zeichensatz: Jedes gesetzte Bild steht für einen zu zeichnenden Punkt.

Da jeder Grafikpunkt in der Zeichensatztafel ein Bit belegt, sind für die Speicherung eines Zeichens zum Beispiel im 8x8-Zeichensatz

$8 \times 8 \text{ Bit} = 8 \text{ Byte}$ erforderlich. Die Reihenfolge der Daten entspricht dabei derjenigen der ASCII-Codes, so daß beispielsweise die Anfangsadresse der Grafikdaten des Zeichens A (ASCII-Code 65) nach der Formel

$\text{Offset} = \text{Tabellenstart} + 8 \times 65$ berechenbar ist. Beim 8x8-Zeichensatz enthält ein Byte immer eine komplette Zeile eines Zeichens, so daß das zweite Byte für die zweite Zeile zuständig ist und so weiter. Das höchstwertige Bit eines Bytes repräsentiert dabei den am weitesten links liegenden Grafikpunkt, das niederwertigste

den am weitesten rechts liegenden. Zur Veranschaulichung gehen die acht Byte des Zeichens A im 8x8-Zeichensatz aus Bild 1 hervor. Ein Zeichen im 8x14-Zeichensatz belegt entsprechend 14 und ein Zeichen im 8x16-Zeichensatz 16 Byte, da es 14 beziehungsweise 16 Zeilen aufweist.

Die unterste Zeile jedes Zeichens bleibt immer frei, um ein Unterstreichen zu erlauben, analog die beiden rechten Spalten, um den notwendigen Abstand zum nächsten Zeichen zu erhalten. Wenn man sich das Beispiel aus Bild 1 ansieht, kann man leicht die Inhalte der acht Byte in der Zeichensatztafel ermitteln: In der nullten Zeile sind die Bits 4 ($2^4=16$) und 5 ($2^5=32$) gesetzt, das heißt, das erste Byte weist den Wert

$$32 + 16 = 48$$

auf. Das zweite, vierte, fünfte und sechste Byte ist identisch aufgebaut und enthält die Werte

$$128 + 64 + 8 + 4 = 204$$

Mit der gleichen Technik können Sie die Werte des ersten und dritten Bytes ermitteln, während das siebte Byte mit Inhalt 0 kein zu setzendes Zeichenmuster enthält.

Die Textroutine muß also abhängig davon, ob ein Bit ein- oder ausgeschaltet ist, den entsprechenden Grafikpunkt setzen beziehungsweise löschen. Wenn man einen

Zeile 0	*
Zeile 1	*
Zeile 2	*
Zeile 3	*
Zeile 4	*
Zeile 5	*
Zeile 6	*
Zeile 7	*

Bild 2. Vergrößerung eines Zeichens in x-Richtung um den Faktor 2 durch Verdoppelung jedes Grafikpunktes, jedoch ohne Kantenglättung

Schrifttyp	Kippung	zl	zu	sl	su	zch	spt
Normal	0 Grad	0	1	1	0	8	0
Linkskursiv	0 Grad	-1	1	1	0	8	0
Rechtskursiv	0 Grad	1	1	1	0	8	0
Normal	90 Grad	-1	0	0	1	0	8
Linkskursiv	90 Grad	-1	-1	0	1	0	8
Rechtskursiv	90 Grad	-1	1	0	1	0	8
Normal	180 Grad	0	-1	-1	0	-8	0
Linkskursiv	180 Grad	-1	-1	-1	0	-8	0
Rechtskursiv	180 Grad	1	-1	-1	0	-8	0
Normal	270 Grad	1	0	0	-1	0	-8
Linkskursiv	270 Grad	1	-1	0	-1	0	-8
Rechtskursiv	270 Grad	1	1	0	-1	0	-8

Schriftvariationen durch unterschiedliche Variablenkombinationen

Vergrößerungsfaktor größer als 1 einführt, wird jeder Punkt in x-Richtung so oft gesetzt, wie dies der Faktor festlegt (Bild 2). Bei einer Vergrößerung in y-Richtung sind dementsprechend alle Zeilen gemäß dem Skalierungsmaß wiederholt auszugeben.

Üblicherweise gelangt man zum nächsten Punkt innerhalb einer Zeile dadurch, daß man einen Punkt in x-Richtung fortschreibt und die y-Koordinate unverändert läßt. Um ein Kippen des Textes zu gestatten, läßt sich dieser Weg aber auch flexibel handhaben, zum Beispiel würde durch eine Bewegung um einen Punkt in y-Richtung und das Konstanthalten der x-Koordinate eine Kippung um 90 Grad erreicht.

Allgemein gesprochen benötigt eine Textausgaberroutine also zwei Variablen, die festlegen, um wie viele Pixel in x- beziehungsweise y-Richtung fortgeschritten werden soll, um den nächsten Punkt innerhalb einer Zeile zu erreichen. In unserer Routine »text.asm« (Listing 2) sind die Variablen *su* (Bewegung nach oben/unten) und *sl* (Bewegung nach links/rechts) dafür zuständig.

Ist eine Zeile bearbeitet, folgt das Lesen des nächsten Bytes und die Positionierung an den Anfang der folgenden Zeile. Auch hierfür stehen zwei Variablen bereit: *zu* (Bewegung nach oben/unten) und *zl* (Bewegung nach links/rechts). Sie teilen dem Modul mit, um wie viele Punkte in x- beziehungsweise y-Richtung der neue Zeilenanfang relativ zum Zeilenanfang der vorhergehenden Zeile zu finden ist. Bei aufrecht stehendem Text muß man natürlich einen Punkt in y-Richtung weiterpositionieren, während die x-Koordinate konstant bleibt (Differenz: 0 Punkte). Für gekippten Text gelten entsprechend andere Werte.

An dieser Stelle fällt auf, daß ein solcher Algorithmus praktisch als kostenlose Zugabe die Fähigkeit zur Erzeugung von Kursivschrift anbietet. Wenn nämlich der neue Zeilenanfang es so festlegt, bewegt sich die Programmsequenz auch um einen Punkt in x- und y-Richtung. Insgesamt stehen damit ein Dutzend sinnvolle Schriftvariationen zur Verfügung. Die Tabelle bietet eine Übersicht über den jeweiligen

Schrifttyp zusammen mit den erforderlichen Daten für *su*, *sl*, *zu* und *zl*. Die Variablen *zch* und *spt* enthalten dabei die Abstände in x- beziehungsweise y-Richtung vom Ausgangspunkt eines Zeichens (die linke, obere Ecke) zur Startposition des nächsten Zeichens.

Nachdem ein Zeichenmuster auf den Bildschirm komplett ausgegeben wurde, folgen gegebenenfalls weitere Werte. Die Textroutine erhält einen Pointer (Segment/Offset) auf den auszugebenden String, wobei die Länge der Zeichenkette aus dem ersten Code hervorgeht. »text.asm« baut nacheinander vier Grafikbildschirme mit unterschiedlichen Textvariationen auf.

Dem Unterprogramm »AsmText« in Listing 2 (»text.asm«) wird unter anderem noch ein Parameter zur Festlegung des

Listing 1: biostext.asm

```

1: text db "Hallo Welt !"
2: mov es, ds
3: ;Segmentadresse des Ausgabetextes holen
4: mov bp, Offset Text
5: ;Offsetadresse des Ausgabetextes holen
6: mov cx, zeichen
7: ;Anzahl Zeichen des Ausgabetextes
8: mov ax, 1300h
9: mov bl, farbe
10: ;Farbwert 0-15
11: mov dh, zeile
12: ;Bildschirmzeile d. Ausgabestartpunktes
13: mov dl, spalte
14: ;Bildschirmspalte d. Ausgabestartpunktes
15: xor bh, bh
16: int 10h
17: ;VGA-BIOS-Interrupt aufrufen

```

Routine zur Textausgabe über den VGA-BIOS-Interrupt 10h

Listing 2: text.asm

```

1: ;Programm: text.asm
2: ;Funktion: Demo-Programm für den
3: ; Grafikbefehl
4: ; TEXT (Text schreiben)
5: ;Sprache : Assembler
6: ;Autor : Frank Riemenschneider
7: ;(c) 1993 DMV Daten- und Medienverlag
8: ;
9: IDEAL
10: ; IDEAL-Modus einschalten
11: MODEL SMALL
12: ; Speichermodell SMALL wählen
13: STACK
14: db 1024 dup (?)
15: DASEG
16: demotext db 14, 'Hallo Welt !!!'
17: zaehl dw 0
18: kipp dw 0

```

Überschreibmodus in den oberen acht Bit des Registers DI übergeben. Wenn man den Wert 0 lädt, beispielsweise mit der Anweisung

```
and di, 00FFh
```

so werden die nicht gesetzten Punkte im Punktraster eines Zeichens auch nicht bearbeitet, das heißt, nicht mit der Hintergrundfarbe gefüllt. Initialisieren Sie hingegen die oberen acht Bit des Registers DI mit dem Wert 1, zum Beispiel mit der Anweisung

```
or di, 0100
```

so setzt das Modul diese Pixel in der Hintergrundfarbe und überschreibt damit die Grafikinformatoren, die sich zuvor an dieser Stelle befunden haben.

(Frank Riemenschneider/bj)

```

19: kiptab dw 30,30,50,100,600,30,-100,3
20: 0,600,450,-50,-100,30,450,100,-30
21: xofs dw 0
22: yofs dw 0
23: CODESEG
24: ;
25: ; ASMINITGRAPHIC:
26: ; Schaltet die VGA-Karte in den
27: ; Grafikmodus und löscht Bildschirm
28: ;
29: ; Eingabe: keine
30: ; Ausgabe: keine
31: PROC AsmInitGraphic
32: push ax
33: ; Für alle Fälle alle Register retten
34: push bx
35: push cx
36: push dx
37: push ds
38: push di
39: xor ah, ah
40: ; Funktionsnummer laden
41: mov al, 12h
42: ; Grafikmodus 640 x 480 Punkte in 16
43: int 10h
44: ; Farben setzen
45: mov dx, 3CEh
46: ; Port-Adresse des VGA-Grafik-
47: ; controllers laden
48: mov ax, 0205h
49: ; Read-Modus 0 und Write-Modus 2
50: ; im Modusregister setzen
51: out dx, ax
52: mov ax, 0003h
53: ; Verknüpfungsmodus = Ersetzen im
54: ; Function-Select-Register setzen
55: out dx, ax
56: pop di
57: ; Alle Registerinhalte wieder
58: ; herstellen
59: pop ds
60: pop dx
61: pop cx
62: pop bx
63: pop ax
64: ret
65: ; Back home
66: ENDP
67: ;
68: ; ASMINITTEXT:
69: ; Schaltet die VGA-Karte in den Text-
70: ; modus
71: ; Eingabe: keine
72: ; Ausgabe: keine
73: PROC AsmInitText
74: push ax
75: ; Für alle Fälle alle Register retten
76: push bx
77: push cx
78: push dx
79: push ds
80: push di
81: mov dx, 3CEh
82: ; Port-Adresse des VGA-Grafik-
83: ; controllers laden
84: mov ax, 0005h
85: ; Mode- und Function-Select-Register
86: ; mit ihren
87: ; Standardwerten laden
88: out dx, ax
89: mov ax, 0003h
90: out dx, ax
91: xor ah, ah
92: ; Funktionsnummer laden
93: mov al, 3
94: ; Textmodus 80 x 25 Zeichen in 16

```

»text.asm« zeigt die Verwendung von Text in Grafikapplikationen.


```

95: ; Farben setzen
96: int 10h
97: pop di
98: ; Alle Registerinhalte wieder
99: ; herstellen
100: pop ds
101: pop dx
102: pop cx
103: pop bx
104: pop ax
105: ret
106: ; Back home
107: ENDP
108: ;
109: ; CALCXY:
110: ; Berechnet Segment und Offset des
111: ; Byte im Video-RAM, sowie die Bit-
112: ; Maske für das passende Bit bei
113: ; gegebener X- und Y-Koordinate
114: ;
115: ; Eingabe: AX = X-Koordinate (0-639)
116: ; BX = Y-Koordinate (0-479)
117: ; Ausgabe: ES = Segmentadresse des
118: ; Bytes im Video-RAM
119: ; DI = Offsetadresse des Bytes im
120: ; Video-RAM
121: ; DL = Bit-Maske des Bytes im
122: ; Video-RAM
123: PROC CalcXY
124: push ax
125: push bx
126: push cx
127: mov cx, ax
128: ; Koordinaten nochmal für spätere
129: ; Verwendung in der Berechnung retten
130: mov ax, 80
131: ; Pro Zeile 80 Bytes
132: mul bx
133: mov dx, 0A000h
134: ; Offset VGA-Karte laden
135: mov es, dx
136: ; Segmentadresse laden
137: mov dx, ax
138: mov ax, cx
139: shr ax, 1
140: shr ax, 1
141: shr ax, 1
142: ; x/8
143: add dx, ax
144: ; Spaltenoffset addieren
145: mov di, dx
146: ; DI mit Offset laden
147: ;
148: ; Berechnung der Bit-Maske
149: and cl, 7
150: ; x MOD 8
151: mov dl, 80h
152: ; Maske für Bit 0
153: shr di, cl
154: ; Maske für gew. Bit durch Rotation
155: ; der ursprünglichen Maske erzeugen
156: pop cx
157: pop bx
158: pop ax
159: ret
160: ; Back home
161: ENDP CalcXY
162: ;
163: ; VGAPLOT:
164: ; Setzt einen Bildpunkt im Video-RAM
165: ; der VGA-Karte
166: ;
167: ; Eingabe: AX=X-Koordinate des Pixels
168: ; BX=Y-Koordinate des Pixels
169: ; DI=Farbe Low-Byte
170: ; Ausgabe: keine
171: PROC AsmPlot
172: push ax
173: push bx
174: push cx
175: push dx
176: push si
177: push di
178: mov cx, di
179: ; Farbe retten
180: call CalcXY
181: ; Offset- und Segmentadresse, sowie die
182: ; Bit-Maske aus den X- und
183: ; Y-Koordinaten berechnen
184: mov ah, dl
185: ; Bit-Maske
186: mov dx, 3CEh
187: ; Port-Adresse des VGA-Grafik-
188: ; Controllers laden
189: mov al, 08h
190: ; Bit-Maske im Mask-Register setzen
191: out dx, ax
192: mov ch, [es:di]
193: ; Latch-Register laden
194: mov [es:di], cl
195: ; Farbe setzen
196: pop di
197: pop si
198: pop dx
199: pop cx
200: pop bx
201: pop ax
202: ret
203: ; Back home
204: ENDP
205: ;
206: ; ASMTEXT:

```

```

207: ; Schreibt einen Text in den Grafik-
208: ; Bildschirm
209: ;
210: ; Eingabe: AX=X-Koordinate (0-639)
211: ; BX=Y-Koordinate (0-479)
212: ; CL=Farbe (0-15)
213: ; CH=Zeichenhöhe (8, 14 oder 16)
214: ; DL=Vergrößerungsfaktor in
215: ; X-Richtung (1-255)
216: ; DH=Vergrößerungsfaktor in
217: ; Y-Richtung (1-255)
218: ; DI=Code (0-11, Low-Byte,
219: ; Überschreiben (1=Ja, 0=Nein)
220: ;
221: ; High-Byte)
222: ; ES=Segment des Textes
223: ; SI=Offsetadresse des Textes
224: ; Ausgabe: keine
225: DATASEG
226: CharSetOfs dw 0
227: TextSeg dw (?)
228: Farbe db 0
229: Z_Hoehe db 0
230: ueber db 0
231: vartab:
232: zl dw 0
233: zu dw 0
234: sl dw 0
235: su dw 0
236: zch dw 0
237: spt dw 0
238: xver dw 0
239: ; Vergrößerungsfaktor in X-Richtung
240: yver dw 0
241: ; Vergrößerungsfaktor in Y-Richtung
242: codetab dw 0,1,1,0,8,0
243: ;Normal, 0 Grad Kippung
244: dw -1,1,1,0,8,0
245: ;Linkskursiv, 0 Grad Kippung
246: dw 1,1,1,0,8,0
247: ;Rechtskursiv, 0 Grad Kippung
248: dw -1,0,0,1,0,8
249: ;Normal, 90 Grad Kippung
250: dw -1,-1,0,1,0,8
251: ;Linkskursiv, 90 Grad Kippung
252: dw -1,1,0,1,0,8
253: ;Rechtskursiv, 90 Grad Kippung
254: dw 0,-1,-1,0,-8,0
255: ;Normal, 180 Grad Kippung
256: dw -1,-1,-1,0,-8,0
257: ;Linkskursiv, 180 Grad Kippung
258: dw 1,-1,-1,0,-8,0
259: ;Rechtskursiv, 180 Grad Kippung
260: dw 1,0,0,-1,0,-8
261: ;Normal, 270 Grad Kippung
262: dw 1,-1,0,-1,0,-8
263: ;Linkskursiv, 270 Grad Kippung
264: dw 1,1,0,-1,0,-8
265: ;Rechtskursiv, 270 Grad Kippung
266: CODESEG
267: PROC AsmText
268: push ax
269: ; Register retten
270: push bx
271: push cx
272: push dx
273: push es
274: push si
275: push di
276: push ax
277: push cx
278: push es
279: push si
280: mov ax, di
281: mov ueber, ah
282: ;Überschreibmodus retten
283: mov si, offset codetab
284: codeloop:
285: or al, al
286: ;Aus Code Index in Tabelle berechnen
287: je endcode
288: add si, 12
289: dec al
290: jmp codeloop
291: endcode:
292: mov di, offset vartab
293: mov cx, 6
294: push ds
295: ;Tabelle in Variablen kopieren
296: pop es
297: rep movsw
298: mov [BYTE xver], dl
299: mov [BYTE yver], dh
300: mov al, [BYTE zch]
301: ;Vergrößerungsfaktoren setzen
302: imul dl
303: mov zch, ax
304: mov al, [BYTE spt]
305: imul dh
306: mov spt, ax
307: pop si
308: pop es
309: pop cx
310: pop ax
311: mov dx, es
312: mov TextSeg, dx
313: mov farbe, cl
314: mov Z_Hoehe, ch
315: ; Zeichenhöhe merken
316: cmp ch, 8
317: ; Code für VGA-BIOS setzen
318: jne no_8

```

```

319: mov ch, 3
320: jmp weiter
321: no_8:
322: cmp ch, 14
323: jne no_14
324: mov ch, 2
325: jmp weiter
326: no_14:
327: mov ch, 6
328: weiter:
329: push ax
330: push bx
331: push dx
332: mov ah, 11h
333: mov al, 30h
334: mov bh, ch
335: ;Adresse der Zeichensatztable holen
336: int 10h
337: mov CharSetOfs, bp
338: mov dx, 3CEh
339: ; Port-Adresse des VGA-Grafik-
340: ; Controllers laden
341: mov ax, 0205h
342: ; Read-Modus 0 und Write-Modus 2
343: ; im Modusregister setzen
344: out dx, ax
345: mov ax, 0003h
346: ; Verknüpfungsmodus=Ersetzen im
347: ; Function-Select-Register setzen
348: out dx, ax
349: pop dx
350: pop bx
351: pop ax
352: push ds
353: mov cx, [TextSeg]
354: ; Segment Text holen
355: mov ds, cx
356: mov cl, [si]
357: ; Länge des Textes holen
358: xor ch, ch
359: pop ds
360: inc si
361: ; Zeiger auf ersten Buchstaben
362: mov dh, ueber
363: ; Überschreibmodus holen
364: TextLoop1:
365: push ax
366: push bx
367: push cx
368: ; Zeichenzähler retten
369: push ds
370: ; Datensegment Pascal retten
371: mov cx, [TextSeg]
372: ; Segment Text holen
373: mov ds, cx
374: mov cl, [si]
375: ; ASCII-Code des Buchstabens holen
376: pop ds
377: push ax
378: mov al, Z_Hoehe
379: ; Zeichenhöhe laden
380: mul cl
381: ; Zeichenmatrix beginnt innerhalb der
382: ; Tabelle an der Adresse (ASCII-Code*
383: ; Z_Höhe)
384: ;
385: pop ax
386: mov di, CharSetOfs
387: ; Offset der Zeichentabelle laden
388: add di, cx
389: ; ES:DI auf erste Zeile der Matrix
390: xor ch, ch
391: mov cl, Z_Hoehe
392: ; Zeilenzähler laden
393: TextLoop2:
394: push cx
395: ; Zeilenzähler retten
396: mov cx, [WORD yver]
397: ; Vergrößerungsfaktor Y
398: TextLoop5:
399: mov dl, [es:di]
400: ; Zeile der Zeichenmatrix holen
401: push cx
402: push ax
403: ; Koordinaten merken
404: push bx
405: ; von Ausgangspunkt Spalte
406: push di
407: ; Zeiger auf Zeichentabelle retten
408: push es
409: mov cx, 8
410: ; Pixelzähler laden
411: TextLoop3:
412: push cx
413: ; Pixelzähler retten
414: xor ch, ch
415: mov cl, [BYTE farbe]
416: ; Annahme: Punkt gesetzt
417: shl dl, 1
418: ; Pixel ins Carry schieben
419: jc TextLab1
420: ;Punkt gesetzt -> in Vordergrundfarbe
421: ; setzen
422: mov cl, 0
423: ; Punkt in Hintergrundfarbe setzen
424: TextLab1:
425: mov di, cx

```

»text.asm« zeigt die Verwendung von Text in Grafikapplikationen.


```

426: ;Farbe setzen
427:   mov cx, [WORD xver]
428: ;Vergrößerungsfaktor X laden
429: TextLoop4:
430:   or di, di
431:   jne plot
432:   or dh, dh
433:   je noplot
434: plot:
435:   call AsmPlot
436: ; Punkt setzen
437: noplot:
438:   add bx, [WORD su]
439: ; Y-Koordinate weiterzählen
440:   add ax, [WORD sl]
441: ; X-Koordinate weiterzählen
442:   loop TextLoop4
443: ; Bei Vergrößerung nochmals plotten
444:   pop cx
445: ; Pixelzähler wieder laden
446:   loop TextLoop3
447: ; Weitere Punkte setzen
448:   pop es
449: ; Zeiger auf Zeichentabelle laden
450:   pop di
451:   pop bx
452: ; Y-Koordinate zurücksetzen
453:   pop ax
454: ; X-Koordinate zurücksetzen
455:   add ax, [WORD zl]
456: ; Nächste Zeile bearbeiten,
457:   add bx, [WORD zu]
458: ; dazu Zeilenoffsets hinzuaddieren
459:   pop cx
460: ; Zeile bei Y-Vergrößerung
461:   loop TextLoop5
462: ;
463: ;nochmals plotten
464:   pop cx
465: ; Zeilenzähler wieder laden
466:   inc di
467:   loop TextLoop2
468: ; Nächste Zeile bearbeiten
469:   pop cx
470: ; Zeichenzähler wieder laden
471:   pop bx

```

```

472:   pop ax
473:   add ax, [WORD zch]
474: ; Nächstes Zeichen
475: ; plotten
476:   add bx, [WORD spt]
477:   inc si
478:   loop TextLoop1
479:   pop di
480: ; Alle Register wieder vom
481: ; Stack herunterholen
482:   pop si
483:   pop es
484:   pop dx
485:   pop cx
486:   pop bx
487:   pop ax
488:   ret
489: ; Back home
490: ENDP
491: GfxStart:
492:   mov ax, @data
493:   mov ds, ax
494:   mov [kipptab], 1
495:   mov cl, 15
496: ; Startfarbe wählen
497:   mov ch, 8
498: ; Zeichensatz wählen
499: _loop:
500:   call AsmInitGraphic
501: ; Grafikmodus einschalten
502:   mov [zaehl], 3
503: ; 3 Durchgänge
504:   mov di, [kipptab]
505:   dec di
506:   shl di, 1
507:   shl di, 1
508:   shl di, 1
509:   add di, offset kipptab
510:   mov ax, [di+4]
511: ; Offsets holen
512:   mov bx, [di+6]
513:   mov xofs, ax
514:   mov yofs, bx
515:   mov ax, [di]
516: ; Koordinaten holen
517:   mov bx, [di+2]

```

```

518:   push ds
519:   pop es
520:   mov dl, 1
521: ; X-Vergrößerung setzen
522:   mov dh, 1
523: ; Y-Vergrößerung setzen
524:   mov si, Offset demotext
525: _loop1:
526:   mov di, [kipptab]
527: ; Schrifttyp wählen
528:   add di, [kipptab]
529:   add di, [kipptab]
530:   sub di, [zaehl]
531:   and di, 00FFh
532: ; Überschreibmodus setzen
533:   call AsmText
534:   add ax, xofs
535:   add bx, yofs
536:   add zch, 8
537:   inc dl
538:   inc dh
539:   dec cl
540:   dec [zaehl]
541:   mov di, [zaehl]
542:   or di, di
543:   jnz _loop1
544:   xor ah, ah
545: ; Zeichen abholen
546:   int 16h
547:   inc [kipptab]
548: ; Nächsten Bildschirm
549:   mov ax, [kipptab]
550:   cmp ax, 5
551:   jnz _loop
552:   call AsmInitText
553: ; Textmodus wieder einschalten
554:   mov ax, 4C00h
555: ; Zurück zu DOS
556:   int 21h
557: END GfxStart

```

»text.asm« zeigt die Verwendung von Text in Grafikapplikationen.

**Zeigen Sie sich doch einmal von Ihrer besten Seite.
Zum Beispiel hier!**

**Anzeigenschluß für die Ausgabe Nr. 2/94 ist der
20.04.94**

**Wenden Sie sich einfach
gleich an Ihren nächsten Mediaberater:**

**DMV Daten- und Medienverlag
Gruberstraße 46a • 85586 Poing
☎ 08121/769-300**

**DMV Daten- und Medienverlag
Fuldaer Straße 6 • 37269 Eschwege
☎ 05651/809-375**

**DMV Daten- und Medienverlag
Postfach 130309 • 20103 Hamburg
☎ 040/40 44 12 05**

Aufgeschlüsselte Standards

Unterschiedliche Daten werden als Informationen unterschiedlich für den Rechner auf Datenträgern wie Festplatten oder Disketten gespeichert. Die Daten werden in Standardformate umgewandelt. Für viele dieser Datenformate, wie Texte oder Datenbankbestände, gibt es dabei mindestens ein Standardformat, welches von den meisten Programmen gelesen und geschrieben werden kann.

Während für einfache Texte das ASCII-, für formatierte das Word- und für Datenbankdateien das dBase-Format als Standard existieren, gibt es einen Standard für Grafikdateien leider noch nicht. Zwar sind erste Schritte in Richtung einheitlicher Grafikstandard getan worden, durch immer neue Programme vermehrte sich jedoch die Zahl der verwendeten Grafikformate. Seitdem sich Windows durchsetzen konnte, bekommt auch dessen Grafikformat mit der Dateierendung BMP eine stärkere Rolle. So steht neben TIF, GIF, PCX und den anderen Formaten eine weitere Form der Grafikspeicherung zur Verfügung, die unverzichtbar ist. Für den Programmierer wird es jedoch schwierig, mit den einzelnen Grafikformaten in eigenen Programmen umzugehen. Daher werden hier die verbreiteten Grafikformate vorgestellt.

Das GIF-Format

Die Firma Compuserve hat 1987 ein einheitliches Grafikformat für den Datenaustausch über Mailboxen geschaffen. Das GIF-Format enthält ein Datenprotokoll, das in der PC-Welt und auf allen anderen Rechnern, vom C64 bis zum Apple Macintosh, eingesetzt wird. In einer einzelnen Datei können mehrere Bilder abgelegt werden. Zur Datenübertragung eignet sich

das Format wegen eines implementierten Komprimierungsverfahrens. So wird die Datenmenge automatisch verkleinert, was die Übertragungszeiten über das Telefonnetz verkürzt. Dies senkt natürlich die Kosten. Ein weiterer Vorteil liegt in der hardwareunabhängigen Speichermethode der Daten.

Das GIF-Format kann Bilder mit einer maximalen Auflösung von 16 000 x 16 000 Punkten ablegen. Bis zu 256 Farben aus über 16 Millionen Farbstufen lassen sich abbilden. Die GIF-Datei besteht aus unterschiedlichen Blöcken, die sich ungefähr wie in Tabelle 1 aufteilen.

Die GIF-Datei beginnt mit einer Signatur, die sie als GIF-Format erkenntlich macht. Nur wenn die ersten drei Zeichen der Datei den Text »GIF« enthalten, handelt es sich auch wirklich um eine GIF-Datei. Die folgenden drei Zeichen kennzeichnen die Version des verwendeten GIF-Formates. Weit verbreitet ist immer noch Version 87a, obwohl bereits Version 89a existiert.

Wir beziehen uns hier auf das ältere Format 87a. Da aber nur geringe Veränderungen vorgenommen wurden, sind die Anga-

ben auch auf 89a übertragbar.

An Signatur und Formatangabe schließt sich ein sieben Byte großer Block mit den Informationen für die Bildschirmdefinition an (Tabelle 2). Hier werden Bildschirmauflösung und Farbtiefe sowie die Hintergrundfarbe definiert.

Die ersten vier Byte

des Bildschirmdeskriptors beinhalten die zu verwendende Grafikauflösung. Entsprechend dem Intel-Format wird zuerst das niederwertige, dann das höherwertige Bit abgelegt. Um den korrekten Wert für die Bildbreite zu erhalten, muß also das zweite Byte mit 256 multipliziert werden. Anschließend wird zu dem Ergebnis der Inhalt des ersten Byte addiert. Genauso muß mit den folgenden beiden Byte verfahren werden, wenn die Bildhöhe ausgelesen werden soll.

Das Auflösungs-Byte enthält mehrere Informationen, die in den einzelnen Bits abgelegt sind. Das Bit 7 enthält die Informationen über die globale Farbtabelle. Ist Bit 7 gesetzt, ist eine globale Farbtabelle vorhanden. Ist es nicht gesetzt, ist keine globale Farbtabelle definiert. Statt dessen werden lokale Farbtabelle eingesetzt.

Sie können das Programm so erweitern, daß auch lokale Farbtabelle korrekt berücksichtigt werden. Die folgenden drei Bits (6, 5 und 4) geben an, wie viele Bits in der Farbdefinition zur Angabe der einzelnen Farbanteile verwendet werden.

Offset vom Dateianfang	Byte	Feld
06	2	Bildbreite
08	2	Bildhöhe
0A	1	Auflösung
0B	1	Hintergrundfarbe
0C	1	Ende des Blockes: '0'

Tabelle 2. Der Bildschirmdeskriptor einer GIF-Datei

Die Farbtiefe ergibt sich aus dem abgelegten Wert plus 1. Maximal können also acht Bit zur Definition einer Farbe in der Farbtabelle verwendet werden. Dann handelt es sich um eine Grafik mit bis zu 256 aus 16 Millionen Farben. Mindestens muß jedoch ein Bit zur Ablage der Farbinformationen in der Farbtabelle verwendet werden. Dann liegt eine zweifarbige Grafik vor.

Bit 3 ist in der Version 87a immer mit dem Wert 0 belegt. Daran schließt sich die Definition der Pixeltiefe im Rasterblock an. Hier wird angegeben, wie viele Bit im Rasterblock zur Definition eines Bildschirmpunktes verwendet werden. Wie oben berechnet sich der Wert aus dem ausgelesenen Betrag plus 1. Der sich ergebende Zahlenwert kennzeichnet die Anzahl der Bits, die zur Darstellung eines Bildschirmpunktes verwendet werden.

Codierung des Auflösungs-Bytes:

Bit: 7 6 5 4 3 2 1

Information: Globale Farbtabelle vorhanden; Farbaufklärung in Bit 0 (reserviert); Pixeltiefe in Bit

Das vorletzte Byte der Bildschirmbeschreibung enthält die Hintergrundfarbe. Hier findet sich ein Zahlenwert, der auf den entsprechenden Eintrag in der Farbta-

Offset	Byte	Feld
00	3	Signatur der GIF-Datei: 'GIF'
03	3	Version des verwendeten GIF-Formates: meist '87a'
06	7	Bildschirmbeschreibungen (Deskriptor)
0D	n	globale Farbtabelle (falls vorhanden)
...	n	Erweiterungsblock (optional)
...	n	Bildbeschreibung (Deskriptor)
...	n	lokale Farbtabelle (falls vorhanden)
...	n	Rasterdaten (Farbinformationen)
...	n	Erweiterungsblock (optional)
...	1	Terminator: '89hex'

Tabelle 1. Aufbau einer GIF-Datei

belle verweist. Mit dieser Farbe ist der Hintergrund der GIF-Grafik auszufüllen, bevor eine Grafik auf dem Bildschirm ausgegeben wird. Das letzte Byte der Bildschirmbeschreibung enthält meist den Wert 0. Hier kann jedoch auch ein anderer Zahlenwert auftauchen. Dieser gibt dann das Auflösungsverhältnis (Aspect Ratio) des Bildschirms an.

Codierung des Auflösungs-Bytes:

Bit: 7 6 5 4 3 2 1

Information: Sortierte Globale Farbtabelle; Pixel Aspect Ratio

Byte	Bedeutung
1	Helligkeit Rot Farbe 1
2	Helligkeit Grün Farbe 1
3	Helligkeit Blau Farbe 1
4	Helligkeit Rot Farbe 2
5	Helligkeit Grün Farbe 2
6	Helligkeit Blau Farbe 2
$2^{\text{Farbtiefe}} * 3 - 2$	Helligkeit Rot Farbe $2^{\text{Farbtiefe}}$
$2^{\text{Farbtiefe}} * 3 - 1$	Helligkeit Grün Farbe $2^{\text{Farbtiefe}}$
$2^{\text{Farbtiefe}} * 3$	Helligkeit Blau Farbe $2^{\text{Farbtiefe}}$

Tabelle 3. Aufbau einer GIF-Farbtabelle

Die Farbtabelle

Auf die Bildschirmdefinition folgt die globale Farbtabelle, falls eine solche vorhanden ist. Hier sind die Definition der einzelnen Farbwerte und deren Zuordnung für die Speicherung der Bilddaten im Rasterblock abgelegt. Die Anzahl der Bits, die für das Speichern einer Farbinformation verwendet werden, ist dabei im Block mit der Bildschirmbeschreibung abzulesen (Tabelle 3).

Daraus ergibt sich auch die Anzahl Farben, die in der Farbpalette eingetragen sind, gleichsam die Farbzahl des Bildes. Diese errechnet sich aus der Potenz $2^{\text{Farbbreite}}$. Bei einer maximalen Farbbreite von acht Bit können also bis zu 256 Farben in einer Grafik verwendet werden. Die Farbtabelle stellt nun die Beziehung zwischen den 256 Farbeinträgen und der tatsächlich zu verwendenden Bildschirmfarbe her, so daß die Daten aus dem Rasterblock korrekt zurückgewonnen werden.

Hierzu wird die Farbpalette durch insgesamt drei Byte pro Farbe definiert. Genau drei aufeinanderfolgende Byte innerhalb der Farbtabelle stellen also einen Farbeintrag dar, jede Farbe wird durch ein Byte-triplet realisiert. Die einzelnen Bytes können dabei Zahlenwerte von 0 bis 256 annehmen.

Sie geben nacheinander die Farbanteile für Rot, Grün und Blau an. Die Farben werden also im RGB-Verfahren definiert. Die Kombination »0/0/0« entspricht also der Farbe Schwarz, bei »256/256/256« muß Weiß als Farbe verwendet werden. Ein sattes Blau wird durch die Kombination

»0/0/256« realisiert. Insgesamt ergeben sich 256^3 Farbkombinationen.

Jeder einzelne Wert für die Helligkeit eines Farbanteils belegt dabei immer genau 1 Byte, unabhängig von der Farbanzahl. Die Länge der Farbtabelle entsteht also folgendermaßen:

Länge in Byte = $2^{\text{Farbtiefe}} * 3$.

Eine lokale Farbtabelle, die vor einem Rasterblock stehen kann, besitzt exakt denselben Aufbau wie die globale Farbtabelle. Auf lokale Farbtabelle kommen wir später zurück. Wird für einen Bildschirmbereich eine lokale Farbtabelle definiert, hat innerhalb dieses Bereiches die lokale Farbtabelle Vorrang vor der globalen Farbtabelle. Hier sind also die entsprechenden Farbdefinitionen der lokalen Farbtabelle einzusetzen. So können in einer GIF-Datei auch Grafiken mit mehr als 256 Farben abgelegt werden.

Erweiterungsblöcke

Sie können in einer GIF-Datei auf die globale Farbtabelle folgen. Sie müssen jedoch nicht vorhanden sein. Man kann sie am einleitenden Zeichen »!<« (ASCII=21hex) erkennen.

In einem Erweiterungsblock können Informationen über die verwendete Hard- und Software, mit der man die Grafik anfertigte, abgelegt werden.

Länge in Byte	Feld
1	Signatur des Erweiterungsblockes: »!<« (ASCII 21hex)
1	Funktionscode
1	Länge des ersten Datenblockes
n	Datenblock 1
1	Länge des zweiten Datenblockes
n	Datenblock 2
...	...
1	Länge des letzten Datenblockes
n	letzter Datenblock
1	Terminator: 00hex

Tabelle 4. Aufbau eines GIF-Erweiterungsblockes

Des weiteren können Erweiterungsblöcke auch neue Grafikfunktionen beinhalten, die im ursprünglichen GIF-Format noch nicht berücksichtigt wurden. Die Art der abgelegten Daten in einem Datenblock kann an Hand des Funktionscodes im zweiten Byte des Datenblockes abgelesen werden. Leider liegen keine genaueren Informationen über die möglichen Funktionen eines Erweiterungsblockes und deren Codes vor.

Anschließend folgen die einzelnen Records mit den Daten. Das erste Byte eines Records gibt immer die Länge des nachfolgenden Datenblockes an. Ein Datenblock kann also maximal 256 inklusive Längenbyte umfassen. Läßt sich die benötigte Information nicht in einem Block unterbringen, müssen mehrere Blöcke eingesetzt werden, die direkt aufeinander folgen. Am Anfang eines jeden Blockes steht wiederum die Länge des nachfolgenden Datenblockes.

Das Ende eines Erweiterungsblockes kennzeichnet das Zeichen 00hex als Terminator. Insgesamt ergibt sich also der Aufbau eines Erweiterungsblockes, den Sie Tabelle 4 entnehmen können.

Bildbeschreibungsblock

Vor den endgültigen Rasterdaten mit der Beschreibung der einzelnen Bildpunkte wird das Bild beschrieben. Während der Block mit der Bildschirmbeschreibung Informationen über Grafikauflösung und Farbzahl enthält, verfügt dieser Block über entsprechende Informationen zu Bildgröße und -positionierung (Tabelle 5).

Erkannt werden kann die Bildbeschreibung durch den einleitenden Code »,<« (ASCII 2Chex). Auf dieses Erkennungszeichen folgen zwei Byte mit der Koordinate des linken Bildrandes und zwei Byte mit der Koordinate des oberen Bildrandes. Sie geben die Position der Grafik auf dem Grafikbildschirm an.

Anschließend folgen zwei Byte zur Breite der Grafik und zwei zur Höhe. Wie bei der Bildschirmauflösung wird auch hier zuerst das niederwertige, anschließend das höherwertige Byte abgelegt. Die Rückgewinnung der tatsächlichen Bildgröße erfolgt also wie oben für die Grafikauflösung beschrieben.

Schließlich folgt ein letztes Byte, das verschiedene Flags, also Statusanzeigen, beinhaltet. Das Bit 7 gibt an, ob eine lokale Farbtabelle für diese Grafik vorhanden ist. Sollte dieses Bit gesetzt sein, folgt auf die Grafikbeschreibung eine zusätzliche Farbtabelle, die für diesen Grafikteil ihre Gültigkeit besitzt.

Bit 6 gibt an, ob das Bild interlaced oder non-interlaced gespeichert wurde. Im Interlaced-Modus wird jeweils nur jede achte Zeile gesichert. Sobald das Ende der Grafik erreicht wird, fängt die Datenablage wieder mit der nächsten Zeile von oben aus an. Es werden also nacheinander die

Offset	Datentyp	Feld
0	Byte	Erkennungszeichen: »,<« (ASCII 2Chex)
1	Word	linke Koordinate der linken oberen Bildecke
3	Word	obere Koordinate der linken oberen Bildecke
5	Word	Bildbreite
7	Word	Bildhöhe
9	Word	Flags

Tabelle 5. Der Aufbau des Bildbeschreibungsblockes (Image Descriptor)

Zeilen 0, 8 und 16 übertragen. Anschließend folgen die Zeilen 4, 20 und 28 usw. Die Reihenfolge der Zeilen entspricht folgendem Muster: 0, 6, 4, 2, 1, 3, 5, 7.

Im Gegensatz dazu wird das Bild im sogenannten sequentiellen Modus Zeile für Zeile abgelegt. Der Interlaced-Modus bietet dabei bei der Datenübertragung Vorteile und erlaubt auch ein schnelleres Wiedererkennen der Grafik beim Bildschirmaufbau. Bit 5 gibt an, ob die lokale Farbtabelle nach der Frequenz oder der Farbe sortiert ist. Die letzten drei Bits geben wiederum an, wie viele Bits für eine Dateninformation im Rasterblock abgelegt werden.

Die Rasterdaten

Auf den Image-Deskriptor kann noch eine lokale Farbtabelle folgen. Diese besitzt dann denselben Aufbau wie die oben beschriebene globale Farbtabelle. Die lokale Tabelle gilt jedoch nur für den im Image-Deskriptor beschriebenen Grafikbereich. Anschließend folgen die Rasterdaten, die die eigentliche Information über die Grafik enthalten. In den Rasterdatenblöcken (Tabelle 6) werden Relationen zwischen der Farbtabelle und der zu setzenden Bildschirmfarbe an einem Punkt hergestellt.

Jedem Bildschirmpunkt wird die entsprechende Zahl der Farbe aus der Farbtabelle zugeordnet. So werden die einzelnen Informationen für jeden Bildschirmpunkt abgelegt. Das GIF-Format verwendet jedoch ein zusätzliches Komprimierungsverfahren, das die Länge des Rasterblockes erheblich verkürzt. So werden kleinere Dateien und geringere Übertragungszeiten im Postnetz erreicht.

Die Rasterdaten werden jeweils blockweise abgelegt. Am Anfang steht ein Byte, das angibt, wie viele Datenbytes im folgenden Datenblock vorzufinden sind. Sind mehr als 256 Datenbytes vorhanden, folgt auf das erste Zählbyte und die entsprechende Zahl an Datenbytes ein weiteres Zählbit. So können beliebig viele Datenblöcke abgelegt werden. Den letzten Datenblock kennzeichnet ein Zählbyte mit dem Inhalt 00hex als Terminator.

Eingesetzt wird ein leicht modifiziertes LZW-Verfahren. Zunächst werden nur so viele Bits für die Speicherung eines Bildpunktes verwendet, wie tatsächlich benötigt werden. Diese Zahl wird ja im Image-Deskriptor im letzten Byte abgelegt. Bei einer Grafik mit 16 Farben reichen so zum Beispiel vier Bit, um die Informationen für einen Grafikpunkt abzulegen. In einem Byte lassen sich also zwei Pixel beschreiben.

Dies ist jedoch nicht die einzige Verkürzung. Zusätzlich kann man mit einem einzelnen Bit mehrere nebeneinanderliegende Bildpunkte als Kombination aus verschiedenen Farben beschreiben. Das LZW-Verfahren definiert neben der Farbtabelle zwei zusätzliche Codes. Den Code »IC« Einrichten der Codierung und den Code »EC« zum Markieren des Datenendes. Die Va-

riable x soll im folgenden der Zahl der verwendeten Bits pro Bildpunkt entsprechen. Den neuen Codes werden nun die Zahlenwerte 2^x und 2^{x+1} zugeordnet. Um neben den Farbinformationen auch diese beiden Codes verwenden zu können, muß natürlich die Bitbreite einer Farbinformation um den Wert 1 erhöht werden. In einer Grafik mit 16 Farben würden die Codes »IC« und »EC« die Werte 16 und 17 belegen. Diese Zahlen können jedoch erst mit insgesamt fünf Bit dargestellt werden. Somit werden nun fünf Bit für das Ablegen eines Bildpunktes verwendet.

Nun stehen jedoch zusätzliche Codes jenseits von $(2^x)+1$ zur Verfügung. Sie reichen bis zu $(2^{x+1})-1$. Diese zusätzlichen Zahlenwerte können verwendet werden, um Kombinationen aus den einzelnen Farben der Farbtabelle zu speichern. So erst läßt sich Speicherplatz sparen. Zunächst wird eine Codetabelle eingerichtet, die neben der Farbtabelle die zusätzlichen Codes »IC« und »EC« sowie oberhalb von »EC« freie Plätze besitzt. Insgesamt umfaßt die Tabelle 2^{x+1} Einträge.

Jetzt beginnt die eigentliche Codierung der

Länge in Byte	Feld
1	Anzahl der Bit pro Pixel
1	Zahl der Bytes im Datenblock
n	Datenbytes
1	Zählbytes mit Anzahl der Datenbytes im nächsten Datenblock
n	Datenbytes
..	..
1	Terminator (00hex)

Tabelle 6. Der Aufbau des Rasterdatenblockes

Farbinformation. Zunächst wird das erste Pixel der Farbinformationen betrachtet. Zur Komprimierung wird neben dem aktuellen Farbwert auch ein sogenanntes Präfix definiert, welches mehrere Farbwerte aufnehmen kann. Zuerst ermittelt man den aktuellen Farbcode, das Präfix ist leer.

Die Farbinformation liegt an der entsprechenden Position in der Tabelle vor. Somit wird die aktuelle Farbinformation in den Präfix übernommen und der nächste Farbwert eingelesen. Jetzt entsteht eine Kombination aus Präfix und aktuellem Farbwert, die in der Tabelle noch nicht vorhanden ist. Diese Kombination aus Präfix und aktueller Farbe wird am ersten freien Tabellenplatz, also $(2^x)+1$ abgelegt. Der Inhalt des Präfix wird an die Ausgaberroutine geschickt. Diese kann den Farbwert zum Beispiel auf Diskette schreiben.

Das Präfix wird geleert und der aktuelle Farbzustand in das Präfix übernommen. Der folgende Farbwert wird eingelesen und erneut die Kombination aus Präfix und aktueller Farbe mit der Farbtabelle verglichen. Existiert für die Kombination ein Eintrag in der Tabelle, wird die aktuelle

Farbe dem Präfix hinzugefügt und der nächste Farbwert eingelesen. Erneut wird ein Vergleich vorgenommen.

Erst wenn eine Kombination von Präfix und aktueller Farbe auftritt, die in der Tabelle nicht vermerkt ist, wird diese an den ersten freien Tabellenplatz geschrieben. Der Inhalt des Präfix wird wieder zur Ausgaberroutine geschickt, das Präfix entleert und der aktuelle Farbwert zum Inhalt des Präfix gemacht.

Immer dann, wenn eine Kombination aus mehreren Bildschirmpunkten bereits bekannt und in der Tabelle vermerkt ist, tritt beim Speichern eine Verkürzung der Informationen auf. Sobald die Tabelle mit neuen Kombinationen komplett gefüllt ist, wird die Bitbreite der Informationen für einen Bildpunkt um 1 erhöht. Entsprechend wird die Tabelle um die neuen Werte erweitert, so daß neuer Platz für unbekannte Kombinationen entsteht. Gleichzeitig werden nicht mehr $x+1$, sondern $x+2$ Bit für das Ablegen eines Wertes aus der Tabelle verwendet.

Das Ende des verschlüsselten Codes wird mit dem Wert für das Zeichen »EC« gekennzeichnet. Da das GIF-Format keine breiteren Bit-Informationen als maximal 12 Bit unterstützt, wird zudem in seltenen Fällen der Code »IC« benutzt. Sobald eine Tabelle bei einer Bitbreite von 12 Bit gefüllt ist, dies entspricht 4096 Einträgen, müßte die Bitbreite auf 13 Bit erhöht werden. Dies ist jedoch nicht der Fall. Statt dessen wird der Code für das Zeichen »IC« gesendet. An dieser Stelle ist die Bitbreite wieder auf den Ausgangswert zurückzusetzen und eine neue Tabelle anzufertigen, die nur die Standard-Farbdefinitionen enthält. Die Codierung wird dann mit der neuen Farbtabelle wieder aufgenommen.

Die Decodierung verschlüsselter Daten verläuft entsprechend. Zunächst wird eine Tabelle mit 2^{x+1} Einträgen angelegt. Die Einträge von 0 bis $(2^x)-1$ werden mit den Informationen der Farbtabelle erhöht. Die Werte 2^x und $(2^x)+1$ entsprechen den Werten »IC« und »EC«. Die Einträge von $(2^x)+2$ bis 2^{x+1} sind noch frei.

Nun wird die erste Information nach dem Schlüssel »IC« eingelesen. Dies muß noch ein einzelner Farbpunkt ohne Kombination mehrerer Punkte darstellen (siehe oben). Da der Wert für die Information in der Tabelle vermerkt ist, wird die eingelesene Information in das leere Präfix übernommen und der nächste Zahlenwert eingelesen.

Nun entsteht eine Kombination aus Präfix und aktueller Information, die in der Tabelle nicht abgelegt ist. Der eingelesene Wert für sich ist jedoch wiederum in der Tabelle vermerkt. Die Kombination der beiden bekannten Farbwerte wird nun an die erste freie Position innerhalb der Tabelle eingetragen. Der Inhalt des Präfixes wird an die Ausgaberroutine, zum Beispiel zur Ausgabe auf dem Bildschirm oder zur Konvertierung der Daten, übertragen.

Der eingelesene Farbwert wird nun in das zuvor geleerte Präfix übernommen. Die nächste Information wird nun aus der Tabelle eingelesen. Es entsteht eine neue Kombination aus Präfix und aktueller Information. Ist diese Kombination nicht bekannt, wird sie an die erste freie Stelle in der Tabelle übernommen. Der Inhalt des Präfix wird den entsprechenden Farbwerten zugeordnet und die Farbwerte werden an die Ausgabefunktion befördert.

Der Präfix wird schließlich geleert, der aktuelle Farbwert wird in das Präfix übernommen und der neue Farbwert wird eingelesen. Ist die Kombination aus Präfix und aktuellem Zahlenwert jedoch bereits in der Tabelle vorhanden, so wird der aktuelle Wert zum Präfix hinzugefügt und ein weiterer Zahlenwert aus der Datei eingelesen. Ist die neue Kombination aus Präfix und aktuellem Wert ebenfalls in der Tabelle eingetragen, wird wie oben verfahren, so lange, bis eine unbekannte Kombination entsteht, die in die Tabelle übernommen wird.

Nun wird Ihnen auffallen, daß auch hier irgendwann die Tabelle voll werden kann. In diesem Fall wird die Bitbreite einer Dateninformation ebenfalls um 1 erhöht und die Tabelle entsprechend verlängert. Der Zeitpunkt, an dem diese Verlängerung auftritt, muß dabei dem entsprechen, der auch beim Speichern der Daten vorgefunden wurde.

Dies muß logischerweise so sein, da beim Einlesen von Daten mit der identischen Farbtabelle wie beim Speichern dieselbe Zahl unbekannter Farbkombinationen aus aktuellem Wert und Präfix auftritt wie beim Speichern. Auch muß die Reihenfolge des Auftretens unbekannter Kombinationen beim Laden und Speichern gleich sein. Dies erfolgt automatisch.

So lassen sich alle Informationen über jeden einzelnen Bildpunkt korrekt zurückgewinnen und in ein anderes Dateiformat übertragen. Am Ende eines Rasterblockes tritt der Terminator 00hex auf.

Nun kann ein weiterer Image-Deskriptor folgen, an den sich eine lokale Farbtabelle und zusätzliche Rasterblöcke anschließen. So kann eine komplette Grafik durchaus in mehreren Teilgrafiken abgelegt werden. Die Größe und Position einer Teilgrafik ist im Image-Deskriptor abgelegt. Aus diesen Informationen läßt sich die komplette Grafik zurückgewinnen.

Am Ende einer GIF-Datei kann ein Erweiterungsblock folgen, der dieselbe Funktion wie ein bereits beschriebener Erweiterungsblock erfüllt. Letztes Zeichen in einer GIF-Datei ist der Terminator »€« (ASCII: 89hex).

Das PCX-Format

Als nächsten Vertreter der Grafikformate schlüsseln wir das PCX-Format auf. Im Speicherverfahren bestehen zwischen GIF-

Offset	Länge in Byte	Feld
00	128	Dateikopf (Header)
80	n	Bilddaten
80+n+1		optionale Palettendefinition (nur bei Grafiken mit mehr als 16 Farben)

Tabelle 7. Der Aufbau einer PCX-Datei

und PCX-Format einige grundlegende Unterschiede.

Es wird auch von zahlreichen fremden Anwendungen wie DTP-Programmen unterstützt. Die PCX-Datei besteht aus einem PCX-Header, also einem Dateikopf, sowie den eigentlichen Grafikdaten. Bei VGA-Grafiken mit mehr als 16 Farben kann am Ende der Datei im Anschluß an die Bilddaten noch ein eigener Block mit der Palettendefinition für die Farben stehen. An sich ist die Palettendefinition jedoch Bestandteil des Dateikopfes (Tabelle 7).

● Dateikopf einer PCX-Datei

Eine PCX-Datei besitzt, unabhängig von der Version des Grafikformates, immer einen einheitlichen Dateikopf, der aus 128 Byte besteht. Dies gilt sowohl für PCX-Grafiken als auch für Bildausschnitte, die unter der Dateiendung »PCC« abgelegt werden. Der Dateikopf ist in Tabelle 8 dargestellt.

Am Anfang jeder PCX-Datei muß als Kennung der Wert 10 (0Ahex) stehen. Nur wenn dieses Zeichen existiert, handelt es sich wirklich um eine PCX-Datei. Anschließend wird das verwendete Format des PCX-Speicherverfahrens deklariert. Wenn hier der Wert 2 auftritt, ist eine zusätzliche Palettendefinition in die PCX-Datei integriert. Bei Farbdarstellungen werden die Palettendaten dann ab dem Offset 44hex abgelegt. Ansonsten fehlen die Palettendaten.

Das dritte Byte gibt Auskunft über das Kompressionsverfahren. Entweder werden die Grafikdaten nicht gepackt oder das Run-Length-Encoding-Verfahren zum Codieren der Grafikinformationen wird eingesetzt. Hierauf kommen wir später

zu sprechen. Anschließend findet sich im Header die Information über die Anzahl der Bits, die pro Pixel abgelegt werden. Meist wird hier der Wert 1 abgelegt.

Daran schließt sich ein acht Byte großer Block an, der die Größe des Bildschirm-ausschnittes und die obere, linke und die untere, rechte Ecke des Bildes definiert. Die Angabe der Informationen erfolgt, wie beim GIF-Format, in der Intel-Notierung: Lowbyte – Highbyte.

Die hier abgelegten Daten sind auch bei der Rückgewinnung des Grafikinhalt von Bedeutung, da im Datenbereich die einzelnen Bildzeilen nur in kompletten Worten abgelegt werden. Somit kann am rechten und unteren Rand etwas zuviel an Informationen in den Daten vorhanden sein. Diese müssen entsprechend abgeschnitten werden. Doch auch dazu später mehr.

Die folgenden beiden Worte beinhalten Informationen über den zu verwendenden Grafikmodus. Hier finden sich die horizontale und die vertikale Auflösung des Bildes. In den unterschiedlichen Formaten des PCX-Verfahrens werden diese Werte

Offset	Länge in Byte	Feld
0	1	Identifikationsbyte: 0Ahex (Kennung für PCX-Datei)
1	1	Version des PCX-Formates: 0 = 2.5, 2 = 2.8 mit Palettendaten, 3 = 2.8 ohne Palettendaten, 5 = 3.0
2	1	Kennungsbyte für komprimierte PCX-Dateien: 0 = uncodiert, 1 = RLE-Codierung
3	1	Bits pro Pixel (oder pro Ebene): 1 = S/W-Grafik, 2 = CGA-Grafik (4 Farben), 4 = EGA-Grafik (16 Farben), 8 = VGA-Grafik (256 Farben)
4	2	untere x-Koordinate der Bildschirmposition (XMIN)
6	2	untere y-Koordinate der Bildschirmposition (YMIN)
8	2	obere x-Koordinate der Bildschirmposition (XMAX)
A	2	obere y-Koordinate der Bildschirmposition (YMAX)
C	2	horizontale Auflösung eines Punktes in dots per inch (dpi)
E	2	vertikale Auflösung eines Punktes in dots per inch (dpi)
10	48	Farbtabelle mit der Definition der Farbpalette, organisiert als 3-Byte Felder. Kann maximal 16 Farben aufnehmen.
40	1	reserviert
41	1	Zahl der Farbebenen (maximal 4)
42	2	Bytes pro Bildzeile (gerade Zahl)
44	2	Palettendaten: 1 = Farbe - S/W, 2 = Graustufen
46	58	Leerbytes zum Erreichen der Headergröße (128 Byte)

Tabelle 8. Der Aufbau des PCX-Headers

jedoch leicht modifiziert, so daß sie nur mit Vorsicht zu genießen sind.

Ab dem Offset 10hex ist ein 28 Byte langer Bereich zur Aufnahme der Farbpalettendefinition für Grafiken mit bis zu 16 Farben reserviert. In jeweils drei Bytes wird die Information für die zu verwendende Farbe abgelegt. Somit können hier maximal 16 Farbdefinitionen untergebracht werden. Die Reihenfolge der einzelnen Bytes entspricht dem Vorhandensein der Farben Rot, Grün und Blau in der gewünschten Farbe. Durch ein zusätzliches Intensitätsbit, welches nicht in der Palettendefinition enthalten ist, lassen sich so insgesamt 16 Farben darstellen.

Auf neueren Grafikkarten lassen sich jedoch weit mehr Farben darstellen. Hier können die Grundfarben auch in unterschiedlichen Intensitäten miteinander gemischt werden. Durch die Verwendung eines Bytes zum Darstellen der Anteile einer Grundfarbe in einer gewünschten Farbe lassen sich bis zu 255 Farbabstufungen der Grundfarbe erreichen. Die einzelnen Farben werden dann, um die Datenmenge im Bildbereich zu verringern, in einer Palette abgelegt. Nun wird, wie beim GIF-Verfahren, lediglich noch ein Bezug zwischen der Farbe eines Pixels und der Palette hergestellt.

Somit kann ein Pixel in einer 16farbigen Grafik durch 4 Bit, in einer Grafik mit 256 Farben durch 8 Bit dargestellt werden. Das Speichern der Farbpalette wird im PCX-Format jedoch je nach verwendeter Farbzahl und Grafikkarte unterschiedlich vorgenommen.

● CGA-Farbpalette

Das Speicherformat der CGA-Farbpalette ist gleichzeitig das komplizierteste unter den Palettenformaten. Wie oben beschrieben, werden die insgesamt 48 Byte in 3 x 16 Byte unterteilt. Jedem Tripel wird eine der möglichen Farben zugeordnet. Die einzelnen Bytes eines Tripels nehmen die eigentlichen Farbinformationen des gewünschten Farbtons auf.

Im ersten Byte eines Tripels steht dabei die Hintergrundfarbe. Es werden nur die obersten vier Bit zur Darstellung der Hintergrundfarbe eingesetzt. Somit kann eine Auswahl aus 16 Farben vorgenommen werden. Im zweiten Byte des Tripels finden sich die Informationen über die gewünschte Farbpalette. Hier werden nur die oberen drei Bit belegt.

In Bit 7 ist die Information über die Farbe oder Monochrom-Palette abgelegt. Der Wert 0 entspricht einer Farbpalette, der Wert 1 hingegen definierte eine S/W-Palette. In Bit 6 wird die gewünschte Farbpalette ausgewählt. Der Wert 0 steht für eine Farbpalette aus den Farben Grün, Rot und Braun, der Wert 1 verwendet die Palette aus den Farben Cyan, Magenta und Weiß. Bit 5 schließlich gibt die Intensität eines Bildpunktes an. Der Wert 0 erzeugt einen dunklen, der Wert 1 einen hellen Bild-

punkt. Mit Hilfe der drei Bits lassen sich acht verschiedene Kombinationen realisieren. Das letzte Byte des Tripels ist unbelegt.

● EGA/VGA-Palette mit 16 Farben

Bei einer Grafik mit insgesamt 16 Farben werden die 48 Byte des Palettenblockes wiederum zu insgesamt 16 Tripeln zusammengefaßt. Jedem Byte wird eine der Grundfarben in der Reihenfolge Rot, Grün und Blau zugeordnet. Das Byte enthält dabei den Farbanteil der jeweiligen Grundfarbe in dem gewünschten Farbton. Es werden jedoch nur die oberen beiden Bits eines jeden Bytes zur Speicherung der Informationen verwendet. Somit ergeben sich jeweils vier Farbabstufungen.

Durch Division des Zahlenwertes in einem Byte durch die Zahl 64 erhält man die Anteile in den Abstufungen 0, 1, 2 oder 3. Ein reines Grün wird dabei durch die Kombination 0, 3, 0 dargestellt.

● VGA-Palette mit 256 Farben

In einem Byte der Palettendefinition lassen sich im Grunde genommen 256 Farbabstufungen definieren. Um

auch Grafiken einer VGA-Karte mit bis zu 256 Farben speichern zu können, wurde die Version 3.0 des PCX-Formates definiert. Hier findet sich am Ende der Grafikdatei ein Block mit der Farbpalettendefinition. Die 48 Byte im Header weisen in diesem Falle alle den Wert 05hex auf. Der Anfang der VGA-Palettendefinition wird durch das Zeichen 0Ch gekennzeichnet. Findet sich dieses Zeichen an der 769. Stelle vor dem Dateende, so folgt anschließend die gültige Palettendefinition in den verbleibenden 768 Byte. Die Farbdefinition erfolgt wie bei der 16-Farb-Grafik und wie beim GIF-Verfahren durch die Angabe der Farbanteile eines Farbtones. Jeder Farbton wird wiederum durch ein Tripel realisiert.

● Der Rest des Headers

Auf die 48 Byte zur Aufnahme einer Palettendefinition folgt im Dateikopf die Zahl der Farbebenen in der Grafik. Hier sind maximal vier Farbebenen machbar. Auf die Bedeutung einer Farbebene kommen wir später zu sprechen.

Anschließend wird definiert, wieviele Bytes die Informationen einer Grafikzeile enthalten. Da die Datenablage in Worten erfolgt, muß in dem zwei Byte umfassenden Feld ein gerader Zahlenwert abgelegt sein. Auf die Bedeutung dieses Wertes kommen wir ebenfalls bei der Besprechung des Datenblockes zurück.

Das Datenwort am Offset 44hex enthält

schließlich Informationen über die Interpretation der Farbinformationen. Bei einem Wert 1 sind die Eintragungen der Palettendefinition als S/W- oder als Farbpalette zu interpretieren. Bei dem Wert 2 sind alle Farbinformationen hingegen als Graustufen zu deuten.

Der Header schließt mit Leerbytes (00hex) ab, um die vorgegebene Kopflänge von 128 Byte zu erreichen. Gleichzeitig können hier zusätzliche Daten in einer der folgenden PCX-Versionen abgelegt werden.

● Der Datenbereich

Auf die ersten 128 Byte des Headers folgt der eigentliche Datenbereich mit den Grafikinformationen. Hierzu wird jeder Bildpunkt in unterschiedlichen Ebenen abgelegt. Insgesamt können bis zu vier Ebenen zur Speicherung der Daten verwendet werden, die die Informationen für einen Bildpunkt enthalten.

Offset	Länge in Byte	Feld
00	8	Header
08	n	IFD 1
..	..	IFD 2
..	..	Daten
..	..	..
..	..	IFD n

Tabelle 9. Der Aufbau einer TIF-Datei

Die Anzahl der Speicherebenen steht ebenfalls im Dateikopf. Ebenso die Zahl der Bytes pro Zeile. Jede einzelne Bildzeile wird nämlich einzeln abgelegt. Die Speicherung einer kompletten Zeile erfolgt in jeder Ebene wortweise. Eine Informationseinheit, die durchaus mehrere Bildpunkte beschreiben kann, besteht also immer aus zwei Byte.

Wenn am Rand des Bildes noch freie Bytes vorhanden sind, die zusätzliche Bildpunkte aufnehmen können, sind diese durch Leerstellen (00hex) an das Zeilenende angefügt. Diese müssen zur Rückgewinnung der korrekten Grafik abgeschnitten werden. Das läßt sich mit Hilfe der Koordinatenangaben im Dateikopf erledigen.

Die einzelnen Speicherebenen einer Zeile teilen sich in die einzelnen Farbwerte Rot, Grün und Blau sowie in die Farbintensität auf. In S/W-Grafiken wird natürlich nur eine Farbebene für das Speichern der Daten eingesetzt. Auf VGA-Karten werden üblicherweise alle vier Ebenen genutzt.

Zur Speicherung der Farbinformationen wird eine komplette Grafikzeile eingelesen. Anschließend wird sie in die einzelnen Farbanteile wie oben beschrieben zerlegt. Die einzelnen Farbzeilen werden nun, entsprechend aufgefüllt, abgelegt. Wie beim GIF-Verfahren werden die einzelnen Bildpunktinformationen im uncodierten Code direkt aneinander gehängt.

So beinhaltet ein Byte in einer S/W-Grafik also die Informationen für insgesamt acht

Bildpunkte, da hier nur die Information 1 oder 0 für die Darstellung eines Bildpunktes notwendig ist.

Nachdem so die rote Farbebene abgelegt wurde, wird entsprechend mit der grünen und der blauen Farbebene verfahren. Schließlich wird die Intensität einzeln abgelegt. Dann erst kann die nächste Grafikzeile eingelesen und gesichert werden. Die auszulesenden Farbwerte können durch direkten Zugriff auf den Videospeicher ermittelt werden.

Im Grunde genommen wird nichts anderes als eine komplette Kopie des Speichers auf die Diskette geschrieben. Bei einer 256farbigen VGA-Grafik werden die Grafikinformationen im Videospeicher ebenfalls in den einzelnen Ebenen getrennt abgelegt.

Die zeilenweise Speicherung wird bis zur letzten Zeile durchgeführt. Eventuell können am Grafikende ebenfalls überflüssige Zeilen angetroffen werden, die entsprechend den Koordinatenangaben ebenfalls abgeschnitten werden müssen.

Die Komprimierung von Daten

Wie Sie bemerkt haben, belegt dieses Speicherverfahren eine Menge Speicherplatz auf dem Datenträger. Daher wird in den meisten PCX-Dateien das RLE-Verfahren zur Datenkomprimierung eingesetzt. Ob die Daten komprimiert sind, wird im Header angezeigt. Das RLE-Verfahren faßt aufeinanderfolgende, gleichartige Pixel zusammen und gibt die Anzahl der Wiederholungen der gleichen Farbinformation an.

Solche identischen Farbinformationen treten zum Beispiel bei großen, gleichmäßig eingefärbten Flächen auf. Aber auch bei eher komplexen Grafiken macht sich die Komprimierung durch die Speicherung der Informationen in den einzelnen Ebenen deutlich bemerkbar.

Das RLE-Verfahren faßt alle identischen, aufeinanderfolgenden Bits zusammen und gibt zusätzlich die Anzahl der im Original vorliegenden Wiederholungen des identischen Bits an. Die Unterscheidung, ob in einer Information nun eine komprimierte Information oder ein gespeicherter Bildpunkt vorliegt, wird anhand der oberen beiden Bits des Bytes vorgenommen.

Sind beide Bits gesetzt, liegt eine komprimierte Information vor. In den restlichen fünf Bit wird die Wiederholrate der Information angegeben, die im nächsten Byte folgt. Im folgenden Byte ist auf jeden Fall die entsprechende Bildinformation abgelegt, unabhängig von den oberen beiden Bits.

Sollte eine Grafikinformation ohne Wiederholung vorliegen, bei der die oberen Bits ebenfalls gesetzt werden müssen, folgt zunächst ein Wiederholungsindikator mit dem -zähler 1. Anschließend folgt die entsprechende Bildinformation. Meist sind jedoch die oberen beiden Bits einer Bildinformation nicht gesetzt. Dann liegt ein reines Datenbyte vor. Die entsprechenden

Grafikinformationen finden sich dann direkt in den Bits 0 bis 5. Wie oben beschrieben, wird eine Zeile nur in kompletten Datenwörtern abgelegt. Jede Zeile muß also aus einer geraden Anzahl Bytes bestehen. Diese Information bezieht sich jedoch auf die unkomprimierte Datenspeicherung. Durch die RLE-Komprimierung kann die tatsächliche Zeilenlänge entsprechend verändert werden. Die entsprechende Zeilenlänge wird aber beim Regenerieren der originalen Daten wieder zurückgewonnen.

Mit diesem Verfahren können bis zu 64 Byte, also maximal 512 Bit komprimiert werden. Wenn eine Grafik jedoch keinerlei Wiederholungen aufweist, wird der Programmcode verlängert, da die Bits 6 und 7 zur Speicherung von Grafikdaten nicht einsetzbar sind.

Ein kleines Beispiel soll zum Abschluß die Komprimierung im RLE-Verfahren und das Problem mit den ersten gesetzten Bits in einem Datenbyte verdeutlichen: die Information aus

63 * 07hex , 12hex , F0hex

soll komprimiert werden. Anfangs entstehen hier keine Probleme. Die Wiederholung von dem Datenbyte 07hex wird durch ein vorangestelltes Byte FFhex angezeigt.

Offset	Länge in Byte	Feld
00	2	Speichermodus: »II« = Intel, »MM« = Motorola
02	2	Versionsnummer des TIF-Formates
04	4	Zeiger auf die erste IFD

Tabelle 10. Der Aufbau des TIF-Headers

Hier sind die ersten Bits als Indikatoren gesetzt. Die restlichen sechs geben die Wiederholungsrate an, hier 63. Das Datenbyte 12hex kann direkt in den Datenbereich geschrieben werden. Es wird korrekt interpretiert, da die ersten beiden Bits nicht gesetzt sind.

Das Byte F0hex als Datenbyte bereitet jedoch Probleme. Würde es direkt in den Datenbereich geschrieben werden, würde es als Wiederholungsbyte mit einer Wiederholungsrate von 48 fehlinterpretiert werden. Daher muß zunächst ein Wiederholungsbyte mit der Wiederholungsrate 1 eingefügt werden. Somit sieht der komprimierte Code wie folgt aus:

FFhex 07hex 12hex C1hex F0hex

Die Rückgewinnung des Originalcodes erfolgt durch Umkehr des Speicherverfahrens.

Das TIF-Format

Das TIF-Format entstand bereits frühzeitig als Standard für die Grafikspeicherung. Es erlaubt durch seinen modularen Aufbau die Erweiterung des Informationsgehaltes um beliebige Komponenten.

Zudem werden unterschiedliche Speichertypen und auch verschiedene Komprimie-

rungsverfahren eingesetzt. Anders als bei den bisher vorgestellten Grafikformaten verfügt die TIF-Datei über keinen festen Aufbau. Sie besteht aus einem kurzen Dateikopf (Header) und anschließenden Datenblöcken, die als Image File Directories (IFDs) bezeichnet werden. Diese enthalten die eigentlichen Informationen in sogenannten Tags.

Auf die einzelnen IFDs wird mit Hilfe von Dateizeigern verwiesen (Tabelle 9). Die eigentlichen Daten, wie die Rasterdaten, können an beliebigen Freiräumen innerhalb der TIF-Datei abgelegt werden. Sie werden von den Tags aus durch eigene Zeiger adressiert.

Alle benötigten Informationen, von den eigentlichen Rasterdaten über die Bildschirmauflösung bis hin zur Farbdefinition, sind alle Daten in den IFDs untergebracht. Leider macht dieses Speicherverfahren die Datei nicht nur flexibler, sondern auch komplexer und für den Programmierer etwas unübersichtlich.

Der Dateikopf einer TIF-Datei

Der Dateikopf einer TIF-Datei umfaßt nur die ersten acht Byte des Dateianfangs. Hier sind die wichtigsten Informationen über

das TIF-Format, das Speicherformat und der Zeiger auf die erste IFD abgelegt (Tabelle 10).

Wie bereits besprochen, wird das TIF-Format sowohl im PC, also mit einem Intel-Prozessor oder

einem kompatiblen, als auch auf dem Macintosh eingesetzt. Letzterer verfügt über einen Motorola-Prozessor. Das Speicherverfahren von Zahlenwerten in den einzelnen Bytes ist bei diesen Rechnern jedoch unterschiedlich realisiert.

Ein vierstelliger Zahlenwert wird bei Intel im Format Lowbyte – Highbyte abgelegt. Bei einem Motorola-Prozessor ist dies genau umgekehrt.

Anhand der ersten beiden Zeichen des Dateikopfes kann die TIF-Datei erkannt werden. Außerdem wird hier angegeben, in welchem Speicherformat man arbeitet. Bestehen die ersten beiden Zeichen aus dem ASCII-String »II«, ist das Intel-Format zu verwenden. Sind hingegen die Zeichen »MM« am Dateianfang abgelegt, so muß das Motorola-Format eingesetzt werden. Stehen am Dateianfang andere Zeichen, handelt es sich nicht um eine TIF-Datei.

Ab dem Offset 2 findet sich in der Grafikdatei die Versionsnummer des TIF-Formates. Diese wird, entsprechend dem zu verwendenden Notationsformat, als 16-Bit-Wert abgelegt. Sie belegte also insgesamt zwei Byte. Weit verbreitet ist immer noch Version 4.2. Also findet sich hier meist der Wert 42 (2Ah). Die letzten vier Byte des Dateikopfes enthalten einen Zeiger auf das

erste IFD. Dieses muß von hier aus entsprechend angesprungen werden, um die einzelnen Daten aus der Datei korrekt auslesen. Von dem IFD führt dann ein neuer Zeiger zum nächsten IFD, bis alle IFDs abgearbeitet sind. Entsprechend dem Notationsformat können die vier Byte wieder in eine Speicheradresse als Offset vom Datei-anfang aus umgewandelt werden.

Offset	Länge in Byte	Feld
00	2	Zahl der folgenden Tag-Felder (n)
02	12	Tag 0
0E	12	Tag 1
..	..	..
(n-1)*12+2	12	Tag n
n*12 + 02	4	Zeiger auf das nächste IFD. Wenn kein weiteres IFD vorhanden ist, findet sich hier der Wert 00hex.

Tabelle 11. Aufbau eines IFDs

● Aufbau eines IFD

Wie bereits oben erläutert, können die Daten in der TIF-Datei auf unterschiedliche Stellen verteilt sein. Entsprechend ist die Position der einzelnen IFDs nicht festzulegen. Daher enthält jedes IFD einen Zeiger auf das nachfolgende IFD. So lassen sich alle IFDs problemlos nacheinander abarbeiten.

Ein IFD setzt sich dabei aus unterschiedlichen, sogenannten Tags zusammen. Diese können entweder direkt Daten enthalten,

Offset	Länge in Byte	Feld
00	2	Typus des Tag-Feldes
02	2	Datentyp
04	4	Länge des Datenbereiches
08	4	Zeiger auf den Datenbereich oder der entsprechende Datenwert, wenn kein eigener Datenbereich vorhanden ist.

Tabelle 12. Der Aufbau eines Tags

oder sie verweisen auf eine Speicherstelle, an der die entsprechenden Daten, zum Beispiel die Bildinformationen, abgelegt sind. Am Beginn eines IFD-Blocks steht dabei ein Zähler, der angibt, wie viele Tags das IFD enthält.

Dann folgt die entsprechende Anzahl von Tag-Feldern, die alle eine Länge von 12 Byte besitzen. Die letzten vier Byte hingegen enthalten wiederum den Zeiger auf das nächste IFD. Wenn kein weiteres IFD mehr folgt, findet sich in diesem Feld der Wert 00hex. Somit ergibt sich der in Tabelle 11 beschriebene Aufbau eines IFDs. Die eigentlichen Informationen

über die einzelnen Daten und die Verweise auf die gewünschten Datenbereiche finden sich also innerhalb der einzelnen Tag-Felder eines IFDs. Ein Tag-Feld ist daher immer genau 12 Byte lang und besitzt die in Tabelle 12 beschriebene Struktur.

Das erste Byte gibt an, welche Informationen in dem Tag-Feld abgelegt sind. Tabelle 13 hilft bei der Entschlüsselung. Die

dort angegebenen Tag-Typen sind jedoch lediglich die als publik deklarierten Tags.

Dann werden andere Kennungswerte verwendet. Beim Einlesen einer TIF-Grafik mit zusätzlichen Tag-Typen, die dem lesenden Programm nicht bekannt sind, können diese Tags einfach übergangen

werden. Es werden dann nur die bekannten Tags eingelesen und bearbeitet.

Das zweite Feld eines Tags, das ebenfalls zwei Byte umfaßt, beinhaltet Informationen über den Datentyp (Tabelle 13).

An die Aufschlüsselung des Datentypes schließt sich ein Feld mit Informationen zu der Länge des eventuell vorhandenen Datenblockes außerhalb des Tags an. Wenn die Daten nicht direkt im Tag abgelegt werden, findet sich in den letzten vier Byte des Tags ein Zeiger auf den entsprechenden Datenbereich.

Hier sind dann die einzelnen Informationen abgelegt.

Wenn ein solcher Datenbereich nicht vorhanden ist, sind die Informationen in den letzten vier Byte des Tags enthalten. Meist reichen die vier Byte

jedoch nicht zur Speicherung der gewünschten Informationen. Die Länge dieser ausgelagerten Daten wird in den vier Byte vor dem Zeiger definiert.

Die Längenangabe, die sich hier findet, bezieht sich immer auf den entsprechenden Datentyp und gibt die Anzahl der entsprechenden Datenworte an. Die Information von zehn Datenworten beim

Code	Datentyp	Bedeutung
01	Byte	8 Bit Byte
02	ASCII	8 Bit ASCII-Code
03	SHORT	16 Bit Integerwert ohne Vorzeichen (unsigned)
04	LONG	32 Bit Integerwert ohne Vorzeichen (unsigned)
05	RATIONAL	insgesamt 64 Bit, bestehend aus zwei Daten des Types LONG. Die erste Integerzahl gibt den Zähler der Realzahl an, die zweite Integerzahl den Nenner.

Tabelle 13. Die Decodierung des Datentypes

Datentyp 01 entspricht also zehn Byte. Beim Datentyp rational hingegen entspricht dieselbe Information einer Länge von 640 Byte (10 x 64 Byte).

Eine Besonderheit ist bei ASCII-Strings zu beachten. Diese müssen im TIF-Format nullterminiert sein. Am Ende der Stringkette muß also das Zeichen 00hex stehen. Dieses Schlußbyte ist in der Längenangabe berücksichtigt.

Falls der String eine ungerade Bytezahl umfaßt, kann ein zusätzliches Füllbyte eingefügt werden, das in der Längenangabe nicht berücksichtigt ist. Im Normalfall ergeben sich beim Zurücklesen eines ASCII-Strings jedoch hiermit keine Probleme, weil automatisch die Daten korrekt zurückgelesen oder auch geschrieben werden.

● Die einzelnen Tag-Typen

Im folgenden und in Tabelle 14 finden Sie eine Liste mit der Beschreibung der einzelnen Tag-Typen, die vorkommen können. Die entsprechenden Codes finden sich in den ersten beiden Bytes eines Tag-Feldes. Beachten Sie aber unbedingt die unterschiedlichen Notationsformen im Intel- und Motorola-Format. Für das korrekte Auslesen der Daten müssen die beiden Bytes korrekt interpretiert werden. Sonst entsteht ein Datenchaos.

Hier die einzelnen Tags:

Subfile Tag (FFhex): In diesem Tag werden die globalen Beschreibungen der Bitmap-daten abgelegt. Er ist in der Definition des TIF-Formates zwar nicht zwingend vorgeschrieben, wird dieser Tag jedoch eingesetzt, sind auch die Tags Bildbreite und Bildhöhe sowie Stripoffset zwingend erforderlich. Der Subfile-Tag wird hauptsächlich eingesetzt, wenn in einer TIF-Datei mehrere Grafiken gespeichert werden. Entsprechend Tabelle 15 ist dieser Tag aufgebaut:

Tag Typ = FFhex
Datentyp = SHORT
Länge Datenbereich = 1
Subfile Typ = Zahlenwert im Bereich von 1 bis 3

Der Subfile kann entsprechend der angegebenen Zahl einen der in Tabelle 15 dargelegten Typen einnehmen.

Wenn ein Bild mit reduzierter Auflösung gespeichert wurde, muß ein weiterer Tag enthalten sein, der die volle Auflösung der Grafik definiert. Sobald der Wert 3 im Datenbereich auftritt, liegt eine Definition eines Teilbildes der Komplettegrafik vor. In diesem Fall muß ein Tag mit der Seitennummer vorhanden sein.

Wenn kein Subfile-Tag in einer TIF-Datei vorhanden ist, sollte die volle Auflösung der Grafikkarte als Standard gesetzt werden.

Bildbreite (100hex): Dieser Tag definiert die Breite eines Bildes in Pixeln. Gemäß

Tabelle 15 ergibt sich folgender Aufbau des Tags:

Tag Typ = 100hex
Datentyp = SHORT
Länge Datenbereich = 1
Bildbreite = Zahlenwert im Bereich von 0 bis 65535

Die Bildbreite wird dabei auf der x-Achse abgelegt, sie entspricht also der Ausdehnung des Bildes in horizontaler Richtung. Dieser Tag muß in jeder TIF-Datei enthalten sein. Er kann gleichzeitig als Anzahl der Bildspalten interpretiert werden.

Bildhöhe (101hex): Entsprechend dem vorherigen Tag definiert dieser die Bildhöhe entlang der Y-Achse. Auch dieser Tag ist erforderlich. Er kann als Anzahl der Bildzeilen interpretiert werden.

Tag Typ = 101hex
Datentyp = SHORT
Länge Datenbereich = 1
Bildhöhe = Zahlenwert im Bereich von 0 bis 65535

Bits pro Ebene (102hex): Dieser Tag definiert die Anzahl Bits, die zur Speicherung eines Pixels in einer Speicherebene verwendet werden.

Eine S/W-Grafik enthält normalerweise 1 Bit pro Bildpunkt. Für vier Graustufen werden bereits vier Bit belegt. Bei drei Farben und acht Bit pro Farbinformation sind bereits 24 Bit pro Bildpunkt eventuell erforderlich.

Da die einzelnen Farben in unterschiedlichen Farbenen (Rot, Grün und Blau) abgelegt werden, können für einzelne Farbenen auch unterschiedliche Bitbreiten definiert werden.

Ab Offset 08hex finden sich dabei in den ersten beiden Bytes die Werte für die einzelnen Farbenen in der Reihenfolge Rot, Grün und Blau.

Üblicherweise werden für Graustufenbilder 4 oder 6 Bit definiert, für Farbgrafiken 8 oder 12 Bit.

Tag Typ = 102hex
Datentyp = SHORT
Länge Datenbereich = 1
Bildhöhe = Zahlenwert im Bereich von 1 bis 65535

Kompressionsart (103hex): Dieser Tag definiert eines der Komprimierungsverfahren, die zu den Bilddaten dieses IFDs gehören. Er besitzt folgenden Aufbau:

Tag Typ = 103hex
Datentyp = SHORT
Länge Datenbereich = 1
Bildhöhe = Zahlenwert im Bereich von 1 bis 65535

Findet sich ab Offset 08hex der Wert 1, liegen die Daten unkomprimiert vor. Sie sind lediglich byteweise direkt zusammengefaßt, wie zum Beispiel beim GIF-Verfahren. In einer S/W-Grafik beschreibt ein

Tags zur Bildbeschreibung	
Code (hexadezimal)	Tag-Gruppe
0FF	Subfile
100	Bildbreite
101	Bildhöhe
112	Orientierung
11C	Planar Konfiguration
11A	Auflösung in x-Richtung
11B	Auflösung in y-Richtung
128	Auflösungseinheit (ResolutionUnit)
Tags, die auf die Bilddaten zeigen	
111	StripOffset
116	Zeilen pro Strip
117	StripByte Zähler
Bildbeschreibungs-Tags	
102	Bits pro Ebene
106	Farbinterpretation (Photometric Interpretation)
107	Tresholding
108	Zellbreite (CellWidth)
109	Zellhöhe (CellLength)
115	Ebenen pro Bildpunkt
118	minimaler Ebenenwert
119	maximaler Ebenenwert
122	Graustufeneinheit
123	Graustufenkurve
12C	Farbeinheit
12D	Farbkurve
Tags zur Beschreibung der Datenablage	
10A	FillOrder Tag
Kompressions-Tags	
103	Kompressionsart
124	Optionen Gruppe 3
125	Optionen Gruppe 4
Tags zur Beschreibung der Dokumentation und der Dateneingabe	
10D	Dokumentname
10E	Bildbeschreibung
10F	Scannermake
110	Scanner Modell
11D	Seitenname
11E	x-Position
11F	y-Position
129	Seitennummer
Tags zur Organisation der Datenspeicherung	
120	freie Offsets
121	Zähler für freie Bytes

Tabelle 14. Die einzelnen Tags und ihre Erkennungscodes

Byte also acht Bildpunkte. Das letzte Byte einer Zeile wird gegebenenfalls mit Leerstellen aufgefüllt.

Wenn die Bitbreite größer als acht Bit ist, werden die Informationen in 16- oder 32-Bit-Blöcken abgelegt.

Beachten Sie dabei die Speicherkonventionen für das Motorola- beziehungsweise für das Intel-Format. Bei unkomprimierten Daten müssen zusätzlich die Bitbreite sowie der minimale und maximale Ebenenwert definiert sein.

Die Bitbreite wird dabei immer auf den entsprechenden Wert im Bereich von 2^x definiert. Durch Hinzuziehen der maximalen und minimalen Zahlenwerte innerhalb der Ebene kann die tatsächliche Bitbreite zurückgewonnen werden. Ein Beispiel:

Wert	Bedeutung
1	Bild mit voller Auflösung
2	Bild mit reduzierter Auflösung
3	einseitiges Teilbild

Tabelle 15. Die Bedeutung der Datenbytes im Subfile-Tag

Bei einer Grafik mit einer Bitbreite von 6 Bit ist die Bitbreite auf acht zu setzen. Der minimale Ebenenwert entspricht 0, der maximale 64. Somit werden von den acht lediglich sechs Bit für die Realisierung eines Bildpunktes benötigt.

Der Wert 2 im Datenbereich definiert, daß die Grafikinformatoren nach dem CCITT/3-ID-Verfahren komprimiert wurden. Dabei werden abweichend von diesem Komprimierungsstandard jedoch keine Endzeichen benutzt. Auch wird jede Bildzeile unabhängig von den anderen Zeilen komprimiert und gespeichert.

Die Komprimierungsverfahren 3 und 4 werden im FAX-Mode eingesetzt. Sie entsprechen dem FAX CCITT Group 3 beziehungsweise dem FAX CCITT Group 4-Komprimierungsverfahren. Die StripOffset-Tags dienen beim CCITT-Komprimierungsverfahren der Optimierung der Kompressionsrate.

Die Zahlenwerte im Bereich von 32 768 bis 65 535 sind für herstellereinspezifische Komprimierungsverfahren vorgesehen. Bisher wurden zwei dieser Verfahren als öffentlich deklariert:

Mit dem Wert 32 771 wird der Speichertyp 1 ohne Komprimierung eingesetzt. Dafür beginnt jede Zeile auf der nächstliegenden freien Wortgrenze. Der Zahlenwert 32 773 benutzt die Packbit-Komprimierung.

Hinweise zu den Komprimierungsverfahren finden Sie im Anschluß an die Beschreibung der Tag-Typen.

Farbinterpretation (106hex): Jedes TIF-Bild enthält eine bestimmte Anzahl von Bilddaten.

Der Wert 106hex definiert nun zusammen mit den maximalen und minimalen Ebenenwerten die korrekte Interpretation der Bilddaten. Der Tag ist aufgebaut:

Tag Typ = 106hex
Datentyp = SHORT
Länge Datenbereich = 1
Bildhöhe = Zahlenwert im Bereich von 0 bis 2

Die Zahlenwerte im Wertebereich bezeichnen Farben. 0 bedeutet: Der Zahlenwert, der als minimaler Ebenenwert definiert wurde, ist als weiße Farbe zu betrachten. Der maximale Ebenenwert entspricht Schwarz. Bei Graustufenbildern sind den entsprechenden anderen Zahlen die einzelnen Graustufen zuzuordnen. 1 bedeutet: Hier wird genau umgekehrt wie bei 0 vorgegangen. 2 bedeutet: Die Daten des Bildbereiches sind im RGB-Verfahren codiert. Der minimale Ebenenwert entspricht der Intensität 0% der entsprechenden Farbebene, der maximale Ebenenwert der Intensität 100%.

Wenn die Bilddaten nach dem CCITT/3-Verfahren codiert sind, darf dieser Tag nicht verwendet werden. In diesem Fall ist durch die Komprimierungsart das Speicherverfahren der Bilddaten bereits definiert.

Thresholding Tag (107hex): Dieser Tag definiert bei nachbearbeiteten Bildern das Bearbeitungsverfahren an. Eine Grafik mit einem solchen Tag sollte nicht mehr skaliert werden, da sonst die Bilddaten grundlegend verändert würden.

Tag Typ = 107hex
Datentyp = SHORT
Länge Datenbereich = 1
Bildhöhe = Zahlenwert im Bereich von 1 bis 3

Die Werte sind folgendermaßen zu deuten: 1 bedeutet: Ein Schwarzweißbild (Bilevel) wurde gespeichert. Die Zahl der Bits pro Pixel ist auf eins zu setzen. 2 bedeutet: Ein Graustufenbild wurde in ein S/W-Bild transformiert (Dithering). Die Bits pro Pixel entsprechen wieder dem Wert eins. 3 bedeutet: Die Error-Diffusions-Methode wurde angewendet, um ein Graustufenbild in ein S/W-Bild zu konvertieren.

Zellbreite (108hex): Dieser Tag kann auftreten, wenn ein Thresholding-Tag den Wert 2 enthält. In diesem Fall wird die Breite der Matrix beim Dithering definiert.

Tag Typ = 108hex
Datentyp = SHORT
Länge Datenbereich = 1
Matrixbreite = Zahlenwert im Bereich von 1 bis 3

Zellhöhe (109hex): Wie zuvor wird dieser Wert beim Dithering zur Aufnahme der Matrixhöhe eingesetzt.

Tag Typ = 109hex
Datentyp = SHORT
Länge Datenbereich = 1
Matrixbreite = Zahlenwert im Bereich von 1 bis 3

Fillorder Tag (10Ahex): Dieser Tag definiert die Speicherungsmethode der Bilddaten in den einzelnen Bytes. Er besitzt folgenden Aufbau:

Tag Typ = 10Ahex
Datentyp = SHORT
Länge Datenbereich = 1
Matrixhöhe = Zahlenwert im Bereich von 1 bis 2

Dabei können die beiden folgenden Werte auftreten:

1 bedeutet: Die Bilddaten werden in den einzelnen Bytes fortlaufend abgelegt. Der erste Bildpunkt beginnt dabei im Bit 7 des höherwertigen Bytes. Die folgenden Bildpunkte werden in den niederwertigen Bits abgelegt. Dieses Verfahren entspricht dem Standard. 2 bedeutet: Wie oben werden die Daten in einem Byte fortlaufend gespeichert. Der erste Punkt beginnt jedoch rechts im niederwertigen Bit. Die folgenden Punkte sind in dem höherwertigen Bit untergebracht.

Dokumentname (10Dhex): Dieser Tag definiert den Namen des Dokumentes, von dem das Bild erzeugt wurde. Der eigentliche Datenwert steht dabei irgendwo innerhalb der Datei als nullterminierter ASCII-String. Der Tag enthält einen Zeiger auf diesen Text als Offset vom Dateianfang.

Tag Typ = 10Dhex
Datentyp = ASCII
Länge Datenbereich = n
Zeiger auf den Text als Offset vom Dateianfang

Bildbeschreibung (10Ehex): Dieser Tag kann unterschiedliche Beschreibungen zur Grafik enthalten. Auch hier wird auf einen nullterminierten ASCII-String durch einen Zeiger als Offset vom Dateianfang verwiesen. Hier können unterschiedliche Zusatzinformationen abgelegt werden.

Tag Typ = 10Ehex
Datentyp = ASCII
Länge Datenbereich = n
Zeiger auf den Text als Offset vom Dateianfang

Scannermake Tag (10Fhex): Mit diesem Tag wird auf den Namen eines Scannerherstellers verwiesen, der als nullterminierter ASCII-String abgelegt wurde.

Wert	1. Bitmap-Reihe	1. Spalte
1	obere Kante	linke Kante
2	obere Kante	rechte Kante
3	untere Kante	linke Kante
4	untere Kante	rechte Kante
5	linke Kante	obere Kante
6	rechte Kante	obere Kante
7	linke Kante	untere Kante
8	rechte Kante	untere Kante

Tabelle 16. Die einzelnen Orientierungsvarianten

Tag Typ = 10Fhex
Datentyp = ASCII
Länge Datenbereich = n
Zeiger auf den Text als Offset vom Dateianfang

Scanner-Modell-Tag (110hex): Dieser Tag spezifiziert Scannermodell und Nummer in einem nullterminierten ASCII-String.

Tag Typ = 110hex
Datentyp = ASCII
Länge Datenbereich = n
Zeiger auf den Text als Offset vom Dateianfang

Stripoffset-Tag (111hex): Dieser Tag enthält einen Zeiger auf das Bitmap mit den Bilddaten. Er enthält immer einen 4-Byte-Zeiger als Offset vom Dateianfang. Dieser verweist auf den Beginn des Datenfeldes mit den Bilddaten. Zur Lokalisierung der Bilddaten muß mindestens ein Stripoffset-Tag vorhanden sein.

Es kann jedoch auch der Fall sein, daß der Zeiger im Stripoffset-Tag auf eine Tabelle mit den eigentlichen Zeigern auf den Datenanfang verweist. Diese Tabelle enthält dann für jede einzelne Bildzeile einen solchen Zeiger.

Der Unterschied, ob auf eine Tabelle oder den tatsächlichen Datenanfang verwiesen wird, wird im Wert des Längenfeldes vorgegeben.

Liegt hier der Wert 1 vor, so handelt es sich um einen Zeiger auf den tatsächlichen Bildanfang. Bei einem Wert größer als 1 wird auf eine Zeigertabelle hingewiesen. In dieser Tabelle finden sich dann n Einträge mit den entsprechenden Zeigern. Die Zahl n ist dem Längenfeld des Tags zu entnehmen.

Normalerweise wird der Wert mit dem Datentyp LONG verbunden. Nur in Ausnahmefällen handelt es sich um den Datentyp SHORT.

Tag Typ = 111hex
Datentyp = SHORT oder LONG
Länge Datenbereich = n
Zeiger auf den Bildanfang oder auf eine entsprechende Zeigertabelle als Offset vom Dateianfang

Orientierung (112hex): Dieser Wert definiert die Orientierung des Bildes in x- und y-Richtung, wobei acht unterschiedliche Werte auftreten können. Die Decodierung entnehmen Sie Tabelle 16. Der Standardwert entspricht 1. Beim Verändern dieses Zahlenwertes rotiert das Bild um den Ursprung.

Tag Typ = 112hex
Datentyp = SHORT
Länge Datenbereich = 1
Orientierung = Zahlenwert im Bereich von 1 bis 8

Ebenen pro Bildpunkt (115hex): Dieser Wert gibt die Anzahl der Ebenen, in der ein einzelner Bildpunkt abgelegt wird. Bei S/W-Grafiken findet sich der Wert 1, bei Farbgrafiken der Wert 3. Graustufenbilder mit vier oder sechs Graustufen enthalten ebenfalls nur eine Farbebene.

Tag Typ = 115hex
Datentyp = SHORT
Länge Datenbereich = 1
Orientierung = Zahlenwert im Bereich von 1 bis 65535

Zeilen pro Ebene (116hex): Dieser Tag definiert die Zeilen eines unkomprimierten Teilbildes (Strips). Alle einzelnen Teilbilder mit Ausnahme des letzten Teilbildes müssen dabei dieselbe Zeilenzahl aufweisen. Bei komprimierten Bilddaten entspricht dieser Wert nicht der tatsächlichen Zeilenzahl.

Tag Typ = 116hex
Datentyp = SHORT oder LONG
Länge Datenbereich = 1
Zeilen pro Ebene = Zahlenwert im Bereich von 0 bis $(2^{32}-1)$

Strip-Byte-Zähler (117hex): Dieser Tag wird bei der Datenkomprimierung des CCITT/3-Verfahrens genutzt. Mit diesem Wert kann geprüft werden, ob die korrekte Anzahl an Daten gelesen wurde. Der Tag wird für jeden komprimierten Datensatz abgelegt, auf den ein Stripoffset-Zeiger verweist.

Tag Typ = 117hex
Datentyp = LONG
Länge Datenbereich = n
Anzahl der Bytes in einem Teilbild = Zahlenwert im Bereich von 0 bis $(2^{32}-1)$

Minimaler Ebenenwert (118hex): In diesem Tag wird der kleinstmögliche Wert innerhalb der interpretierten Ebene abgelegt. Er dient zur korrekten Interpretation der Farbwerte. Normalerweise wird für die richtige Farinterpretation hier der Wert 0 eingetragen.

Tag Typ = 117hex
Datentyp = SHORT
Länge Datenbereich = 1
Minimaler Ebenenwert = Zahlenwert im Bereich von 0 bis 65535

Maximaler Ebenenwert (119hex): Dieser Tag gibt die größtmögliche Zahl an, die als Bildinformation in einer Ebene auftreten kann. Bei S/W-Grafiken wird dieser Wert auf 1 gesetzt. Der Wert gibt die maximale Farbzahl wieder, wird als Standard jedoch auch auf 65535 gesetzt.

Tag Typ = 119hex
Datentyp = SHORT
Länge Datenbereich = 1
Maximaler Ebenenwert = Zahlenwert im Bereich von 0 bis 65535

Auflösung in X-Richtung (11Ahex): Dieser Tag definiert die Auflösung des Bildes in x-Richtung. Er ist für Scanner wichtig, die in unterschiedlichen Auflösungen arbeiten. Somit kann die Skalierung den Gerätefähigkeiten angepaßt werden. Normalerweise werden Auflösungen von 75 bis 300 dpi unterstützt. Im Wertefeld wird eine Auflösung von 300 dpi mit 300/1 definiert. Die Auflösung von 75 dpi wird als 150/2 realisiert. Das Feld liegt innerhalb der Datei, im Tag verweist nur ein Zeiger als Offset vom Dateianfang auf dieses Feld.

Tag Typ = 11Ahex
Datentyp = RATIONAL
Länge Datenbereich = 1
Maximaler Ebenenwert = Wert 1 / $[1..(2^{32}-1)]$

Auflösung in Y-Richtung (11Bhex): Dieser Wert entspricht der Auflösung in x-Richtung, gibt jedoch die vertikale Auflösung entlang der y-Achse an.

Tag Typ = 11Bhex
Datentyp = RATIONAL
Länge Datenbereich = 1
Maximaler Ebenenwert = Wert 1 / $[1..(2^{32}-1)]$

Planar Konfiguration (11Chex): Dieser Tag wird nur bei Farb- und Graustufengrafiken eingesetzt. Er beschreibt die Organisation der Bilddaten.

Tag Typ = 11Chex
Datentyp = SHORT
Länge Datenbereich = 1
Konfiguration = Wert im Zahlenbereich von 1 bis 2

Die beiden folgenden Konfigurationen sind realisierbar: 1 = Alle Informationen sind zusammenhängend in einem Bild-

punkt abgelegt, die Farbenbits stehen direkt hintereinander. 2 = Die Informationen über einen Bildpunkt sind auf mehrere Farbebenen verteilt. Hier werden erst alle Bildpunkte einer Ebene beschrieben. Anschließend wird die nächste Ebene definiert.

Seitenname (11Dhex): Dieser Tag definiert den Seitennamen der Seite, von dem das Bild gescannt wurde.

Tag Typ = 11Dhex
Datentyp = ASCII
Länge Datenbereich = n
Zeiger auf den Seitennamen als Offset vom Dateianfang

X-Position (11Ehex): Dieser Tag definiert die linke obere Ecke des Bildes in Zoll. Er gibt dabei die x-Koordinate als Rational-Wert an, auf den durch einen Zeiger verwiesen wird.

Tag Typ = 11Ehex
Datentyp = RATIONAL
Länge Datenbereich = 1
Zeiger auf den Wert der X-Position als Offset vom Dateianfang

Y-Position (11Fhex): Dieser Wert wird wie oben abgelegt. Er enthält die y-Koordinate der linken oberen Bildecke in Zoll. Im TIF-Format ist die positive y-Achse dabei nach unten gerichtet.

Tag Typ = 11Fhex
Datentyp = RATIONAL
Länge Datenbereich = 1
Zeiger auf den Wert der Y-Position als Offset vom Dateianfang

Freie Offsets (120hex): Dieser Tag beinhaltet einen Zeiger, der auf eine Tabelle mit weiteren Zeigern verweist. Diese Zeiger definieren freie Bereiche innerhalb der TIF-Datei. Die Zahl der Zeiger ist dem Längenfeld zu entnehmen.

Tag Typ = 120hex
Datentyp = LONG
Länge Datenbereich = n
Zeiger auf eine Tabelle mit Verweisen auf freie Speicherstellen

Zähler für freie Bytes (121hex): Damit immer die Anzahl der freien Bytes bekannt ist, wird diesem Zähler die Anzahl der jeweils freien Bytes innerhalb eines Blockes definiert.

Der Tag enthält einen Zeiger auf eine Datentabelle. Diese umfaßt n Byte Einträge, die die Länge des zugehörigen freien Bereiches definieren. Die Lage dieser Bereiche wird über den freien Offset (120hex) definiert. Die Zahl n entspricht dem Eintrag im Längenfeld.

Tag Typ = 121hex
Datentyp = LONG
Länge Datenbereich = n
Zeiger auf eine Datentabelle

Graustufeneinheit (122hex): Dieser Wert erlaubt zusammen mit der Graustufenkurve die korrekte Interpretation der Bilddaten.

Tag Typ = 122hex
Datentyp = SHORT
Länge Datenbereich = 1
Graustufeneinheit = Wert im Zahlenbereich von 1 bis 5

Die einzelnen Zahlenwerte definieren die Wertigkeit der Graustufenkurve:

1 = 1/10
 2 = 1/100
 3 = 1/1000
 4 = 1/10000
 5 = 1/100000

Die Graustufe ergibt sich aus Graustufeneinheit und -kurve. Liegt hier der Wert 243 vor, während die Graustufeneinheit den Wert 4 enthält, so entspricht dies einer Graustufe von 0,0243.

Graustufenkurve (123hex): Dieser Tag definiert zusammen mit der Graustufeneinheit die korrekte Bilddateninterpretation. Er enthält einen Vektor auf die eigentlichen Daten. Für jede Ebene ist hier ein Eintrag vorhanden, der als Graustufe für die Interpretation der Ebene dient. Die Stufen geben die Schwellen zur Umrechnung der einzelnen Grautöne innerhalb des minimalen und maximalen Ebenenwertes an.

Tag Typ = 123hex
Datentyp = SHORT
Länge Datenbereich = 2^{Bits} pro Ebene
Graustufenkurve = Wert im Zahlenbereich zwischen 0 und 65535
Graustufeneinheit = Wert im Zahlenbereich von 1 bis 5

Optionen Gruppe 3 (124hex): Dieser Tag wird bei einer Komprimierung der Bilddaten nach dem CCITT/3-Verfahren eingesetzt. Hiermit werden einige Optionen definiert.

Tag Typ = 124hex
Datentyp = LONG
Länge Datenbereich = 1
Gruppe 3 Optionen = 32 Bit Flag

Die Interpretation der vier Datenbytes wird als 32-Bit-Flag vorgenommen. Alle unbenutzten Bits werden auf den Wert 0 gesetzt. Die Bits 0 bis 2 haben eine Bedeutung, wenn sie gesetzt sind. 0: Zweidimensionale Codierung (eindimensionale Co-

dierung, wenn nicht gesetzt). 1: Unkomprimierter Modus wird eingesetzt. 2: Füllbits am Ende der Zeile wurden eingefügt.

Bei einer zweidimensionalen Codierung der Daten im Datenbereich ist die erste Zeile immer eindimensional zu codieren.

Gruppe 4 Optionen (125hex): Bei einer Codierung nach dem CCITT/4-Verfahren werden mit diesem Tag wie beim oberen einige Optionen definiert.

Tag Typ = 125hex
Datentyp = LONG
Länge Datenbereich = 1
Gruppe 3 Optionen = 32 Bit Flag

Hierbei wird jedoch ausschließlich Bit 1 zur Anzeige eines unkomprimierten Modus eingesetzt.

Auflösungseinheit (128hex): Mit diesem Tag wird die Einheit definiert, die von den Tags Auflösung in x- und y-Richtung eingesetzt werden.

Tag Typ = 128hex
Datentyp = SHORT
Länge Datenbereich = 1
Auflösungseinheit = Wert im Zahlenbereich zwischen 1 und 3

Die einzelnen Zahlenwerte bedeuten:
 1: Keine Informationen über die Auflösung
 2: Auflösung in Zoll
 3: Auflösung in Zentimetern

Bei 0 werden die Angaben hinsichtlich der Bildschirmauflösungen lediglich zur Definition des Bildverhältnisses (Aspect Ratio) von x-Achse zu y-Achse eingesetzt.

Seitennummer (129hex): Dieser Tag gibt die Bildseitennummer bei TIF-Dateien mit mehreren Bildern an. Der Wert ist besonders für FAX-Übertragungen wichtig. Das Wertefeld enthält zwei 16-Bit-Werte. Der erste gibt die aktuelle Seitennummer an, der zweite die Zahl der Seiten innerhalb der Datei. Die Reihenfolge der Seiten innerhalb der Datei ist beliebig.

Tag Typ = 129hex
Datentyp = SHORT
Länge Datenbereich = 2
Seitennummer / Seitenanzahl = erster Zahlenwert / zweiter Zahlenwert

Farbeinheit (12Chex): Dieser Wert ist wie der Wert Graustufeneinheit aufgebaut und wird ebenso interpretiert.

Farbkurve (12Dhex): Wie bei der Graustufenkurve gibt dieser Wert zusammen mit der Farbeinheit die korrekte Interpretation der Bilddaten an. Die Farbkurve enthält dabei Werte zwischen 0 und 6553. Der Tag hat folgenden Aufbau:

Tag Typ = 12Dhex
Datentyp = SHORT
Länge Datenbereich = Bits pro Ebene * 3
Farbkurve = Wert aus dem Zahlenbereich zwischen 0 und 6553

Der Tag definiert wiederum einen Vektor auf die eigentlichen Daten. Für jede Ebene findet sich hier ein Tripel von drei Einträgen. Diese geben die Intensitäten von Rot, Grün und Blau wieder. Die Zahl der Tabelleneinträge ergibt sich entsprechend aus dem Wert:

Bits pro Ebene²

● Kompressionsverfahren des TIF-Formates

Beim TIF-Format existieren unterschiedliche Kompressionsverfahren. Eine Packbit-Komprimierung kann vorgenommen werden. Dieses Verfahren wird durch die Kennzahl 32 773 im entsprechenden Kompressions-Tag gekennzeichnet.

Bei diesem Kompressionsverfahren werden die Bilddaten in Records gespeichert. Das erste Byte eines Records ist der Header, er gibt die Zahl der nachfolgenden Bytes an. In Abhängigkeit von dem Wert des Headers gibt es zwei verschiedene Recordarten.

Wenn der Header mit den Bytes 00hex – 7Fhex anfängt, besteht der Record aus n+1 Datenbytes zusätzlich zum Header. Der Wert n steht im Header. In diesem Fall werden die ersten n Bytes der Eingabedatei ausgelesen und zur Ausgabe geschickt. Der Record enthält unkomprimierte Daten. Wenn der Header hingegen die Bytes 81hex – FFhex aufweist, besteht der Record aus dem Header und einem Datenbyte. Vom Wert des Headers ist 80hex zu subtrahieren. Das Datenbyte wird mit der sich ergebenden Wiederholung an die Ausgabe geschickt.

● FAX-Komprimierung

Ein zweites Kompressionsverfahren wird mit der FAX-Komprimierung nach dem CCITT-Standard angeboten. Hierbei werden zusammenhängende Gruppen aus schwarzen und weißen Bildpunkten codiert. Dabei werden, ähnlich wie bei der GIF-LZW-Komprimierung, zwei Tabellen eingerichtet. Die erste enthält an den ersten 64 Stellen zwischen 0 und 63 die entsprechenden Wiederholungen ausschließlich weißer Bildpunkte. Die zweite entsprechend die Wiederholung schwarzer Bildpunkte. Außerdem können weitere, diskrete Kombinationen aus weißen und schwarzen Bildpunkten in der Tabelle oberhalb der ersten 64 Einträge aufgenommen werden. Bei der Abtastung einer Zeile wird nun einer entsprechenden Punktfolge der zugehörige Tabellenwert zugewiesen. Das TIF-Kompressionsformat unterliegt dabei folgenden Auflagen:

– Die Zeilen der Bitmap werden unabhängig voneinander eingelesen und verschlüsselt.

Offset	Länge in Byte	Bedeutung
00	2	Kennzeichnung der Datei »BM« für Bitmap
02	4	Gesamtgröße der Datei in Bytes
06	2	Reserviert: In der momentanen Version wird hier der Wert 00hex abgelegt.
08	2	Reserviert: Auch hier findet sich der Wert 00hex.
0A	4	Startadresse der eigentlichen Bitmap-Daten als Offset vom Dateianfang in Bytes
0E	4	Größe der Datenstruktur in Bytes
12	4	Breite der Bitmap in Bildschirmpunkten
16	4	Höhe der Bitmap in Bildschirmpunkten
1A	2	Anzahl der Farbebenen, wird in der aktuellen Version auf 1 gesetzt
1C	2	Bits pro Pixel (1,4,8 oder 24)
1E	4	Kennziffer für die Komprimierung (keine Komprimierung = 0)
22	4	Größe des Bitmaps in Bytes
26	4	horizontale Auflösung
2A	4	vertikale Auflösung
2E	4	Anzahl der Farben oder 00hex
32	4	Anzahl »wichtiger« Farben

Tabelle 17. Der Dateikopf einer BMP-Datei

– Jede codierte Zeile hat mit einem weißen Bitmuster zu beginnen. Bei Bildern, die mit einem schwarzen Pixel beginnen, wird der Code für eine Folge von 0 weißen Pixeln zunächst eingeschoben.

– Die Version 3 des CCITT-Standards kennzeichnet ein Zeilenende mit einem EOF-Zeichen. Dies entfällt in der TIF-Datei.

– Füllbits am Bildrand werden nicht codiert, da sie weder weißen noch schwarzen Punkten zuzuordnen sind. Der genauere Aufbau der Tabellen beim CCITT/3-Verfahren kann dem entsprechenden ISO-Dokument entnommen werden.

Das BMP-Format

Beim BMP-Format handelt es sich ebenfalls um eine geräteunabhängige Bitmap (Device Independent Bitmap; DIP). Die Bildpunkte werden unabhängig vom Ausgabegerät zeilenweise in einzelnen Ebenen organisiert dargestellt. Auch lassen sich dabei Komprimierungsverfahren einsetzen.

Der Dateikopf

Die Windows-Bitmap-Datei besitzt ab Windows 3.0 einen einheitlichen Dateikopf. Die Struktur gliedert sich auf, wie in Tabelle 17 dargelegt.

Am Anfang der Datei findet sich immer die zweistellige Kennung »BM«. Ist sie nicht vorhanden, liegt keine BMP-Datei vor. Anschließend wird in insgesamt vier Bytes die gesamte Größe der Grafikdatei angegeben. Dabei wird das Intel-Notationsverfahren wie oben beschrieben eingesetzt.

Nach vier reservierten Bytes, die mit dem Wert 00hex belegt sind, folgt schließlich ein Zeiger auf den Anfang der eigentlichen Bilddaten. Ab der angegebenen Stelle finden sich also die einzelnen Informationen bezüglich der einzelnen Bildpunkte.

In einem weiteren, ebenfalls vier Byte

wird schließlich definiert, wie viele Bits zur Festlegung eines Pixels verwendet werden. Hieraus ergibt sich die Farbzahl. Bei dem Wert 1 handelt es sich um eine zweifarbige Grafik. Der Wert 4 deutet auf 16 Farben, der Wert 8 auf 256 und der Wert 24 auf über 16 Millionen Farben hin, die in einer Grafik verwendet werden können.

Die Auswertung der Farbzahl ist aber nur dann erforderlich, wenn bei der Angabe der verwendeten Farbzahl der Wert 0 eingetragen worden ist.

Die folgenden Felder können mit dem Standardwert 00hex belegt werden. Sie können aber auch entsprechende Informationen bezüglich der Grafik aufnehmen. Das erste Feld weist auf ein Komprimierungsverfahren hin. Standardmäßig findet sich hier der Wert 0, der keine besondere Komprimierung andeutet. Die Grafikdaten werden lediglich bitweise entsprechend der Bit-Tiefe abgelegt.

Außerdem kann die Bitmap-Größe in Bytes definiert werden. Dieser Wert wird bei einer Komprimierung verwendet, um die korrekte Bitmap-Größe zurückzugewinnen zu können. Diese ist von der Datengröße verschieden, die zuvor bereits abgelegt wurde.

Zudem kann die horizontale und vertikale Auflösung des Ausgabegerätes angegeben werden. Wie beim TIF-Format sind diese Werte vor allem für Scanner von Interesse. Hier wird die entsprechende Auflösung in Pixeln pro Meter definiert. Im folgenden Feld wird schließlich die Anzahl der verwendeten Farben angegeben. Ist hier der Wert 0 eingetragen, muß wie oben beschrieben die Farbzahl berechnet werden. Ansonsten gilt dieser Wert.

Im letzten Feld des Dateikopfes findet sich schließlich die Anzahl der wirklich »wichtigen« Farbwerte. Dieser Zahlenwert hat jedoch an sich keine besondere Bedeutung. Auch hier kann 00hex als Standardwert eingetragen werden.

großen Feld wird die Länge dieser Bitmap-Daten in Bytes angegeben. Schließlich folgen Breite und Höhe der Grafik, ebenfalls als 4-Byte-Feld, welches nach dem Intel-Format zu interpretieren ist. Die Anzahl der Farbebenen innerhalb der Grafik wird in der aktuellen Windows-Version noch konstant auf den Wert 1 gesetzt. Die einzelnen Farbinformationen werden also in einer Ebene (Plane) abgelegt.

In einem zwei Byte umfassenden Feld

Die Farbpalette

Wenn die Bitmap-Daten eine Farbtiefe von weniger als 24 Bit aufweisen, wenn die Grafik also über weniger als insgesamt 16 Millionen Farben verfügt, dann folgt im Anschluß an den Dateikopf die Ablage der Farbpalette mit den absoluten Farbdefinitionen für die Grafik. Die einzelnen Farbwerte werden dabei in jeweils vier Byte abgelegt. Für die Definition einer Farbe werden also vier Byte verwendet. Das erste Byte gibt den Farbanteil der Farbe Blau, das zweite den der Farbe Grün und das Dritte den Anteil der Farbe Rot an. Das vierte Feld ist momentan noch reserviert und weist den Wert 00hex auf. Die Größe der Farbpalette ergibt sich wie oben beschrieben aus der Bit-Tiefe der Grafikinformationen.

Wenn im vorletzten Feld des Dateikopfes die tatsächliche Anzahl an Farben angegeben ist, so gilt natürlich dieser Wert für die Länge der Farbinformationsdaten. Mit 4 multipliziert berechnet sich dann die Länge der Farbpalette in Bytes.

Die Rasterdaten

Auf die Farbdefinition folgen die Rasterdaten. Die Position ist als Zeiger im Dateikopf der BMP-Datei vermerkt. Hier sind die einzelnen Farbinformationen für einen Bildpunkt zeilenweise abgelegt. Jede einzelne Grafikzeile wird also einzeln gespeichert.

Ein einzelner Bildpunkt belegt dann je nach der im Dateikopf angegebenen Bit-Tiefe 1, 4, 8 oder 24 Bit.

Anders als bei den bisherigen Grafikformaten wird die Grafik jedoch zeilenweise nicht von oben nach unten, sondern umgekehrt von unten nach oben gespeichert. Die erste Zeile an Rasterinformationen bezieht sich also auf die unterste Bildschirmzeile der Grafik.

Die einzelnen Bitmap-Informationen werden dabei direkt aneinandergehängt. Bei einer S/W-Grafik werden in einem Byte also acht Bildschirmpunkte wiedergegeben. Das höchstwertige Bit des Bytes entspricht dabei dem ersten Pixel. Der Wert 0 würde einem gelöschten, der Wert 1 einem gesetzten Bildpunkt entsprechen.

Entsprechend wird bei einer Bit-Tiefe von 4 Bit verfahren. Hier werden zwei Pixel in einem Byte abgelegt. Das erste Pixel belegt dabei die höherwertigen Bits 7 bis 4. Die korrekte Bildschirmfarbe wird durch den Verweis auf den entsprechenden Eintrag der Farbtabelle zurückgewonnen. Bei 256farbigen Grafiken entspricht ein Byte genau einem Bildpunkt.

Jede einzelne Grafikzeile wird für sich gespeichert. Sie muß dabei eine gerade und durch 4 teilbare Anzahl an Bytes aufweisen, da im BMP-Format doppelwortweise gearbeitet wird. Notfalls sind am Ende jeder Zeile Bits mit dem Wert 0 zu ergänzen. Nun wünschen wir Ihnen viel Spaß beim hoffentlich erfolgreichen Umsetzen der vielen Informationen

(Jörg Rensmann/Michael Starke/uk)

3D-Objekte im Raum

Das hier vorgestellte Objekt kann dreidimensionale Körper im Raum bewegen. Die Körper können Sie rotieren, skalieren oder verschieben. Es eignet sich daher für Grafikanimationen und Simulationen. Außerdem können Sie abgeleitete Objekte in Bildschirm-schonern, Animationsprogrammen, Simulatoren für Mehrkörperprobleme oder auch Windows-Spielen einsetzen.

Das Objekt trägt den Namen *t3DFigure*. Es ist in Pascal für Windows als abstraktes Objekt implementiert. Daraus ergibt sich, daß eine direkte Instanziierung nicht sinnvoll ist. Vielmehr leiten Sie die eigentlichen Nutzobjekte vom Basisobjekt ab und überschrei-

ben die entsprechenden abstrakten Darstellungsmethoden. Das Objekt *t3DFigure* befindet sich in der Unit »u3dfgr.pas« (Listing 1).

Dort sind zusätzlich weitere Konstanten zu Darstellungsmodi und zur Initialisierung des Objektes enthalten.

Das Objekt *t3DFigure*

Das Objekt *t3DFigure* stellt Transformationsalgorithmen für die Bewegung dreidimensionaler Körper im Raum zur Verfügung. Es werden eine Reihe von Konstanten vordefiniert, die den Darstellungsmodus von abgeleiteten Objekten festlegen. Das Objekt verwaltet die folgenden Variablen:

FigurePoints ist eine *tCollection* von Objekten des Typs *tFigurePoint*, welche die Informationen über den aktuellen und originalen Wert der x-, y- und z-Koordinate eines einzelnen Punktes einer Figur speichern.

DrawDC ist ein Handle eines DC (Device Context = Gerätekontext), in den die Figur bei der Darstellung gezeichnet wird. Dieser Kontext muß bei Aufruf des Objektkonstruktors übergeben werden.

OriginFigure ist der Ursprung des lokalen Körperkoordinatensystems.

AxisAngles sind die Rotationswinkel der Achsen des Körperkoordinatensystems.

ZoomFactor ist der aktuelle Vergrößerungsfaktor.

RotMatrix ist eine von der Methode *CalcRotMatrix* berechnete Transformationsmatrix für die 3D-Rotationen in Abhängigkeit von *AxisAngles*.

AnimaParams sind Parameter für eine automatische inkrementelle Animationsmethode. Hier sind Winkel-, Verschiebungs- und Skalierungskrement der Figur festgelegt.

MaxX, MaxY, MaxZ sind die Grenzen des Bewegungs-bereichs der Figur, die von der Methode *Bounce* genutzt werden, wenn als Zeichnungsstil *ds_bouncing* festgelegt ist.

DrawWnd ist das Handle des Zeichnungsfensters.

Die Methoden

Init(Wnd:hWnd;DC:hDC): initialisiert das Objekt und legt die Animationsparameter fest. Parameter: *Wnd* ist das Handle des Zeichnungsfensters, *DC* ist ein Handle auf einen gültigen Device Context.

Done: zerstört das Objekt und eventuell in Nachfolgern erzeugte dynamische Variablen bei Umdefinition

MoveFgr(P:tTriple;M:word): bewegt die Figur zum Punkt P. M bestimmt den Darstellungsmodus. Parameter: P enthält die x-, y- und z-Koordinate des Zielpunktes. M ist der Darstellungsmodus.

RotaFgr(A:tTriple;M:word): dreht die Figur um den Winkel A. M bestimmt den Darstellungsmodus. Parameter: A enthält den Rotationswinkel um die x-, y- und z-Achse des Körperkoordinatensystems. M ist der Darstellungsmodus.

ZoomFgr(S:real;M:word): vergrößert die Figur um S. M bestimmt den Darstellungsmodus. Parameter: S ist der

Skalierungsfaktor. M ist der Darstellungsmodus.

Animate: bewegt die Figur einmal um das in *AnimaParams* festgelegte Inkrement. Der Darstellungsmodus ist in *AnimaParams* enthalten.

ChangeAnimaParams(P:tAnimaParams): ändert die Animationsstandards. Parameter: P ist der neue Animationsstandard.

GetAnimaParams(var P:tAnimaParams): liefert die aktuellen Animationsparameter. Parameter: P enthält die aktuellen Animationsparameter.

AddPoint(P:tTriple): fügt einen neuen Punkt der Figur in die *FigurePoints* ein. Parameter: P enthält x-, y- und z-Koordinate des neuen Punktes.

Appear: abstrakte Darstellungsmethode. Diese müssen Sie überschreiben. Die Methode wird von *ZoomFgr*, *MoveFgr*, *RotaFgr* und *Animate* aufgerufen, wenn der Darstellungsmodus *ds_appear* gewählt ist. Hier können Sie entsprechende Algorithmen zum Zeichnen von Linien, welche die Punkte aus *FigurePoints* verbinden, oder zum Ausfüllen von Regionen eintragen, die für das spezielle Nachfolgeobjekt gültig sind.

Disappear: abstrakte Methode, die Sie überschreiben. Diese Methode wird von den Bewegungsmethoden aufgerufen, wenn der Darstellungsmodus *ds_RemoveOriginal* gewählt ist. Sie sorgt für ein Entfernen des nach einer Bewegung nicht mehr aktuellen Objektes.

Bounce: sorgt für die Reflexion des Objektes an den Grenzen *MaxX*, *MaxY* und *MaxZ*. Kann beim Vererben entsprechend überschrieben werden. Die Methode wird aktiviert, wenn der Darstellungsmodus *ds_bouncing* gewählt wurde.

DrawLine(N:word): zeichnet eine Linie vom aktuellen Standpunkt (wird mit *SetStart* festgelegt) zum Punkt N der Figur in die xy-Projektionsebene. Parameter: N muß einem Index aus *FigurePoints* entsprechen.

SetStart(N:word): setzt den Startpunkt für das Linienzeichnen auf den Punkt N. Parameter: N muß einem Index aus *FigurePoints* entsprechen.

DoAll(F:pFigurePoint): rotiert, bewegt und vergrößert die Figur ausgehend von den Originalkoordinaten um die festgelegten Werte. Wird nicht direkt gerufen. Transformationen dürfen nur auf die Originalkoordinaten angewandt werden, damit es nicht zu einer Änderung der Koordinaten infolge von Rundungsfehlern kommt. Parameter: F ist ein zu transformierender Punkt.

CalcRotMatrix: berechnet Rotationsmatrix entsprechend *AxisAngles*. Wird nicht direkt aufgerufen.

CalcMatProd(var a,b,c:a3DMatrix): Matrizenprodukt C:=AB

RotPoint(var P:tTriple): Rotation des Punktes P

Die Objekte *tCube*, *tBounceBall* und *tRubberBand*

In den Konstruktoren dieser Objekte werden die Punkte, die zur Figur gehören, festgelegt. Für die Darstellung sorgen die im Basisobjekt der Hierarchie definierten Methoden *appear* und *disappear*. Es werden einige neue Methoden und Variablen aufgenommen, die spezielle Eigenschaften der Objekte widerspiegeln. So bestimmt zum Beispiel die Variable *DirSign* die Richtung, in die sich der Ball entsprechend bei den Objekten *tBounceBall* und *tRubberBand* bewegt.

Abgeleitete Objekte

Zur Demonstration der Eigenschaften der Unit »u3dfgr.pas« wurden die drei Objekte *tCube*, *tBounceBall* und *tRubberBand* vom Basisobjekt *t3DFigure* abgeleitet. Diese sind in der Unit »ucubes.pas« (Listing 2) enthalten. *tCube* stellt einen im Bildschirmbereich rotierenden und sich bewegenden Würfel dar. Von diesem ist *tBounceBall* abgeleitet. Als zusätzliche Eigenschaft wird der Würfel von einem Ball begleitet, der von der Würfeloberfläche abspringt und dennoch wieder auf den Würfel zurückfällt. Dabei bewegt sich der Würfel rotierend durch den dreidimensionalen Raum. Das Objekt *tRubberBand* schließlich ist von *tBounceBall* abgeleitet. Hier wird der springende Ball mit der Oberfläche durch ein Gummiseil verbunden.

Die Programme

Das Hauptprogramm für die Demonstration heißt »demo3d.pas« (Listing 3). Hier werden das Windows-Fenster aufgebaut, in dem die Darstellung ablaufen soll, Initialisierungen vorgenommen und die drei Objekte jeweils einmal instanziiert. Danach wird die Animation gestartet, die darin besteht, daß die virtuelle Objektmethode *Animate* innerhalb der OWL-Methode *Paint* zyklisch aufgerufen wird. Um die Anwendung unterbrechen zu können, wird der Zyklus durch einen Aufruf der Methode *InvalidateRect* erzeugt. Dabei wird innerhalb der sich ergebenden Schleife der Windows-Nachrichtenmechanismus durch die Funktion *KeepWindowsAlive* in jedem Zyklus aktiviert. (Matthias G. Döring/bm)

Mehr Information

1. DOS International 7'92 bis 10'92, Die dritte Dimension
2. Alan Watt, Fundamentals of Three-Dimensional Computer Graphics, Addison Wesley 1991, 430 Seiten, ISBN 0-201-15442-0, zirka 100 Mark
3. Matthias Oberdorfer, 3-Dimensionale Computergraphiken in C, IWT 1993, ISBN 3-88322-428-6, 330 Seiten, 89 Mark
4. Loren Heiny, Windows Graphics Programming with Borland C++, John Wiley & Sons Inc., Coriolis Group 1992, ISBN 0-471-54891-X, 474 Seiten, 80 Mark

Listing 1: u3dfr.pas

```

1: {Borland Pascal}
2: unit u3dfr;
3: ($X+)
4: interface
5: Uses Objects, OWindows, WinProcs, Win
  Types;
6: {Prefixvereinbarungen :
7: o - Original
8: a - Actual
9: w - Winkel
10: d - Delta
11: i - Init
12: ds - DrawStyle
13: c - Constant}
14: const
15: cWhite=$0FFFFFFF;
16: cBlack=$00000000;
17: cAnimaAngle =1.0;
18: cAnimaMove =1.0;
19: cAnimaZoom =0.0;
20: ds_Absolute =0001;
21: ds_Relative =0002;
22: ds_RemoveOriginal=0004;
23: ds_KeepOriginal =0008;
24: ds_Appear =0010;
25: ds_StayHidden =0020;
26: ds_Bouncing =0040;
27: ds_NonBouncing =0080;
28: ds_Standard=ds_Absolute or
29: ds_RemoveOriginal or ds_Appear or
30: ds_Bouncing;
31: ds_Rel=ds_Relative or
32: ds_RemoveOriginal or
33: ds_Appear or ds_Bouncing;
34: ds_Anima=ds_RemoveOriginal or
35: ds_Appear or ds_Bouncing;
36: ds_Init=ds_Absolute or
37: ds_KeepOriginal or
38: ds_StayHidden;
39: type
40: tTriple=record
41: x,y,z : real;
42: end;
43: a3DMatrix=array[1..3,1..3] of real;
44: tAnimaParams=record
45: Modus : word;
46: AngleDelta : tTriple;
47: MoveDelta : tTriple;
48: ZoomDelta : real;
49: end;
50: pFigurePoint=^tFigurePoint;
51: tFigurePoint=object(tObject)
52: constructor init(P : tTriple);
53: procedure MoveTo(P : tTriple);
54: procedure OMoveTo(P : tTriple);
55: procedure increm(D : tTriple);
56: procedure setAX(ix : real);
57: procedure setAY(iy : real);
58: procedure setAZ(iz : real);
59: function getOX:real;
60: function getOY:real;
61: function getOZ:real;
62: function getAX:real;
63: function getAY:real;
64: function getAZ:real;
65: private
66: oX,oY,oZ : real; {Originalkoordinaten}
67: aX,aY,aZ : real; {akt. Koordinaten}
68: end;
69: p3DAngle=^t3DAngle;
70: t3DAngle=object(tObject)
71: constructor init(A : tTriple);
72: procedure RotTo(A : tTriple);
73: procedure increm(D : tTriple);
74: procedure setWX(ix : real);
75: procedure setWY(iy : real);
76: procedure setWZ(iz : real);
77: function getWX:real;
78: function getWY:real;
79: function getWZ:real;
80: private
81: wX,wY,wZ : real;
82: end;
83: p3DDelta=^t3DDelta;
84: t3DDelta=object(tObject)
85: constructor init(D : tTriple);
86: procedure setDX(ix : real);
87: procedure setDY(iy : real);
88: procedure setDZ(iz : real);
89: function getDX:real;
90: function getDY:real;
91: function getDZ:real;
92: private
93: dX,dY,dZ : real;
94: end;
95: p3DFigure=^t3DFigure;
96: t3DFigure=object(tObject)
97: {Koordinaten d. Figur tFigurePoints}
98: FigurePoints : pCollection;
99: DrawDC : HDC;
100: {Ursprung K-rperkoordsystem}
101: OriginFigure : tFigurePoint;
102: {Rotationswinkel Achsen Koordsystem}
103: AxisAngles : t3DAngle;
104: ZoomFactor : real;
105: RotMatrix : a3DMatrix;
106: AnimaParams : tAnimaParams;
107: MaxX,MaxY,MaxZ : integer;
108: DrawWnd : hWnd;
109: constructor init(Wnd: hWnd;DC: HDC);
110: destructor done; virtual;
111: procedure MoveFgr
  (P : tTriple; M : word); virtual;
112: procedure RotaFgr
  (A : tTriple; M : word); virtual;
113: procedure ZoomFgr
  (S : real; M : word); virtual;

```

```

117: procedure Animate; virtual;
118: procedure ChangeAnimaParams
  (P : tAnimaParams); virtual;
119: procedure GetAnimaParams
  (var P : tAnimaParams); virtual;
120: procedure AddPoint
  (P : tTriple); virtual;
121: procedure appear; virtual;
122: procedure disappear; virtual;
123: procedure bounce; virtual;
124: procedure DrawLine(N : word);
125: procedure SetStart(N : word);
126: procedure DoAll(F : pFigurePoint);
127: procedure CalcRotMatrix;
128: procedure CalcMatProd(var a,b,c:
  a3DMatrix);
129: procedure RotPoint(var P: tTriple);
130: end;
131: implementation
132: { t3DDelta }
133: function t3DDelta.getDZ:real;
134: begin getDZ:=dZ; end;
135: function t3DDelta.getDZ:real;
136: begin getDZ:=dZ; end;
137: function t3DDelta.getDZ:real;
138: begin getDZ:=dZ; end;
139: function t3DDelta.getDZ:real;
140: begin getDZ:=dZ; end;
141: function t3DDelta.getDZ:real;
142: begin getDZ:=dZ; end;
143: function t3DDelta.getDZ:real;
144: begin getDZ:=dZ; end;
145: procedure t3DDelta.setDZ(iz : real);
146: begin dZ:=iz; end;
147: procedure t3DDelta.setDY(iy : real);
148: begin dY:=iy; end;
149: procedure t3DDelta.setDX(ix : real);
150: begin dX:=ix; end;
151: constructor t3DDelta.init(D: tTriple);
152: begin
153: inherited init;
154: dX:=D.x; dY:=D.y; dZ:=D.z;
155: end;
156: { t3DAngle }
157: function t3DAngle.getWZ:real;
158: begin getWZ:=wZ; end;
159: function t3DAngle.getWY:real;
160: begin getWY:=wY; end;
161: function t3DAngle.getWX:real;
162: begin getWX:=wX; end;
163: function t3DAngle.getWZ:real;
164: begin getWZ:=wZ; end;
165: procedure t3DAngle.setWZ(iz : real);
166: begin wZ:=iz; end;
167: procedure t3DAngle.setWY(iy : real);
168: begin wY:=iy; end;
169: procedure t3DAngle.setWX(ix : real);
170: begin wX:=ix; end;
171: procedure t3DAngle.RotTo(A: tTriple);
172: begin wX:=Pi*A.x/180.0; wY:=Pi*A.y/180.0;
173: wZ:=Pi*A.z/180.0;
174: end;
175: procedure t3DAngle.increm(D: tTriple);
176: begin
177: wX:=wX + Pi*D.x/180.0;
178: wY:=wY + Pi*D.y/180.0;
179: wZ:=wZ + Pi*D.z/180.0;
180: end;
181: constructor t3DAngle.init(A: tTriple);
182: begin
183: inherited init;
184: wX:=A.x; wY:=A.y; wZ:=A.z;
185: end;
186: { tFigurePoint }
187: function tFigurePoint.GetOX:real;
188: begin GetOX:=oX; end;
189: function tFigurePoint.GetOY:real;
190: begin GetOY:=oY; end;
191: function tFigurePoint.GetOZ:real;
192: begin GetOZ:=oZ; end;
193: procedure tFigurePoint.SetAX(ix:
  real);
194: begin aX:=ix; end;
195: procedure tFigurePoint.SetAY(iy:
  real);
196: begin aY:=iy; end;
197: procedure tFigurePoint.SetAZ(iz:
  real);
198: begin aZ:=iz; end;
199: function tFigurePoint.GetAX:real;
200: begin GetAX:=aX; end;
201: function tFigurePoint.GetAY:real;
202: begin GetAY:=aY; end;
203: function tFigurePoint.GetAZ:real;
204: begin GetAZ:=aZ; end;
205: procedure tFigurePoint.increm(D: tTr
  iple);
206: begin aX:=aX + D.x; aY:=aY + D.y;
207: aZ:=aZ + D.z; end;
208: procedure tFigurePoint.OMoveTo(P:
  tTriple);
209: begin oX:=P.x; oY:=P.y; oZ:=P.z; end;
210: procedure tFigurePoint.moveTo(P: tTr
  iple);
211: begin aX:=P.x; aY:=P.y; aZ:=P.z; end;
212: constructor tFigurePoint.init(P: tTr
  iple);

```

```

230: begin
231: inherited init;
232: oX:=P.x; oY:=P.y; oZ:=P.z;
233: aX:=P.x; aY:=P.y; aZ:=P.z;
234: end;
235: { t3DFigure }
236: procedure t3DFigure.SetStart(N: word);
237: var P: pFigurePoint;
238: begin
239: P:=FigurePoints^.At(N);
240: MoveTo(DrawDC,round(P^.GetAX),
241: round(P^.GetAY));
242: end;
243: procedure t3DFigure.DrawLine(N: word);
244: var P: pFigurePoint;
245: begin
246: P:=FigurePoints^.At(N);
247: LineTo(DrawDC,round(P^.GetAX),
248: round(P^.GetAY));
249: end;
250: procedure t3DFigure.disappear;
251: begin abstract; end;
252: procedure t3DFigure.appear;
253: begin abstract; end;
254: procedure t3DFigure.CalcMatProd
  (var a,b,c:a3DMatrix);
255: var i,j,k: byte;
256: s : real;
257: begin
258: for i:=1 to 3 do for k:=1 to 3 do
259: for j:=1 to 3 do s:=a[i,j]*b[j,k];
260: c[i,k]:=s;
261: end;
262: end;
263: procedure t3DFigure.CalcRotMatrix;
264: var T,U: a3DMatrix;
265: s,c: real;
266: i,j: byte;
267: begin with AxisAngles do begin
268: for i:=1 to 3 do for j:=1 to 3 do
269: T[i,j]:=0.0; U[i,j]:=0.0; end;
270: s:=sin(getWX); c:=cos(getWX); {x-Dre
  hung}
271: T[1,1]:=1.0; { 1 0
272: 0 }
273: T[2,2]:=c; T[2,3]:=-s; { 0 c
274: -s }
275: T[3,2]:=s; T[3,3]:=c; { 0 s
276: c }
277: s:=sin(getWY); c:=cos(getWY);
278: {y-Drehung}
279: U[1,1]:=c; U[1,3]:=-s; { c 0 s
280: 0 }
281: U[2,2]:=1.0; { 0 1 0
282: 0 }
283: U[3,1]:=s; U[3,3]:=c;
284: {z-Drehung}
285: for i:=1 to 3 do for j:=1 to 3 do
286: T[i,j]:=0.0;
287: T[1,1]:=c; T[1,2]:=-s;
288: { c -s 0 }
289: T[2,1]:=s; T[2,2]:=c;
290: { s c 0 }
291: T[3,3]:=1.0; { 0 0 1 }
292: CalcMatProd(T,U,RotMatrix);
293: {xyz-Drehung}
294: U:=RotMatrix;
295: s:=sin(getWZ); c:=cos(getWZ);
296: for i:=1 to 3 do for j:=1 to 3 do
297: T[i,j]:=0.0;
298: T[1,1]:=c; T[1,2]:=-s;
299: { c -s 0 }
300: T[2,1]:=s; T[2,2]:=c;
301: { s c 0 }
302: T[3,3]:=1.0; { 0 0 1 }
303: CalcMatProd(U,T,RotMatrix);
304: {xyz-Drehung}
305: end;
306: end;
307: procedure t3DFigure.RotPoint(var P:
  tTriple);
308: var T: tTriple;
309: begin
310: T.x:=RotMatrix[1,1]*P.x+
311: RotMatrix[1,2]*P.y+
312: RotMatrix[1,3]*P.z;
313: T.y:=RotMatrix[2,1]*P.x+
314: RotMatrix[2,2]*P.y+
315: RotMatrix[2,3]*P.z;
316: T.z:=RotMatrix[3,1]*P.x+
317: RotMatrix[3,2]*P.y+
318: RotMatrix[3,3]*P.z;
319: P:=T;
320: end;
321: procedure t3DFigure.DoAll(F:
  pFigurePoint);
322: var T: tTriple;
323: begin
324: T.x:=F^.GetOX * ZoomFactor; {Zoom}
325: T.y:=F^.GetOY * ZoomFactor;
326: T.z:=F^.GetOZ * ZoomFactor;
327: RotPoint(T);
328: T.x:=OriginFigure.GetAX + T.x;
329: T.y:=OriginFigure.GetAY + T.y;
330: {Translation}
331: T.z:=OriginFigure.GetAZ + T.z;
332: F^.MoveTo(T);
333: end;
334: procedure t3DFigure.RotaFgr(A:tTriple;
  M:word);
335: procedure All(F: pFigurePoint); far;
336: begin DoAll(F); end;
337: if (M and ds_RemoveOriginal) =

```

»u3dfr.pas« enthält das dreidimensionale Basisobjekt.


```

344: ds_RemoveOriginal then disappear;
345: if (M and ds_Relative)=ds_Relative
346: then AxisAngles.increm(A);
347: else AxisAngles.rotTo(A);
348: CalcRotMatrix;
349: FigurePoints^.ForEach(@All);
350: if (M and ds_Appear)=ds_Appear then
351: appear;
352: end;
353:
354: procedure t3DFigure.ZoomFgr(S: real;
355: M: word);
356: procedure All(F: pFigurePoint); far;
357: begin DoAll(F); end;
358: begin
359: if (M and ds_RemoveOriginal) =
360: ds_RemoveOriginal then disappear;
361: if (M and ds_Relative)=ds_Relative
362: then ZoomFactor:=ZoomFactor * S
363: else ZoomFactor:=S;
364: FigurePoints^.ForEach(@All);
365: if (M and ds_Appear)=ds_Appear then
366: appear;
367: end;
368:
369: procedure t3DFigure.MoveFgr(P: tTriple;
370: M: word);
371: procedure All(F: pFigurePoint); far;
372: begin DoAll(F); end;
373: begin
374: if (M and ds_RemoveOriginal) =
375: ds_RemoveOriginal then disappear;
376: if (M and ds_Relative)=ds_Relative
377: then OriginFigure.increm(P)
378: else OriginFigure.moveTo(P);
379: FigurePoints^.ForEach(@All);
380: if (M and ds_Appear)=ds_Appear then
381: appear;
382: end;
383:
384: procedure t3DFigure.Bounce;
385: begin
386: if (OriginFigure.getAX > MaxX)
387: or (OriginFigure.getAX < 0.0)
388: then AnimaParams.MoveDelta.x:=
389: -AnimaParams.MoveDelta.x;
390: if (OriginFigure.getAY > MaxY)
391: or (OriginFigure.getAY < 0.0)
392: then AnimaParams.MoveDelta.y:=
393: -AnimaParams.MoveDelta.y;
394: if (OriginFigure.getAZ > MaxZ)
395: or (OriginFigure.getAZ < 0.0)
396: then AnimaParams.MoveDelta.z:=
397: -AnimaParams.MoveDelta.z;
398: OriginFigure.increm
399: (AnimaParams.MoveDelta);
400: end;
401:
402: procedure t3DFigure.Animate;
403: procedure All(F: pFigurePoint); far;
404: begin DoAll(F); end;
405: begin
406: if (AnimaParams.Modus and
407: ds_RemoveOriginal)
408: =ds_RemoveOriginal
409: then disappear;
410: if (AnimaParams.Modus and
411: ds_Bouncing)=ds_Bouncing then
412: Bounce else OriginFigure.increm
413: (AnimaParams.MoveDelta);
414: AxisAngles.increm
415: (AnimaParams.AngleDelta);
416: ZoomFactor:=ZoomFactor *
417: AnimaParams.ZoomDelta;
418: CalcRotMatrix;
419: FigurePoints^.ForEach(@All);
420: if (AnimaParams.Modus and
421: ds_Appear)=ds_Appear then appear;
422: end;
423:
424: procedure t3DFigure.ChangeAnimaParams
425: (P: tAnimaParams);
426: begin AnimaParams:=P; end;
427: procedure t3DFigure.GetAnimaParams
428: (var P: tAnimaParams);
429: begin P:=AnimaParams; end;
430: procedure t3DFigure.AddPoint(P: tTri
431: ple);
432: begin FigurePoints^.Insert
433: (new(pFigurePoint,init(P))); end;
434:
435: destructor t3DFigure.done;
436: begin
437: dispose(FigurePoints,done);
438: inherited done;
439: end;
440:
441: constructor t3DFigure.init(Wnd: hWnd;
442: DC: HDC);
443: var T: tTriple;
444: i,j: byte;
445: begin
446: inherited init;
447: DrawWind:=wnd;
448: DrawDC:=DC;
449: (Animationdefaults)
450: AnimaParams.Modus:=ds_Anima;
451: T.x:=cAnimaAngle;
452: T.y:=cAnimaAngle;
453: T.z:=cAnimaAngle;
454: AnimaParams.AngleDelta:=T;
455: T.x:=cAnimaMove;
456: T.y:=cAnimaMove;
457: T.z:=cAnimaMove;
458: AnimaParams.MoveDelta:=T;
459: AnimaParams.ZoomDelta:=cAnimaZoom;
460: (-Defaults)
461: MaxX:=GetSystemMetrics(sm_CXScreen);
462: MaxY:=GetSystemMetrics(sm_CYScreen);
463: MaxZ:=100;

```

```

463: T.x:=0.0; T.y:=0.0; T.z:=0.0;
464: OriginFigure.init(T);
465: AxisAngles.init(T);
466: ZoomFactor:=1.0;
467: for i:=1 to 3 do
468: for j:=1 to 3 do RotMatrix[i,j]:=0.0;
469: for i:=1 to 3 do RotMatrix[i,i]:=1.0;
470: FigurePoints:=new(pCollection,init
471: (20,10));
472: end;
473: begin
474: end.

```

»u3dfgr.pas« enthält das dreidimensionale Basisobjekt.

Listing 2: ucubes.pas

```

1: (Borland Pascal)
2: unit ucubes;
3: interface
4: Uses Objects, OWindows, WinProcs,
5: WinTypes, u3dfgr;
6: const
7: CubeWidth = 60.0;
8: BallHeight = 60.0 + CubeWidth/2.0;
9: BallRadius = 10.0;
10: BounceStep = 3.0;
11:
12: type
13: pCube = ^tCube;
14: tCube = object (t3DFigure)
15: constructor init(Wnd: hWnd; DC: HDC);
16: procedure disappear; virtual;
17: procedure appear; virtual;
18: procedure bounce; virtual;
19: procedure CubeDraw; virtual;
20: end;
21:
22: pBounceBall = ^tBounceBall;
23: tBounceBall = object (tCube)
24: DirSign: real;
25: constructor init(Wnd: hWnd; DC: HDC);
26: procedure CubeDraw; virtual;
27: procedure MoveFgr(P: tTriple;
28: M: word); virtual;
29: procedure Animate; virtual;
30: procedure bounce; virtual;
31: procedure Ball; virtual;
32: end;
33:
34: pRubberBand = ^tRubberBand;
35: tRubberBand = object (tBounceBall)
36: constructor init(Wnd: hWnd; DC: HDC);
37: procedure CubeDraw; virtual;
38: procedure Ball; virtual;
39: end;
40:
41: implementation
42: { tRubberBand }
43: procedure tRubberBand.Ball;
44: var Z1,Z2: pFigurePoint;
45: T: tTriple;
46: begin
47: inherited Ball;
48: Z1 := FigurePoints^.At(8);
49: Z2 := FigurePoints^.At(10);
50: T.x:= Z1^.getOX; T.y:= Z1^.getOY;
51: T.z:= Z1^.getOZ - BallRadius;
52: Z2^.OMoveTo(T);
53: end;
54:
55: procedure tRubberBand.CubeDraw;
56: var P: pFigurePoint;
57: begin
58: inherited CubeDraw;
59: SetStart(9); DrawLine(10);
60: end;
61:
62: constructor tRubberBand.init(Wnd: hWnd;
63: DC: HDC);
64: var T: tTriple;
65: begin
66: inherited init(Wnd,DC);
67: T.x:= 0.0; T.y:= 0.0;
68: T.z:= CubeWidth/2.0; AddPoint(T);
69: T.z:= BallHeight - BallRadius;
70: AddPoint(T);
71: { tBounceBall }
72: procedure tBounceBall.Bounce;
73: begin
74: if (OriginFigure.getAX>MaxX-Ball
75: Height)or
76: (OriginFigure.getAX < 0.0 + Bal
77: lHeight)
78: then AnimaParams.MoveDelta.x
79: := -AnimaParams.MoveDelta.x;
80: if (OriginFigure.getAY > MaxY - Bal
81: lHeight)or
82: (OriginFigure.getAY < 0.0 + Bal
83: lHeight)
84: then AnimaParams.MoveDelta.y:=
85: -AnimaParams.MoveDelta.y;
86: if (OriginFigure.getAZ > MaxZ)
87: or (OriginFigure.getAZ < -MaxZ)
88: then AnimaParams.MoveDelta.z:=
89: -AnimaParams.MoveDelta.z;
90: OriginFigure.increm
91: (AnimaParams.MoveDelta);
92: end;
93: procedure tBounceBall.Ball;
94: var Z: pFigurePoint;
95: T: tTriple;

```

```

92: begin
93: Z:= FigurePoints^.At(8);
94: T.x:= Z^.getOX;
95: T.y:= Z^.getOY;
96: T.z:=Z^.getOZ-DirSign*BounceStep;
97: if T.z < CubeWidth/2.0 + BallRadius
98: then DirSign:= -1.0;
99: if T.z>BallHeight then DirSign:=1.0;
100: Z^.OMoveTo(T);
101: end;
102:
103: procedure tBounceBall.Animate;
104: procedure All(F: pFigurePoint); far;
105: begin DoAll(F); end;
106: begin
107: if (AnimaParams.Modus and
108: ds_RemoveOriginal)=ds_RemoveOriginal
109: then disappear;
110: if (AnimaParams.Modus and
111: ds_Bouncing)=
112: ds_Bouncing then Bounce
113: else OriginFigure.increm(
114: AnimaParams.MoveDelta);
115: AxisAngles.increm(AnimaParams.Angle
116: Delta);
117: ZoomFactor:= ZoomFactor *
118: AnimaParams.ZoomDelta;
119: CalcRotMatrix;
120: Ball;
121: FigurePoints^.ForEach(@All);
122: if (AnimaParams.Modus and ds_Appear)=
123: ds_Appear then appear;
124: end;
125:
126: procedure tBounceBall.MoveFgr
127: (P: tTriple; M: word);
128: procedure All(F: pFigurePoint); far;
129: begin DoAll(F); end;
130: begin
131: if (M and ds_RemoveOriginal) =
132: ds_RemoveOriginal then disappear;
133: if (M and ds_Relative)=ds_Relative
134: then OriginFigure.increm(P)
135: else OriginFigure.moveTo(P);
136: Ball;
137: FigurePoints^.ForEach(@All);
138: if (M and ds_Appear)=ds_Appear then
139: appear;
140: end;
141:
142: procedure tBounceBall.CubeDraw;
143: var P: pFigurePoint;
144: Z: real;
145: Brush: hBrush;
146: begin
147: inherited CubeDraw;
148: P := FigurePoints^.At(8);
149: Z := BallRadius;
150: Brush:= GetStockObject(Black_Brush);
151: SelectObject(DrawDC,Brush);
152: Ellipse(DrawDC,round(P^.GetAX-Z),
153: round(P^.GetAX+Z),
154: round(P^.GetAY-Z),
155: round(P^.GetAY+Z));
156: Brush:= SelectObject(DrawDC,Brush);
157: DeleteObject(Brush);
158: end;
159:
160: constructor tBounceBall.init(Wnd: hWnd;
161: DC: HDC);
162: var T: tTriple;
163: begin
164: inherited init(Wnd,DC);
165: T.x:= 0.0; T.y:= 0.0;
166: T.z:= BallHeight;
167: AddPoint(T);
168: DirSign:= 1.0;
169: end;
170:
171: { tCube }
172: procedure tCube.Bounce;
173: begin
174: if (OriginFigure.getAX > MaxX-Cub
175: eWidth)or
176: (OriginFigure.getAX < 0.0+Cub
177: eWidth)
178: then AnimaParams.MoveDelta.x:=
179: -AnimaParams.MoveDelta.x;
180: if (OriginFigure.getAY > MaxY-Cub
181: eWidth)or
182: (OriginFigure.getAY < 0.0+Cub
183: eWidth)
184: then AnimaParams.MoveDelta.y:=
185: -AnimaParams.MoveDelta.y;
186: if (OriginFigure.getAZ > MaxZ) or
187: (OriginFigure.getAZ < -MaxZ)
188: then AnimaParams.MoveDelta.z:=
189: -AnimaParams.MoveDelta.z;
190: OriginFigure.increm
191: (AnimaParams.MoveDelta);
192: end;
193:
194: procedure tCube.CubeDraw;
195: begin
196: SetStart(0);
197: DrawLine(1); DrawLine(2);
198: DrawLine(3); DrawLine(0);
199: DrawLine(4);
200: DrawLine(5); DrawLine(6);
201: DrawLine(7); DrawLine(4);
202: SetStart(1); DrawLine(5);
203: SetStart(2); DrawLine(6);
204: SetStart(3); DrawLine(7);
205: end;
206:
207: procedure tCube.disappear;
208: var Pen: hPen;
209: begin

```

»ucubes.pas« enthält abgeleitete dreidimensionale Objekte.


```

200: Pen:= GetStockObject(Black_Pen);
201: Pen:= SelectObject(DrawDC, Pen);
202: CubeDraw;
203: Pen:= SelectObject(DrawDC, Pen);
204: DeleteObject(Pen);
205: end;
206:
207: procedure tCube.appear;
208: var Pen: hPen;
209: begin
210:   Pen:= GetStockObject(White_Pen);
211:   Pen:= SelectObject(DrawDC, Pen);
212:   CubeDraw;
213:   Pen:= SelectObject(DrawDC, Pen);
214:   DeleteObject(Pen);
215: end;
216:
217: constructor tCube.init(Wnd: hWnd; DC:
    hDC);
218: var T: tTriple;
219:   Z: real;
220: begin
221:   inherited init(Wnd, DC);
222:   Z:= CubeWidth/2.0;
223:   T.x:= -Z; T.y:= +Z; T.z:= -Z; Ad
    dPoint(T);
224:   T.x:= +Z; AddPoint(T);
225:   T.y:= -Z; AddPoint(T);
226:   T.x:= -Z; AddPoint(T);
227:   T.y:= +Z; T.z:= +Z; AddPoint(T);
228:   T.x:= +Z; AddPoint(T);
229:   T.y:= -Z; AddPoint(T);
230:   T.x:= -Z; AddPoint(T);
231: end;
232: begin
233: end.

```

»ucubes.pas« enthält abgeleitete dreidimen-
sionale Objekte.

Listing 3: demo3d.pas

```

1: {Borland Pascal}
2: program Demo3D;
3:
4: uses Objects, OWindows, WinProcs,

```

```

5: WinTypes, u3DFgr, uCubes;
6:
7: type
8:   t3DFunctionTest=object(tApplication)
9:   procedure InitMainWindow; virtual;
10: end;
11:
12: p3DTestWindow=^t3DTestWindow;
13: t3DTestWindow=object(tWindow)
14:   DC : hDC;
15:   Cube : pCube;
16:   BounceBall : pBounceBall;
17:   RubberBand : pRubberBand;
18:   constructor Init(Header: pChar);
19:   destructor done; virtual;
20:   procedure GetWindowClass
    (var AWndClass:tWndClass); virtual;
21:   procedure Paint(PaintDC:hDC;
    var PaintInfo:tPaintStruct); virtual;
22: end;
23:
24: procedure KeepWindowsAlive;
25: var M: TMsg;
26: begin
27:   if PeekMessage(M, 0, 0, 0, PM_REMOVE) then
28:   begin
29:     TranslateMessage(M);
30:     DispatchMessage(M);
31:   end;
32: end;
33:
34: { t3DTestWindow }
35:
36: Procedure t3DTestWindow.GetWindowClass
37: (var AWndClass:tWndClass);
38: Begin
39:   tWindow.GetWindowClass(AWndClass);
40:   AWndClass.style:=0;
41:   AWndClass.hbrBackground:=
    GetStockObject(Black_Brush);
42: End;
43:
44:
45: constructor t3DTestWindow.init(Header:
    pChar);
46: var T: tTriple;
47: begin
48:   inherited init(nil, Header);
49:   Attr.Style:=ws_popup;
50:   Attr.X:=0; Attr.Y:=0;
51:   Attr.W:=GetSystemMetrics(sm_CXScreen);
52:   Attr.H:=GetSystemMetrics(sm_CYScreen);
53:   ShowCursor(false);
54:   DC := getDC(0);
55:   Cube:= new(pCube, init(0, DC));

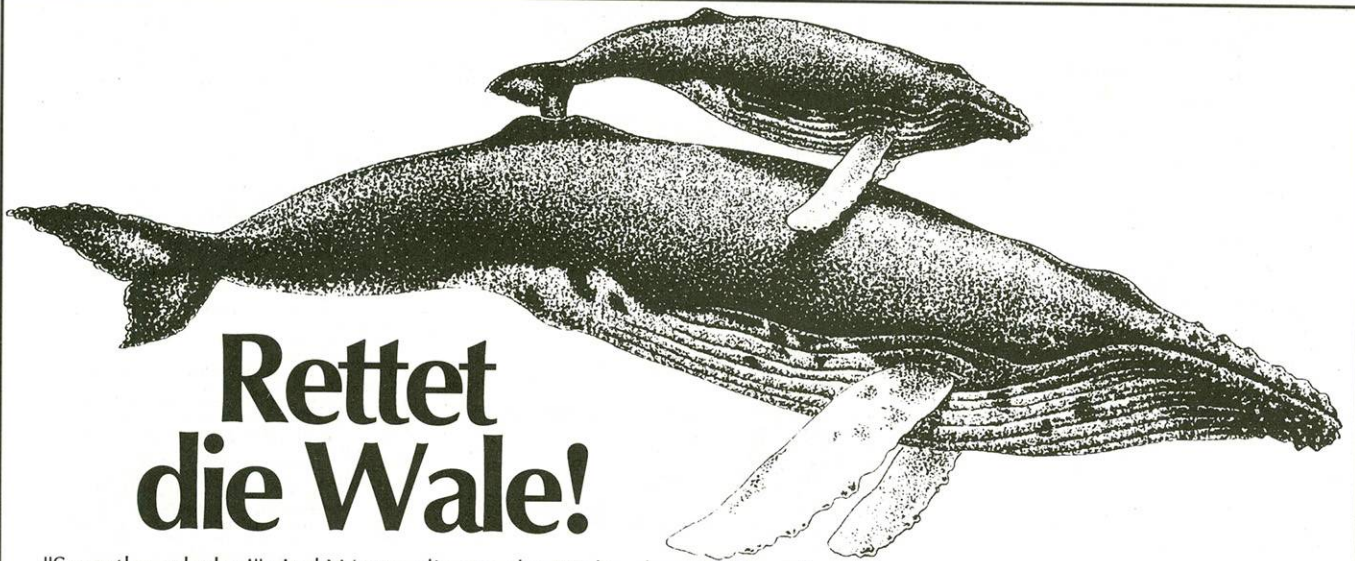
```

```

56: BounceBall:=new(pBounceBall,
    init(0, DC));
57: RubberBand := new(pRubberBand, init
    (0, DC));
58: T.x:=400.0; T.y:=100.0; T.z:=0.0;
59: Cube^.MoveFgr(T, ds_Init);
60: T.x:= 250.0;
61: BounceBall^.MoveFgr(T, ds_Init);
62: T.x:= 100.0;
63: RubberBand^.MoveFgr(T, ds_Init);
64: end;
65:
66: destructor t3DTestWindow.done;
67: begin
68:   dispose(Cube, done);
69:   dispose(BounceBall, done);
70:   dispose(RubberBand, done);
71:   releaseDC(0, DC);
72:   ShowCursor(true);
73:   inherited done;
74: end;
75:
76: procedure t3DTestWindow.Paint(PaintDC:
    hDC;
77: var PaintInfo:tPaintStruct);
78: begin
79:   KeepWindowsAlive;
80:   Cube^.Animate;
81:   BounceBall^.Animate;
82:   RubberBand^.Animate;
83:   InvalidateRect(hWindow, nil, false);
84: end;
85:
86: { t3DFunctionTest }
87: procedure t3DFunctionTest.InitMainWin
    dow;
88: begin MainWindow:= new(p3DTestWindow,
    init('3D-Test')); end;
89:
90:
91: var
92:   Tester: t3DFunctionTest;
93: begin
94:   Tester.init('3D-Test');
95:   Tester.run;
96:   Tester.done;
97: end.

```

»demo3d.pas« demonstriert die dreidimen-
sionalen Objekte.



Rettet die Wale!

"Save the whales!" sind Worte, die um die Welt gehen. Wale mit ihrem ausgeprägten Sozialleben gehören zu den intelligentesten Säugetieren unseres Planeten. Zwar hat die Natur sie vor gefährlichen Feinden weitgehend bewahrt, doch dem Menschen mit seinen ausgeklügelten, modernen Walfangmethoden sind sie hoffnungslos ausgeliefert. Das darf nicht sein! Wale dürfen nicht aussterben!

Die Deutsche Umwelthilfe e.V. unterstützt Projekte zum Schutz der bedrohten Meeressäuger.

Helfen Sie uns bei dieser wichtigen Arbeit mit Ihrer Spende und fordern Sie unser Informationsblatt "Rettet die Wale!" an.

Spendenkonto:

7997
Stadtsparkasse
Frankfurt
BLZ 500 501 02

☐ Ich bitte um Zusendung des Informationsblattes.
DM 1,50 in Briefmarken liegen bei.

☐ Ich unterstütze diese Aktion mit einer Spende.
Ein Scheck über DM _____ liegt bei.

Name: _____

Straße: _____

PLZ/Ort: _____



Deutsche Umwelthilfe

Güttinger Straße 19, 78315 Radolfzell

W1

Der Kampf für die Menschenrechte...

... ist nicht umsonst

Wie leicht gibt man Geld aus für viele Dinge, die über unseren Bedarf hinausgehen. Schon mit einem Teil dieses Geldes können Sie einen wichtigen Beitrag für die Wahrung der Menschenrechte leisten.

amnesty international tritt seit über 30 Jahren für die Freilassung von gewaltlosen politischen Gefangenen, gegen Folter und die Todesstrafe ein.

Tagtäglich werten ai-Mitarbeiterinnen und Mitarbeiter unterschiedliche Informationsquellen aus, Ermittlungsmissionen beobachten Prozesse, sprechen mit Regierungsbehörden und vor allem mit Opfern von Menschenrechtsverletzungen. Mit diesen gesicherten Informationen führen wir internationale Kampagnen und Aktionen durch, üben Druck auf Regierungen aus, um Menschenrechtsverletzungen zu beenden und zu verhindern.

Hierfür benötigen wir Ihre finanzielle Unterstützung. Ihr Beitrag kann für Betroffene das Leben bedeuten.

ai **amnesty international**
Für die Menschenrechte

amnesty international, 53108 Bonn

Spendenkonto 80 90 100

BKD Duisburg (BLZ 350 601 90)

Stichwort: "Wer mitmacht.."

Wer mitmacht, hilft gegen Ohnmacht.

Menüs vom Meister

Was passiert, wenn Sie die rechte Maustaste drücken? Richtig, nichts. In fast allen Anwendungen darf die rechte Maustaste immer nur zugucken. Das muß nicht sein. Wie wäre es mit einem selbstdefinierten Menü, das auf Druck der rechten Maustaste erscheint?

Einige fortgeschrittene Windows-Anwendungen wie etwa Excel erlauben die Erzeugung eigener Menüs in Form von Symbolleisten. Hier können Sie beliebige, oft benötigte Funktionen aufnehmen und diese sehr schnell erreichen. Eine weitere häufig verwendete Variante zum einfachen Zugriff auf ausgewählte Menüfunktionen sind die frei beweglichen Pop-up-Menüs, die per Druck auf die rechte Maustaste erscheinen (Bild). Derartige Menüs heißen Windows-intern *TrackPopupMenu*. Im folgenden wird die Bezeichnung Track-Menü benutzt. Solch ein selbstdefiniertes Track-Menü erzeugen Sie mit den Programmen dieses Beitrags.

■ So funktioniert es

Die Betätigung der rechten Maustaste in Menüs wird per Systemeingriff (mit Hooks) erkannt. Die so markierte Menüfunktion wird, wenn es sich um eine echte Funktion und nicht um den Bezeichner eines Submenüs handelt, in eine Liste aufgenommen. Ausgeführt wird das Kommando nicht.

Die Betätigung der rechten Maustaste außerhalb von Menüs hingegen sorgt für das Einblenden des gewünschten Track-Menüs. Mit einem kleinen Dialog können einzelne Funktionen aus dem Track-Menü wieder entfernt werden.

Sie müssen sich also nicht in zusätzlichen komplizierten Dialogen durch die Menüstruktur der Anwendung quälen, sondern können sich Ihre Wunschfunktionen problemlos aus dem ohnehin vorhandenen Hauptmenü der Anwendung herausuchen. Um das Gesamtsystem durch die beiden Filter nicht zu stark zu belasten, können diese per Menübefehl an- und abgeschaltet werden. So ist auch eine unbeabsichtigte Erweiterung des Track-Menüs ausgeschlossen.

■ Filterfreuden

Das erste zu lösende Problem besteht in der Erkennung der rechten Maustaste innerhalb von Menüs. Da in die eigentliche Menüverarbeitung nicht eingegriffen werden kann – zumindest nicht ohne großen Aufwand –, bietet sich der Einsatz von Filtern an.

Es sind zwei davon notwendig. Der erste heißt *MsgHook* und erkennt innerhalb von Menüs die Betätigung, genauer das Loslassen der rechten Maustaste. Die entsprechende Nachricht wird unterdrückt, statt dessen wird ein Flag (*SaveItem*) gesetzt und mit der API-Funktion *PostMessage* Drücken und Loslassen der linken Maustaste simuliert. Windows erzeugt daraufhin, entsprechend dem Menüeintrag unter dem Mauszeiger, eine neue Nachricht. Diese wird entweder ein Pop-up-Menü aufrufen oder aber eine echte Kommando-botschaft darstellen. Die Umsetzung des Mausclicks in eine Menübotschaft bleibt also weiterhin vollständig Windows überlassen.

Der zweite Filter (*GetHook*) erkennt alle echten Kommando-botschaften. Falls außerdem das Flag *SaveItem* gesetzt wurde, wandelt der Filter das Kommando in ein anderes um (*cm_CatchCommand*). Auf diese Weise wird die eigentliche Anwendung darüber informiert, daß im Menü eine Menüfunktion mit der rechten Maustaste aktiviert wurde.

Die beiden Filter selbst müssen aus Windows-internen Gründen in einer DLL (»tracklib.pas«, Listing 1) stehen. Diese wird in das nachfolgend beschriebene Objekt eingebunden. Das An- und Abschalten der Filter erledigt die Funktion *HookInstall*, die einzige Schnittstelle der DLL nach außen.

■ Das Menüobjekt

Zur Verwaltung des Track-Menüs ist im Hauptfenster der Anwendung eine Instanz des Objektes *TTrackMenu* zu schaffen, das von der Unit *Trackmem* zur Verfügung gestellt wird. Die Anwendung kommuniziert nur mit Methoden dieses einen Objektes.

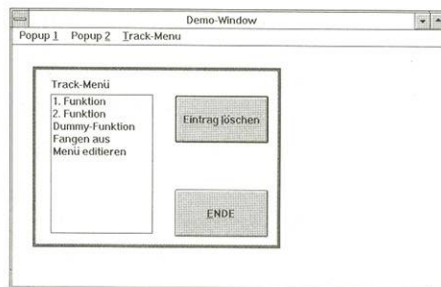
Aus einer *cm_CatchCommand*-Botschaft wird die eigentliche Kommandokonstante herausgelöst und mit der Methode *AppendItem* an das Objekt übergeben. Dort wird die Menüfunktion, genauer deren Kommandokonstante, in eine Kollektion aufgenommen, falls sie dort nicht schon enthalten ist. Die mehrfache Aufnahme eines Menüeintrags ist somit ausgeschlossen. Die Größe der Kollektion ist auf den feststehenden Wert *MaxTrackItems* begrenzt. Ein Meldungsfenster zeigt eine eventuelle

Überschreitung an. Sinnvolle Werte von *MaxTrackItems* liegen zwischen 5 und 8.

Über die Methode *CatchMenuItem* werden die Filter und damit die Sonderfunktion der rechten Maustaste aktiviert oder abgeschaltet. Die Betätigung der rechten Maustaste außerhalb von Menüs führt zum Aufruf der Methode *ActivateMenu*. Diese erzeugt zunächst ein leeres Pop-up-Menü und fügt dann alle Einträge aus der Kollektion ein. Text und Status des jeweiligen Eintrags holt sich die Methode aus dem Menü der Anwendung, womit die Aktualität sichergestellt ist.

Eine weitere Methode (*EditMenu*) steuert das Editieren des Menüinhalts. Sie erzeugt einen bescheidenen Dialog *TMenuDialog*, der lediglich aus einer Listbox und zwei Schaltern besteht. Einer der Schalter entfernt den in der Listbox markierten Eintrag aus der Liste und damit aus dem Track-Menü, der andere beendet den Dialog. Besonderheiten treten dabei nicht auf.

Beachten Sie allerdings, daß die Methode *TMenuDialog.SetupWindow* beim Füllen



Die Listbox zeigt den Inhalt des Track-Menüs und erlaubt das gezielte Löschen eines Eintrags.

der Listbox aus den Bezeichnungen der Menüfunktionen die &-Zeichen entfernen muß. Die Ressourcen für den Dialog sind direkt in die Unit eingebunden, so daß sich die Anwendung auch darum nicht kümmern muß.

■ Das Beispielprogramm

Das Programm »trackdem.pas« (Listing 2) zeigt ein einfaches Beispiel für den Einsatz der Unit »trackmem.pas« (Listing 3). Die Ressourcen für die Listbox befinden sich in »track.rc« (Listing 4). Alle Menüfunktionen für das Track-Menü stehen in einem separaten Pop-up-Menü. Alle anderen Menüeinträge bis auf den zum An- und Abschalten des Signaltons sind Dummies. Der Einfachheit halber wird das Menü direkt, also ohne Ressource erzeugt. Dies erspart bei Experimenten das ständige Wechseln zum Resource Workshop. Insbesondere können Sie erkennen, daß die Einträge im Track-Menü vom Zustand des Hauptmenüs abhängen. Deaktivierungen, Markierungen oder Änderungen des Textes werden zum Beispiel sowohl im Hauptmenü als auch im Track-Menü wirksam. Als Beispiel dazu dienen der

Felder und Methoden des Objektes TTrackMenu

WIND:HWnd; Fensterhandle des übergeordneten Fensters

MENU:HMenu; Menühandle des Hauptmenüs der Anwendung

CONSTRUCTOR Init(w:HWnd;m:HMenu); initialisiert das Objekt und belegt die beiden Felder **WIND** und **MENU**

DESTRUCTOR Done; Destruktor

PROCEDURE EditMenu(Window:PWindowsObject); ruft den Dialog zum Bearbeiten des Track-Menüs auf. Als Parameter ist der Verweis auf das aktuelle Fenster zu übergeben.

PROCEDURE AppendItem(cm:WORD); fügt den Menüeintrag mit der Kommandokonstante **cm** an das Track-Menü an

PROCEDURE ActivateMenu(x,y:INTEGER); stellt das Menü an den Koordinaten **x,y** dar und führt es aus

fPROCEDURE CatchMenuItem(f:BOOLEAN); installiert oder deinstalliert die Filterfunktionen entsprechend dem Inhalt von **f**

Beep-Eintrag (*Beep*) und der zur An- und Abschaltung der Filter (*Menüeintrag fangen* oder *Fangen aus*).

Noch ein Hinweis für Anwendungen, in denen das eigentliche Hauptfenster der Anwendung ein oder mehrere untergeordnete Fenster enthält. Das ist zum Beispiel bei MDI-Anwendungen oder dem Paint-Beispielprogramm der Fall. Das Hauptfenster erhält dabei keine **WM_RButtonUp**-Nachricht, da diese an eines der untergeordneten Fenster gelangt. Sie müssen also entweder den untergeordneten Fenstern eine Referenz auf des Track-Menü-Objekt übergeben, so daß diese *ActivateMenu* aufrufen können, oder aber die untergeordneten Fenster senden beim Erkennen der **WM_RButtonUp**-Nachricht diese an das Hauptfenster weiter.

(Bernd Matzke/bm)

Listing 1: tracklib.pas

```
1: { Tracklib.pas, TFW 1.5 }
2: LIBRARY TRACKLIB;
3:
4: { $C FIXED PERMANENT PRELOAD }
5: { D+ }
6: { L+ }
7:
8: USES
9: WinTypes, WinProcs;
10:
11: CONST
12:   cm_CatchCommand=9999;
13:
14:   OldMsgHook: TFarProc = NIL;
15:   OldGetHook: TFarProc = NIL;
16:   SaveItem: BOOLEAN=FALSE;
17: FUNCTION MsgHook(Code: INTEGER;
18:   WParam: WORD;
19:   LParam: LONGINT): LONGINT; EXPORT;
20: VAR
21:   CallDefHookProc: BOOLEAN;
22:   Result: LONGINT;
23:   p: PMSG;
24: BEGIN
25:   CallDefHookProc:= FALSE;
26:   Result:=0;
27:   CASE Code OF
28:     msgf_Menu: BEGIN
29:       p:=PMSG(LParam);
30:       IF (p^.message = WM_RButtonUp) THEN
31:         BEGIN PostMessage(p^.hwnd,
32:           WM_LButtonDown,p^.wparam, p^.lparam);
33:         PostMessage(p^.hwnd, WM_LButtonUp,
34:           p^.wparam, p^.lparam);
35:         SaveItem:=TRUE;
36:         Result:=1;
37:       END;
38:     END;
39:   hc_NoRemove: BEGIN
```

```
40:   END;
41: ELSE
42:   CallDefHookProc := TRUE;
43: END;
44: IF (Code < 0) OR (CallDefHookProc
45:   AND (OldMsgHook<>NIL))
46: THEN
47:   Result:=DefHookProc(Code, WParam,
48:     LParam, @OldMsgHook);
49:   MsgHook:=Result;
50: END;
51:
52: FUNCTION GetHook(Code: INTEGER;
53:   WParam: WORD; LParam: LONGINT):
54:   LONGINT; EXPORT;
55: VAR
56:   CallDefHookProc: BOOLEAN;
57:   Result: LONGINT; p: PMSG;
58: BEGIN
59:   Result:=0;
60:   CallDefHookProc:=FALSE;
61:   IF Code = hc_action THEN BEGIN
62:     p:=PMSG(LParam);
63:     IF (p^.message=WM_Command) AND
64:       SaveItem
65:     THEN BEGIN
66:       SaveItem:=FALSE;
67:       p^.message:=cm_CatchCommand;
68:     END;
69:   ELSE
70:     CallDefHookProc := TRUE;
71:   IF (Code < 0) OR
72:     (CallDefHookProc AND
73:       (OldGetHook<>NIL)) THEN
74:     Result:=DefHookProc(Code, WParam,
75:       LParam, @OldGetHook);
76:   GetHook:=Result;
77: END;
78:
79: PROCEDURE HookInstall(Install: BOOLEA
80:   N); EXPORT;
81: BEGIN
82:   IF Install THEN BEGIN
83:     OldMsgHook:=SetWindowsHook
84:       (wh_MsgFilter, @MsgHook);
85:     OldGetHook:=SetWindowsHook
86:       (wh_GetMessage, @GetHook);
87:   END
88:   ELSE BEGIN
89:     UnHookWindowsHook(wh_MsgFilter, @
90:       MsgHook);
91:     UnHookWindowsHook(wh_GetMessage,
92:       @GetHook);
93:   END;
94:
95:   OldMsgHook := NIL;
96:   OldGetHook := NIL;
97: END;
98:
99: EXPORTS
100: HookInstall;
101: BEGIN
102:   HookInstall;
103: END.
```

Die DLL »tracklib.pas« enthält die Filterfunktionen.

Listing 2: trackdem.pas

```
1: { Trackdem.pas, TFW 1.5 }
2: PROGRAM TrackDem;
3:
4: USES
5: WinDos, WObjects, WinTypes, WinProcs,
6: StdDlgs, Strings, TrackMen;
7:
8: CONST
9:   Beep: BOOLEAN = FALSE;
10:   CatchMenu: BOOLEAN = FALSE;
11:   cm_Func0 = 1000;
12:   cm_Func1 = 1001;
13:   cm_Func2 = 1002;
14:   cm_Func3 = 1003;
15:   cm_Func4 = 1004;
16:   cm_BeepOnOff = 1005;
17:   cm_CatchOnOff = 1006;
18:   cm_EditMenu = 1007;
19:
20:
21: TYPE
22:
23:   THookApp = Object(TApplication)
24:   PROCEDURE InitMainWindow; Virtual;
25: END;
26:
27:
28: PDemoW = ^TDemoW;
29: TDemoW = Object(TWindow)
30:   PTrackMenu: PTrackMenu;
31:   CONSTRUCTOR Init(AParent:
32:     PWindowsObject; ATitle: PCHAR);
33:   PROCEDURE SetupWindow;
34:   Virtual;
35:   PROCEDURE DefWndProc(VAR msg:
36:     TMessage); Virtual;
37:   PROCEDURE WMRButtonUp(VAR msg:
38:     TMessage); Virtual;
39:   WMRButtonUp;
40:   PROCEDURE CMCatchOnOff(VAR msg: TM
```

```
   message);
41:   Virtual cm_first + cm_CatchOnOff;
42:   PROCEDURE CMCatchCommand(VAR msg:
43:     TMessage);
44:   Virtual wm_first + cm_CatchCommand;
45:   PROCEDURE CMEditMenu(VAR msg: TMessage);
46:   Virtual cm_first + cm_EditMenu;
47:   PROCEDURE CMBeepEinAus
48:     (VAR Msg: TMessage);
49:   Virtual cm_First + cm_BeepOnOff;
50:   FUNCTION MakeMenu:HMenu;
51:   END;
52:
53: CONSTRUCTOR TDemoW.Init(
54:   AParent: PWindowsObject; ATitle: PCHAR);
55: BEGIN
56:   TWindow.Init(AParent, ATitle);
57:   Attr.Menu:=MakeMenu;
58: END;
59:
60: PROCEDURE TDemoW.SetupWindow;
61: BEGIN
62:   TWindow.SetupWindow;
63:   PTMenu:=New(PTrackMenu, Init(hwindow,
64:     attr.menu));
65: END;
66:
67: PROCEDURE TDemoW.DefWndProc(VAR msg:
68:   TMessage);
69: BEGIN
70:   IF msg.Message=WM_Command THEN BEGIN
71:     IF (msg.wparam >= cm_Func0) AND
72:       (msg.wparam <= cm_Func4)
73:     THEN BEGIN
74:       IF beep THEN
75:         MessageBeep(0);
76:       ELSE
77:         MessageBox(HWindow,
78:           'Funktion nicht implementiert',
79:           'Achtung:', mb_OK);
80:     END;
81:   ELSE
82:     TWindow.DefWndProc(msg);
83: END;
84:
85: PROCEDURE TDemoW.WMRButtonUp(VAR msg:
86:   TMessage);
87: VAR menupos:tpoint;
88: BEGIN
89:   MenuPos:=TPoint(msg.lparam);
90:   ClientToScreen(HWindow, MenuPos);
91:   PTMenu^.ActivateMenu(MenuPos.x,
92:     MenuPos.y);
93: END;
94:
95: PROCEDURE TDemoW.CMCatchOnOff(VAR Msg:
96:   TMessage);
97: VAR ItemName:ARRAY[0..20] OF CHAR;
98: BEGIN
99:   GetMenuString(Attr.Menu, cm_CatchOn
100:     Off, @ItemName, 20, mf_ByCommand);
101:   IF CatchMenu THEN BEGIN
102:     ModifyMenu(Attr.Menu, cm_CatchOnOff,
103:       mf_ByCommand, cm_CatchOnOff,
104:       '&Menü funktionen fangen');
105:     PTMenu^.CatchMenuItem(FALSE);
106:   END
107:   ELSE BEGIN
108:     ModifyMenu(Attr.Menu, cm_CatchOnOff,
109:       mf_ByCommand, cm_CatchOnOff,
110:       '&Fangen aus');
111:     PTMenu^.CatchMenuItem(TRUE);
112:   END;
113:   CatchMenu:= NOT CatchMenu;
114: END;
115:
116:
117: PROCEDURE TDemoW.CMCatchCommand(
118:   VAR msg:TMessage);
119: BEGIN
120:   PTMenu^.AppendItem(msg.wparam);
121: END;
122:
123: PROCEDURE TDemoW.CMEditMenu(VAR msg:
124:   TMessage);
125: BEGIN
126:   PTMenu^.EditMenu(@self);
127: END;
128:
129: PROCEDURE TDemoW.CMBeepEinAus(VAR Msg:
130:   TMessage);
131: BEGIN
132:   beep:= NOT beep;
133:   IF beep THEN
134:     CheckMenuItem(Attr.Menu, cm_BeepOnOff,
135:       mf_ByCommand OR mf_Checked);
136:   ELSE
137:     CheckMenuItem(Attr.Menu, cm_BeepOnOff,
138:       mf_ByCommand OR mf_Unchecked);
139: END;
140:
141: FUNCTION TDemoW.MakeMenu:HMenu;
142: VAR m,p,p2:HMenu;
143: BEGIN
144:   m:=CreateMenu;
145:   p:=CreatePopupMenu;
146:   AppendMenu(p, mf_string, cm_Func1,
147:     '&1. Funktion');
148:   AppendMenu(p, mf_string, cm_Func2,
149:     '&2. Funktion');
150:   AppendMenu(m, mf_string OR mf_popup,
151:     p, 'Popup &1');
152:   p:=CreatePopupMenu;
153:   p2:=CreatePopupMenu;
154:   AppendMenu(p2, mf_string,
```

Das Beispielprogramm »trackdem.pas« führt die selbstdefinierbaren Menüs vor.


```

148: cm_BeepOnOff, '&Beep');
149: AppendMenu(p2, mf_string, cm_Func3,
150: '&3. Funktion');
151: AppendMenu(p2, mf_string, cm_Func4,
152: '&4. Funktion');
153: AppendMenu(p, mf_string, cm_Func0,
154: '&Dummy-Funktion');
155: AppendMenu(p, mf_string OR
156: mf_Popup, p2, '&Submenü');
157: AppendMenu(m, mf_string OR mf_Popup,
158: p, 'Popup &2');
159:
160: p:=CreatePopupMenu;
161: AppendMenu(p, mf_string, cm_EditMenu,
162: '&Menü editieren');
163: AppendMenu(p, mf_string, cm_CatchOnOff,
164: '&Kommandos fangen');
165: AppendMenu(m, mf_string OR mf_Popup,
166: p, '&Track-Menü');
167: MakeMenu:=m;
168: END;
169:
170: PROCEDURE THookApp.InitMainWindow;
171: BEGIN
172:   MainWindow := New(PDemoW, Init(NIL,
173:   'Demo-Window'));
174: END;
175:
176: VAR
177:   AHookApp: THookApp;
178:
179: BEGIN
180:   AHookApp.Init('TrackDem');
181:   AHookApp.Run;
182:   AHookApp.Done;
183: END.

```

Das Beispielprogramm »trackdem.pas« führt die selbstdefinierbaren Menüs vor.

Listing 3: trackmem.pas

```

1: { Trackmem.pas, TPW 1.5 }
2: UNIT trackmem;
3: INTERFACE
4: USES WinTypes, WinProcs, WObjects,
5:   Strings;
6: {$R TRACK.RES}
7:
8: CONST cm_CatchCommand=9999;
9:   MaxTrackItems = 7;
10:
11: TYPE
12:   PTrackMenu=^TTrackMenu;
13:   TTrackMenu=Object
14:   Wind:HWND;
15:   Menu:HMenu;
16:   CONSTRUCTOR Init(w:HWND;m:HMenu);
17:   DESTRUCTOR Done;
18:   PROCEDURE EditMenu(Window:PWindow
19:   sObject);
20:   PROCEDURE AppendItem(cm:WORD);
21:   PROCEDURE ActivateMenu(x,y:INTEGER);
22:   PROCEDURE CatchMenuItem(f:BOOLEAN);
23: END;
24:
25: PROCEDURE HookInstall(Install: BOOLEA
26: N);
27:
28: IMPLEMENTATION
29:
30: PROCEDURE HookInstall;
31: EXTERNAL 'TRACKLIB';
32:
33: CONST
34:   id_ListBox=200;
35:   id_DelItem=201;

```

```

34: TYPE
35:   PTrackMenuItem="WORD;
36:
37:   PMenuColl="TMenuColl;
38:   TMenuColl=Object(TCollection)
39:   PROCEDURE FreeItem(Item:POINTER);
40:   Virtual;
41: END;
42:
43: PMenuDialog="TMenuDialog;
44: TMenuDialog=Object(TDialog)
45:   Menu: HMenu;
46:   CONSTRUCTOR Init(w:PWindowsObject;
47:   m:HMenu);
48:   PROCEDURE SetupWindow; Virtual;
49:   PROCEDURE DelItem(VAR Msg: TMessage);
50:   Virtual id_First + id_DelItem;
51: END;
52:
53: VAR MenuItems:PMenuColl;
54: PROCEDURE TMenuDialog.FreeItem
55:   (Item:POINTER);
56: BEGIN
57:   Dispose(PTrackMenuItem(item));
58: END;
59:
60: CONSTRUCTOR TMenuDialog.Init
61:   (w:PWindowsObject; m:HMenu);
62: BEGIN
63:   TDialog.Init(w, 'LISTBOX');
64:   Menu:=m;
65: END;
66:
67: PROCEDURE TMenuDialog.SetupWindow;
68: VAR i : INTEGER;
69:   Item: PTrackMenuItem;
70:   p : ARRAY[0..40] OF CHAR;
71:   c : PCHAR;
72: BEGIN
73:   TWindowsObject.SetupWindow;
74:   FOR i:=0 TO MenuItems.count-1 DO
75:     BEGIN
76:       Item:=MenuItems.at(i);
77:       GetMenuItemString(Menu, Item^, p, 40,
78:       mf_ByCommand);
79:       c:=StrPos(p, '&');
80:       IF c<>NIL THEN BEGIN
81:         c^:=#0;
82:         StrCat(p, c+1);
83:       END;
84:       SendDlgItemMsg(id_ListBox,
85:       lb_AddString, 0, LONGINT(Addr(p)));
86:     END;
87:   END;
88:
89: PROCEDURE TMenuDialog.DelItem(VAR Msg:
90: TMessage);
91: VAR l:LONGINT;
92: BEGIN
93:   l:=SendDlgItemMsg(id_ListBox,
94:   lb_GetCurSel, 0, 0);
95:   IF l <> lb_Err THEN BEGIN
96:     SendDlgItemMsg(id_ListBox,
97:     lb_DeleteString, WORD(l), 0);
98:     MenuItems.AtFree(l);
99:   END;
100: END;
101:
102: CONSTRUCTOR TTrackMenu.Init(w:HWND;
103: m:HMenu);
104: BEGIN
105:   Wind:=w;
106:   Menu:=m;
107:   MenuItems:=New(PMenuColl,
108:   Init(MaxTrackItems, 0));
109: END;
110:
111: DESTRUCTOR TTrackMenu.Done;
112: BEGIN
113:   Dispose(MenuItems.Done);
114: END;
115:
116: PROCEDURE TTrackMenu.AppendItem(cm:WO
117: RD);
118: VAR Item:PTrackMenuItem;
119:
120: FUNCTION existiert(ColItem: POINTER):

```

```

115: BOOLEAN; far;
116: BEGIN
117:   existiert:=WORD(ColItem^)=cm;
118: END;
119:
120: BEGIN
121:   IF MenuItems.FirstThat(@existiert)=NIL
122:   THEN BEGIN
123:     IF MenuItems.count<MaxTrackItems
124:     THEN BEGIN New(Item);
125:       Item^:=cm;
126:       MenuItems.Insert(Item);
127:     END
128:   ELSE
129:     MessageBox(Wind, 'Track-Menü i. voll',
130:     'Achtung', mb_ok);
131:   END;
132: END;
133:
134:
135: PROCEDURE TTrackMenu.EditMenu(
136:   Window:PWindowsObject);
137: BEGIN
138:   Application^.ExecDialog(New
139:   (PMenuDialog, Init(Window, Menu)));
140: END;
141:
142: PROCEDURE TTrackMenu.ActivateMenu
143:   (x,y: INTEGER);
144: VAR TrackMenu:HMenu;
145:   Flags :WORD;
146:   Item :PTrackMenuItem;
147:   i :INTEGER;
148:   s :ARRAY[0..40] OF CHAR;
149: BEGIN
150:   TrackMenu:=CreatePopupMenu;
151:   FOR i:=0 TO MenuItems.count-1 DO
152:     BEGIN
153:       Item:=MenuItems.at(i);
154:       Flags:=GetMenuItemState(Menu, Item^,
155:       mf_ByCommand);
156:       GetMenuItemString(Menu, Item^, s, 40,
157:       mf_ByCommand);
158:       AppendMenu(TrackMenu, Flags, Item^, s);
159:     END;
160:   END;
161:   TrackPopupMenu(TrackMenu, 0, x, y,
162:   0, Wind, NIL);
163:   DestroyMenu(TrackMenu);
164: END;
165:
166: PROCEDURE TTrackMenu.CatchMenuItem
167:   (f: BOOLEAN);
168: BEGIN
169:   IF f THEN
170:     HookInstall(TRUE)
171:   ELSE
172:     HookInstall(FALSE);
173: END;
174:
175: END.

```

Die Unit »trackmem.pas« enthält wichtige Methoden für das Beispielprogramm.

Listing 4: track.rc

```

1: ; track.rc
2: LISTBOX DIALOG 15, 16, 156, 111
3: STYLE WS_POPUP | WS_DLGFRAME
4: BEGIN
5:   CONTROL "LISTBOX", 200, "LISTBOX", L
6:   BS_NOTIFY | WS_CHILD | WS_VISIBLE | WS_
7:   BORDER | WS_VSCROLL | WS_GROUP, 10, 16,
8:   65, 90
9:   PUSHBUTTON "Eintrag &löschen", 201, 90,
10:   16, 60, 30
11:   PUSHBUTTON "&ENDE", 2, 90, 76, 60, 30
12:   LTEXT "Track-Menü", -1, 10, 5, 60, 8
13: END

```

Die Ressourcendatei »track.rc« für die Listbox

Eine Bitte an unsere Abonnenten

Vermerken Sie bei Schriftverkehr und Zahlungen neben der vollständigen Anschrift stets Ihre Abo-Nummer.

Sie vermeiden damit unnötige Verzögerungen bei der Bearbeitung Ihres Abonnements.

Vielen Dank
Ihre DMV-Versandabteilung

Aktiver Strahlenschutz

Wer kennt sie nicht, die kleinen Hilfsprogramme, die eine Wohltat für unsere Bildschirme sind. Deren Programmierung setzte jedoch bisher detaillierte Kenntnisse des Systems voraus. Windows 3.1 zeigt sich hier kooperativer. Es definiert eine Programmierschnittstelle und stellt Funktionen bereit, über die Sie mit geringem Aufwand eigene Bildschirmschoner in Ihr System einhängen.

Wenn Sie schon einige Zeit mit Windows 3.1 arbeiten, wird Ihnen aufgefallen sein, daß die Desktop-Dialogbox der Systemsteuerung eine Liste von zur Verfügung stehenden Bildschirmschonern anbietet. Jeder Eintrag in der Liste entspricht einer Datei im Standard-Windows-Verzeichnis mit der Dateierweiterung »scr«. Bei diesen Dateien handelt es sich um nahezu gewöhnliche Windows-Applikationen, die nur aus Gründen der Erkennbarkeit mit einer anderen Endung versehen sind. Um Ihren Bildschirm und auch Sie selbst in den Genuß weiterer Animationen zu bringen, kopieren Sie nur das entsprechende Programm (Bild 1) – mit der Endung »scr« – in Ihr Windows-Verzeichnis und finden dann in der Desktop-Dialogbox einen weiteren Eintrag vor.

Schoner unter Windows 3.0

Unter Windows 3.0 mußten Sie, falls Sie einen Bildschirmschoner selbst implementieren wollten, sämtliche Tastatur- und Mauseingaben eigenhändig abfangen, einen Zeitgeber (Timer) verwalten und – sofern eine gewisse Zeit lang keine Benutzeraktivität vorlag – das Schonerfenster aktivieren. Dies konnten Sie durch Verwendung zweier Filterprozeduren (Hooks), eine für Tastatur- und eine für Mauseingaben, realisieren. Der Beitrag »Windows am Haken« (WinDOS 8'92, Seite 78) beschreibt, wie Sie solche Filter in eigenen Programmen verwenden. Ein gewaltiger Nachteil dieser Vorgehensweise ist, daß ein derart programmierter Bildschirmschoner zu Beginn einer Windows-Sitzung gestartet wird und dann bis zum bitteren Ende im Arbeitsspeicher verharrt und somit Ressourcen und Speicherplatz bindet.

Das neue Konzept in Windows 3.1

Windows 3.1 bietet hier ein weitaus besseres Konzept: Sobald Sie einen Bildschirmschoner wie oben beschrieben auswählen, kontrolliert Windows selbst die entsprechenden Nachrichten und aktiviert den Schoner erst, wenn keine Benutzeraktivität

vorliegt. Die Zeit, die Windows wartet, stellen Sie ebenfalls über die Desktop-Dialogbox ein. Eine Applikation muß nun folgendermaßen aufgebaut sein, um als Bildschirmschoner Verwendung zu finden: Wichtig ist zunächst, daß Windows die ausgewählte Applikation mit zwei unterschiedlichen Parametern aufruft. Um Ihnen neben einer schönen Animation auch das Gefühl zu geben, daß Sie selbst während der inaktiven Zeit noch Herr Ihrer Maschine sind, stellen Ihnen die meisten Bildschirmschoner eine Dialogbox zur Konfiguration zur Verfügung. Über diese teilen Sie dem Bildschirmschoner mit, wel-

che Farben er verwenden und mit welcher Geschwindigkeit die Animation ablaufen soll. Für den Schonvorgang startet Windows die Applikation mit dem Parameter /s; um die Konfigurationsbox einzublenden, mit /c als Parameter. Folglich entscheiden Sie als Programmierer, welche Fensterklasse, nämlich die mit meist schwarzem Hintergrund für den Schoner oder die für die Dialogbox, in WinMain registriert werden muß und welche Fensterprozedur zur Bearbeitung der Nachrichten herangezogen werden soll. Nun beginnen Sie mit einem Compiler Ihrer Wahl einen Bildschirmschoner für

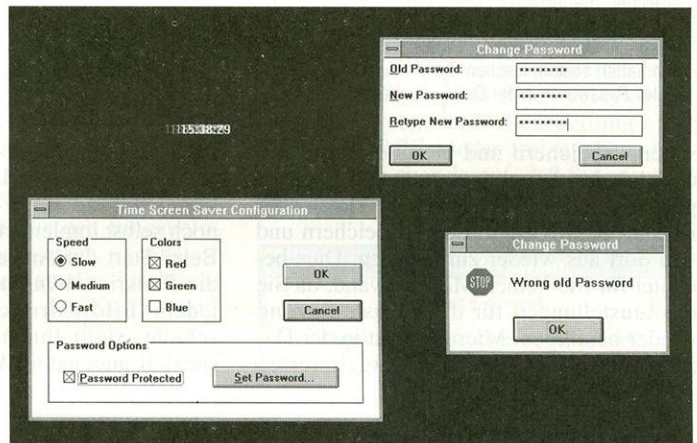


Bild 1. Der Bildschirmschoner »ssstime.scr« schiebt die aktuelle Zeit als Laufschrift über den Bildschirm.

Windows 3.1 zu realisieren. Noch ein wichtiger Hinweis: Da die Applikation entweder zum Konfigurieren oder zum Schonen gestartet wird, wäre es falsch, die im Konfigurationsdialog vorgenommenen Einstellungen in globalen Variablen zwi-

Identifikation	Wert	Nutzung
Dialog:		
DLG_CHANGEPASSWORD	2000	Dialogbox zur Änderung des Paßwortes (Bild 2)
Elemente:		
ID_ETOLD	104	Eingabefeld für altes Paßwort
ID_ETNEW	105	Eingabefeld für neues Paßwort
ID_ETAGAIN	106	Eingabefeld zur Verifikation des neuen Paßwortes
Dialog:		
DLG_ENTERPASSWORD	2001	fragt das Paßwort ab, wenn der Schonvorgang abgebrochen wird (Bild 3)
Element:		
ID_ETPASSWORD	107	Eingabefeld für das Paßwort
Dialog:		
DLG_INVALIDPASSWORD	2002	erscheint bei Eingabe eines falschen Paßwortes (Bild 4)
Dialog:		
DLG_SCRNSAVECONFIGURE	2003	wird bei Betätigung des Setup-Knopfes in der Desktop-Dialogbox eingeblendet (Bild 5)
Elemente:		
ID_XXX	xxx	benutzerdefinierte Elemente

Tabelle 1. Übersicht über die von »scrnsave.lib« benutzten Dialogboxen und deren Elemente. Das Aussehen der Dialogboxen für »ssstime.scr« finden Sie in den Bildern 2 bis 5.

Bemerkungen zum Paßwortschutz

Wenn Sie denken, daß Ihr PC durch Verwendung eines paßwortgeschützten Bildschirmschoners zu einem datenschutzrechtlich sicheren Rechner wird, muß ich Sie leider enttäuschen. Allzuleicht kann dieser Paßwortschutz umgangen werden, denn durch einfaches Ein- und Ausschalten des Rechners ist er bereits außer Kraft gesetzt. Dabei gehen natürlich die nicht gespeicherten Daten der letzten Sitzung verloren. MS-DOS und Windows sind standardmäßig keinesfalls als sichere Betriebssysteme konzipiert. Mit entsprechender Software können Sie jedoch Ihren Rechner auch unter MS-DOS ein wenig sicherer machen und somit die Hemmschwelle für einen Eindringling erhöhen. Und dies erreichen Sie teilweise bereits durch Einsatz eines paßwortgeschützten Bildschirmschoners.

Noch ein Tip: Falls Sie Ihr Paßwort einmal vergessen haben sollten, löschen Sie einfach den Eintrag hinter *Password=* in der Datei »control.ini«.

schenzuspeichern und hierüber auf einfache Art der Animationsroutine zur Verfügung zu stellen. Sie können die Daten aber in einer Konfigurationsdatei speichern und von dort aus wieder zurücklesen. Dies bedeutet für Sie keinen Mehraufwand, da Sie die Einstellungen für die nächste Sitzung wieder benötigen. Microsoft hat in der Datei »control.ini« für die mitgelieferten

Bildschirmschoner die entsprechenden Eintragungen vorgesehen. Die jeweiligen Sektionen besitzen alle die Form *[Screen Saver.name]*, wobei *name* für den jeweiligen Bildschirmschoner steht. Natürlich können Sie auch eine separate Ini-Datei verwenden.

Die Bibliothek »scrnsave.lib«

Microsoft stellt Ihnen mit der Bibliothek »scrnsave.lib«, die im Software Development Kit 3.1 und C/C++ 7.0 enthalten ist, eine einfache Schnittstelle zur Programmierung von Bildschirmschonern zur Verfügung. Diese erlaubt es Ihnen, nur die Teile der Applikation implementieren zu müssen, die bei jedem Bildschirmschoner unterschiedlich sind. Als Beispiel dient ein Bildschirmschoner, der die aktuelle Zeit als Laufschrift über den Bildschirm schiebt: »stime.c« (Listing 1) nutzt »scrnsave.lib« und zeigt Ihnen, was Sie alles noch selbst implementieren müssen.

Beim Start der Anwendung ruft Windows die Funktion *WinMain()* auf. Da dort bei jedem Bildschirmschoner dasselbe geschieht, stellt Ihnen die Bibliothek eine speziell angepaßte *WinMain()* zur Verfü-

gung, die beim Linken zu Ihrem Modul dazugebunden wird. Dort wird je nach Kommandozeilenoption entschieden, ob eine Klasse für die Konfigurationsdialogbox oder für das Schonerfenster registriert wird.

Die Dialogbox zur Konfiguration

Die Beschreibung Ihrer Konfigurationsdialogbox definieren Sie in Ihrer Ressourcen-datei unter der Identifikation *DLG_SCRNSAVECONFIGURE*. Diese Kennung ist in der Include-Datei »scrnsave.h« (Listing 2) als Konstante definiert. Bei Nutzung eines Paßwortschutzes für Ihren Bildschirmschoner

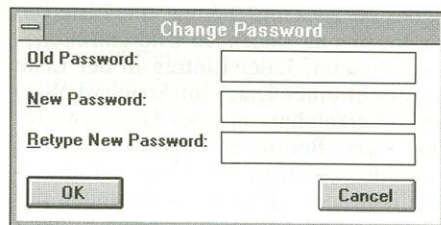


Bild 2. Mit dem Dialog *DLG_CHANGEPASSWORD* ändern Sie das Paßwort.

Ist Ihr Programm der HIT?

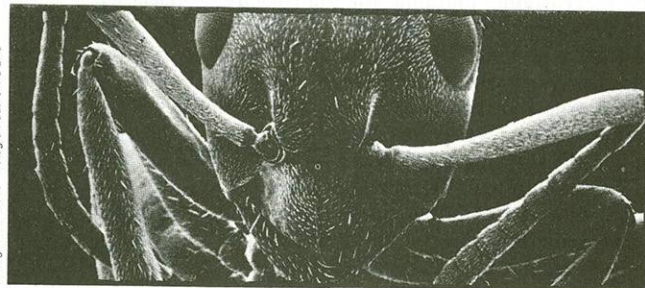
Der DMV-Verlag sucht ständig
nach neuer, interessanter
Software zur Aufnahme in
unser Softwaresortiment.

Dabei ist es einerlei, ob Sie nun
ein Anwendungs- oder ein
Spielprogramm geschrieben
haben.

Der DMV-Verlag bietet ihnen sein
Software-Know-how an!

DMV-Verlag, Abt. Software
Postfach 1146, 85580 Poing

DIE INTELLIGENZ DES MENSCHEN WIRD VON JEDER AMEISE ÜBERTROFFEN.



Über Jahrtausende haben wir es mühsam zum aufrechten Gang gebracht, und jetzt verhalten wir uns dümmer als jeder Einzeller. Kein anderes Lebewesen vernichtet seine eigenen Lebensgrundlagen. Der WWF sichert das Überleben bedrohter Tier-

und Pflanzenarten und damit auch die Basis menschlichen Lebens. Helfen Sie uns. WWF, Postf., W-60591 Frankfurt/M.70.



WWF

Mensch, die Zeit drängt.



Bild 3. Der Dialog `DLG_ENTERPASSWORD` fragt das Paßwort ab, wenn der Schonvorgang abgebrochen wird.

ner sind noch weitere Dialogboxen notwendig. Diese binden Sie auf dieselbe Weise an Ihre Applikation an. Tabelle 1 zeigt eine Übersicht über die vorgesehenen Dialoge und die dazugehörigen Identifikationen. Bei der Registrierung der Klasse für die Konfigurationsdialogbox gibt »scrnsave.lib« die Funktion `ScreenSaverConfigureDialog()` als Fensterprozedur an. Diese Funktion implementieren Sie bei jedem Bildschirmschoner selbst, da dort individuelle Einstellungen vorgenommen werden. Es handelt sich hierbei um eine gewöhnliche Fensterprozedur, Sie erhalten also dieselben Parameter wie bei jeder anderen Fensteroutine.

»sstime.c« erlaubt Ihnen hierbei die Einstellung von Geschwindigkeit und Farbe der Uhrzeitangabe, die innerhalb des Schonerfensters über den Bildschirm huscht. Bei der Initialisierung der Dialogbox (über die Nachricht `WM_INITDIALOG`) liest das Programm zunächst mit der Hilfsfunktion `GetIniSetting()` die alten Einstellungen aus der Datei »control.ini« und setzt anschließend die Marken der Radioschaltflächen und Checkboxes entsprechend. Einen Auszug aus der Datei »control.ini« finden Sie in der Textbox. Erhält `ScreenSaverConfigureDialog()` eine `WM_COMMAND`-Nachricht, stellen Sie über den Parameter `wParam` die Identifikation des Elementes fest, das die Nachricht verursacht hat. `IDOK` ist die Identifikation der OK-Schaltfläche. Wenn das Programm dessen Nachricht empfängt, schließt sich die Dialogbox, und die Einstellungen werden über die Hilfsfunktion `SetIniSetting()` gesichert.

■ Dialog zum Setzen des Paßwortes

Möchten Sie Ihren Schoner mit einem Paßwortschutz versehen, sollten Sie im Konfigurationsdialog hierfür eine Checkbox und eine Schaltfläche vorsehen. Über die Checkbox wählt der Benutzer aus, ob der Paßwortschutz aktiv ist oder nicht. In »sstime.dlg« (Listing 3) ist dies die Checkbox mit der Identifikation `ID_PASSWORDPROTECTED`. Auf eine Nachricht von dieser Checkbox hin sollten Sie die Schaltfläche, über die der Benutzer eine Dialogbox zum Ändern des Paßwortes öffnen kann, aktivieren oder deaktivieren. In »sstime.dlg« ist dies die Schaltfläche mit

der Kennung `ID_SETPASSWORD`. Bei der Änderung des Paßwortes werden Sie wieder von »scrnsave.lib« unterstützt. Für die von Ihnen beschriebene Dialogbox mit der Kennung `DLG_CHANGEPASSWORD` steht Ihnen die Dialogboxroutine `DlgChangePassword()` zur Verfügung, die Ihnen die Bearbeitung der Nachrichten abnimmt. Dazu müssen natürlich die drei Textfelder der Dialogbox die richtige Kennung besitzen (Tabelle 1).

Um die Dialogbox einzublenden, benutzen Sie die Funktion `MakeProcInstance()`, da es sich bei `DlgChangePassword()` um eine Callback-Funktion handelt. Anschließend blenden Sie die Dialogbox mit der Funktion `DialogBox()` ein. In der Dialogbox kann der Benutzer nun das Paßwort setzen. Bei einer Änderung wird das neue Paßwort automatisch in der Datei »control.ini« verschlüsselt gespeichert. `DialogBox()` kehrt erst dann wieder zurück, wenn der Benutzer die Dialogbox schließt. Damit er anschließend nur noch die Eingabetaste betätigen muß, sollten Sie den Eingabefokus auf den OK-Knopf setzen. »sstime.c« realisiert dies, indem es die Nachricht `WM_NEXTDLGCTL` an den Konfigurationsdialog sendet.

■ Das Schonerfenster

Bei Aufruf der Applikation mit dem Parameter `/s`, also wenn der Bildschirmschoner aktiviert werden soll, wird in `WinMain()` innerhalb von »scrnsave.lib« eine Fensterklasse für ein Fenster ohne Rahmen und Titel registriert, das den gesamten Bildschirm überdeckt. Dabei wird als Fensterprozedur die Funktion `ScreenSaverProc()` angegeben. In dieser von Ihnen zu implementierenden Funktion können Sie auf alle Nachrichten reagieren, in der Regel sind jedoch einige wenige ausreichend. Alle nicht ausgewerteten Nachrichten sollten Sie an die Funktion `DefScreenSaverProc()` weiterleiten, die Ihnen die Bibliothek zur Verfügung stellt. Somit kann ein Standardverhalten für alle Bildschirmschoner realisiert werden. `DefScreenSaverProc()` reagiert zum Beispiel auf eine Benutzerakti-

vität mit Beenden der Applikation. Falls der Schoner paßwortgeschützt ist, wird natürlich zuerst das Paßwort abgefragt. »sstime.c« reagiert auf die Nachricht `WM_TIMER` mit dem Anzeigen der Uhrzeit auf dem Bildschirm. Um das Ganze nicht zu langweilig zu gestalten, verändern Sie die Position für die Ausgabe bei jedem Aufruf, indem Sie zur Variablen `xTimePos` einen Wert hinzuaddieren, der von den Einstellungen in der Konfigurationsdialogbox abhängt. Die Einstellungen lesen Sie bei Erhalten der Nachricht `WM_CREATE` mit der Hilfsfunktion `GetIniSetting()` ein. Dort sollten Sie auch den Zeitgeber zur Erzeugung von `WM_TIMER`-Nachrichten initialisieren. »scrnsave.lib« schließt die Applikation durch Versenden einer `WM_CLOSE`-Nachricht. Dadurch ist es Ihnen erlaubt, die von Ihnen belegten Systemressourcen bei Bearbeitung von `WM_DESTROY` wieder freizugeben. Achten Sie dabei aber darauf, daß Sie anschließend entweder `DefScreenSaverProc()` auf-



Bild 4. Der Dialog `DLG_INVALIDPASSWORD` erscheint bei Eingabe eines falschen Paßwortes.

rufen oder die Anwendung selbst mit `PostQuitMessage()` korrekt beenden.

Der Inhalt des Bildschirms bleibt bei Aufruf des Schoners zunächst erhalten. Dadurch können Sie diesen für Ihre Animationen nutzen, wie dies zum Beispiel einige Schoner aus dem Paket »Idle Wild« tun. »sstime.c« nutzt diese Option nicht, sondern füllt den kompletten Bildschirm schwarz, um einen neutralen Hintergrund zu erhalten.

Beachten Sie bei Ihren eigenen Animationen, daß ein Bildschirmschoner den Bildschirm auch tatsächlich schonen sollte. Dafür ist es sinnvoll, den größten Teil des

Name	Zweck
<code>szIniFile</code>	Name der verwendeten Initialisierungsdatei
<code>szScreenSaver</code>	Sektionsname für das Paßwort
<code>szPassword</code>	Name des Eintrags für das Paßwort
<code>szAppName</code>	Name der Sektion für Ihren Bildschirmschoner
<code>szIsPassword</code>	Name des Eintrags für das Paßwort aktiv (0)/inaktiv (1)
<code>szName</code>	Titel für die Dialoge <code>DLG_ENTERPASSWORD</code> und <code>DLG_INVALIDPASSWORD</code>
<code>szChangePW</code>	Titel für den Dialog <code>DLG_CHANGEPASSWORD</code>
<code>szDifferentPW</code>	Fehlermeldung: erstes und zweites neues Paßwort unterschiedlich
<code>szBadOldPw</code>	Fehlermeldung: falsches erstes Paßwort
<code>szNoHelpMemory</code>	Fehlermeldung: nicht genügend Speicher zum Einblenden der Hilfe
<code>szHelpFile</code>	Name der Hilfedatei

Tabelle 2. Übersicht über die globalen Zeichenketten einer Bildschirmschoner-Anwendung

Name	Export	Zweck
ScreenSaverProc	ja	Fensterprozedur des Schonerfensters
ScreenSaverConfigureDialog	ja	Fensterprozedur des Konfigurationsdialogs
DlgChangePassword	ja	Fensterprozedur für den Dialog <i>DLG_CHANGEPASSWORD</i>
DlgGetPassword	ja	Fensterprozedur für den Dialog <i>DLG_ENTERPASSWORD</i>
DlgInvalidPassword	ja	Fensterprozedur für den Dialog <i>DLG_INVALIDPASSWORD</i>
HelpMessageFilterHookFunction	ja	fragt die F1-Taste ab und sendet die Nachricht <i>MyHelp Message</i> an <i>ScreenSaverConfigureDialog</i>
RegisterDialogClasses	nein	erlaubt die Registrierung zusätzlicher Dialogboxen für die Konfiguration
DefScreenSaverProc	nein	Standard-Nachrichtenbearbeitung für <i>ScreenSaverProc</i>

Tabelle 3. Übersicht über die für »scrnsave.lib« wichtigen Funktionen. Wenn unter »Export« »ja« angegeben ist, müssen Sie die Funktion in der Moduldefinitionsdatei exportieren.

des Bildschirms schwarz zu belassen. Vermeiden Sie auf jeden Fall, daß ein Bereich längere Zeit nicht neu gezeichnet wird, denn dadurch kommt es zum Einbrennen des Bildschirms.

Globale Variablen und Strings

Die Datei »sstime.c« enthält eine große Anzahl globaler Variablen. Diese werden an verschiedenen Stellen innerhalb von »scrnsave.lib« benutzt und müssen folglich in Ihrem Modul definiert werden. Die Bedeutung der einzelnen Variablen entnehmen Sie Tabelle 2. Damit die Bibliothek korrekt arbeitet, initialisieren Sie die Variablen mit den entsprechenden Werten. Am sinnvollsten ist es, die Zeichenketten

```
[ScreenSaver]
Password=

[Screen Saver.Time]
TimeSpeed=24
TimeColorRed=255
TimeColorGreen=0
TimeColorBlue=255
PWProtected=1
```

Auszug aus der Datei »control.ini«

in der Zeichenkettentabelle (Stringtable) der Applikation abzulegen und bei Programmstart von dort einzulesen. In der Headerdatei »scrnsave.h« sind dafür auch entsprechende Kennungen definiert. Um die jeweilige Identifikation zu erhalten, ersetzen Sie einfach *sz* durch *ids* im Namen der globalen Variablen. Eine davon, nämlich *idsAppName*, wird in »sstime.rc« (Listing 4) bereits benutzt. Auf diesen Eintrag können Sie nicht verzichten, da die entsprechende Zeichenkette innerhalb der Bibliothek eingelesen und in *szAppName* global abgelegt ist. Sstime verzichtet auf die Nutzung der Stringtable für die übrigen Zeichenketten, da das Programm sonst zu unübersichtlich würde.

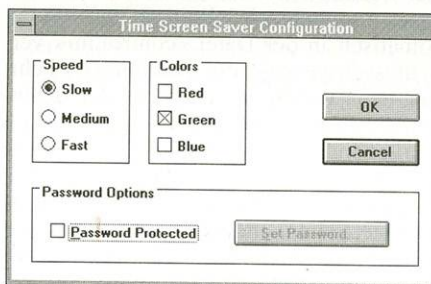


Bild 5. Der Dialog *DLG_SCRNSAVECONFIGURE* wird bei Betätigung des Setup-Knopfes in der Desktop-Dialogbox eingeblendet und dient der Konfiguration des Bildschirmschoners.

Hinzufügen von Hilfe

Dem Anwender Ihrer Windows-Applikation sollten Sie eine entsprechende (kontext-sensitive) Hilfe anbieten, wenn dieser die F1-Taste betätigt. In der Bibliothek »scrnsave.lib« ist dies bereits vorgesehen. Durch Belegen der globalen Variablen *szHelpFile* mit »control.hlp« erhält der Benutzer in allen Situationen automatisch eine angemessene Hilfestellung. Über die Variable *MyHelpMessage* realisieren Sie, wie in »sstime.c« gezeigt, eine Hilfe für Ihren Konfigurationsdialog.

Exportieren von Funktionen

In Tabelle 3 sehen Sie eine Übersicht über alle Funktionen, die für die Nutzung von »scrnsave.lib« wichtig sind. Weiterhin ist dort angegeben, welche dieser Funktionen Sie in Ihrer Definitionsdatei »sstime.def« (Listing 5) exportieren müssen und welche Sie nicht exportieren dürfen.

(Thomas Trotzki/Gerald Wick/uk)

Literaturhinweis

Weitere Hinweise zur Realisierung von Bildschirmschonern finden Sie in der SDK-3.1-Dokumentation von Microsoft, Kapitel 14 in Band 1 der »Programmer's Reference« beschreibt die einzelnen Funktionen der Bildschirmschonerbibliothek. Dort fehlen allerdings wichtige Hinweise auf die Verwendung der globalen Variablen. In der Online-Hilfe »SDK-Reference« unter »Function Groups« finden Sie den Eintrag »Screen Saver Functions«. Dieser enthält nahezu den kompletten Inhalt des Handbuchs.

Listing 1: sstime.c

```
1: // Programm: sstime.c
2: // Funktion: Screen Saver für Windows
3: //
4: // Sprache: MS C/C++ 7.00 + SDK 3.1
5: // Scheck: C-Format, ASCII
6: // Autor: Thomas Trotzki
7: // (C) 1992 DMV Widuch GmbH & Co. KG */
8: #include <windows.h>
9: #include <scrnsave.h>
10: #include <time.h>
11: #include "sstime.h"
12: #define ID_TIMER 1
13:
14: /* Variablen fuer Screen Saver Library */
15: HINSTANCE hMainInstance;
16: HWND hMainWindow;
17: HOOKPROC fpMessageFilter;
18:
19: char szIniFile[MAXFILELEN] =
20:     "control.ini";
21: char szScreenSaver[22] = "ScreenSaver";
22: char szPassword[16] = "Password";
23: char szAppName[TITLEBARNAMELEN];
24: char szIsPassword[22] = "PWProtected";
25:
26: char szName[TITLEBARNAMELEN] =
27:     "Enter Password";
28: char szChangePW[30] = "Change Password";
29:
30: char szDifferentPW[BUFFLEN] =
31:     "Wrong Verification";
32: char szBadOldPW[BUFFLEN] =
33:     "Wrong old Password";
34: char szNoHelpMemory[BUFFLEN] =
35:     "Not enough memory available";
36:
37: char szHelpFile[MAXFILELEN] =
38:     "control.hlp";
39: // MSG-Parameter bei [F1]
40: UINT MyHelpMessage = 1;
41:
42: // wird aus <scrnsave.lib> aufgerufen
43: BOOL RegisterDialogClasses(
44:     HINSTANCE hInst)
45: {
46:     // Registrierung zusätzlicher
47:     // Dialogboxen hier vornehmen
48:     return TRUE;
49: }
50:
51: // liest Werte aus control.ini
52: int GetIniSetting(
53:     int *xTimeSpeed,
54:     COLORREF *lTimeColor,
55:     BOOL *bProtected)
56: {
57:     int iRed, iGreen, iBlue;
58:
59:     *xTimeSpeed = GetPrivateProfileInt(
60:         szAppName, "TimeSpeed",
61:         16, szIniFile);
62:     iRed = GetPrivateProfileInt(
63:         szAppName, "TimeColorRed",
64:         0, szIniFile);
65:     iGreen = GetPrivateProfileInt(
66:         szAppName, "TimeColorGreen",
67:         0, szIniFile);
68:     iBlue = GetPrivateProfileInt(
69:         szAppName, "TimeColorBlue",
70:         0, szIniFile);
71:     *lTimeColor = RGB(iRed, iGreen, iBlue);
72:     *bProtected = GetPrivateProfileInt(
73:         szAppName, szIsPassword,
74:         0, szIniFile);
75: }
76:
77: // Hilfsfunktion
78: void WcCtrlIniInt(
79:     LPCSTR lpszEntry, int iValue)
80: {
81:     char szValue[10];
82:
83:     sprintf(szValue, "%d", iValue);
84:     WritePrivateProfileString(szAppName,
85:         lpszEntry, szValue, szIniFile);
86: }
87:
88: // setzt Werte in control.ini
89: int SetIniSetting(
90:     int xTimeSpeed,
91:     COLORREF lTimeColor,
92:     BOOL bProtected)
93: {
94:     WcCtrlIniInt("TimeSpeed", xTimeSpeed);
95:     WcCtrlIniInt("TimeColorRed",
96:         GetRValue(lTimeColor));
97:     WcCtrlIniInt("TimeColorGreen",
98:         GetGValue(lTimeColor));
99:     WcCtrlIniInt("TimeColorBlue",
100:         GetBValue(lTimeColor));
101:     WcCtrlIniInt("szIsPassword",
102:         bProtected);
103: }
104:
105: // Diese Funktion erhaelt alle messages
106: // der Konfigurationsdialogbox
107: BOOL WINAPI ScreenSaverConfigureDialog(
108:     HWND hDlg, UINT message,
109:     WPARAM wParam, LPARAM lParam)
110: {
111:     static int xTimeSpeed;
112:     static COLORREF lTimeColor;
113:     DLGPROC fpDialog;
114:     static HWND hOK;
```



```

115: static HWND hSetPassword;
116: static BOOL bPassword;
117:
118: switch(message)
119: {
120:     case WM_INITDIALOG:
121:         hOK = GetDlgItem(hDlg, IDOK);
122:         hSetPassword =
123:         GetDlgItem(hDlg, ID_SETPASSWORD);
124:         GetIniSetting(&xTimeSpeed,
125:         &lTimeColor, &bPassword);
126:         CheckDlgButton(hDlg,
127:         ID_PASSWORDPROTECTED, bPassword);
128:         EnableWindow(
129:         hSetPassword, bPassword);
130:         CheckRadioButton(hDlg,
131:         ID_TIMESLOW, ID_TIMEFAST,
132:         ID_TIMESLOW + (xTimeSpeed/8-1));
133:         CheckDlgButton(hDlg, ID_TIMERED,
134:         (lTimeColor & RGB(255,0,0)) ? 1:0);
135:         CheckDlgButton(hDlg, ID_TIMEGREEN,
136:         (lTimeColor & RGB(0,255,0)) ? 1:0);
137:         CheckDlgButton(hDlg, ID_TIMEBLUE,
138:         (lTimeColor & RGB(0,0,255)) ? 1:0);
139:         return TRUE;
140:     case WM_COMMAND:
141:         switch(wParam)
142:         {
143:             case IDOK:
144:                 EndDialog(hDlg, TRUE);
145:                 SetIniSetting(
146:                 xTimeSpeed,
147:                 lTimeColor, bPassword);
148:                 return TRUE;
149:             case IDCANCEL:
150:                 EndDialog(hDlg, FALSE);
151:                 return TRUE;
152:             case ID_TIMESLOW: /* kein break */
153:             case ID_TIMEMEDIUM:
154:             case ID_TIMEFAST:
155:                 xTimeSpeed = (wParam -
156:                 ID_TIMESLOW + 1) * 8;
157:                 return TRUE;
158:             case ID_TIMERED:
159:                 lTimeColor ^= RGB(255,0,0);
160:                 return TRUE;
161:             case ID_TIMEGREEN:
162:                 lTimeColor ^= RGB(0,255,0);
163:                 return TRUE;
164:             case ID_TIMEBLUE:
165:                 lTimeColor ^= RGB(0,0,255);
166:                 return TRUE;
167:             case ID_SETPASSWORD:
168:                 if(!fpDialog =
169:                 (DLGPROC) MakeProcInstance(
170:                 (FARPROC) DlgChangePassword,
171:                 hMainInstance)) == NULL)
172:                     return FALSE;
173:                 DialogBox(hMainInstance,
174:                 MAKEINTRESOURCE(
175:                 DLG_CHANGEPASSWORD),
176:                 hDlg, fpDialog);
177:                 FreeProcInstance(
178:                 (FARPROC) fpDialog);
179:                 SendMessage(hDlg, WM_NEXTDLGCTL,
180:                 (LPARAM) hOK, 1L);
181:                 return TRUE;
182:             case ID_PASSWORDPROTECTED:
183:                 bPassword ^= 1;
184:                 CheckDlgButton(hDlg,
185:                 ID_PASSWORDPROTECTED,
186:                 bPassword);
187:                 EnableWindow(
188:                 hSetPassword, bPassword);
189:                 return TRUE;
190:             default:
191:                 return FALSE;
192:         }
193:     default:
194:         if (message == MyHelpMessage)
195:         {
196:             MessageBox(hMainInstance,
197:             "No Help available", szName, MB_OK);
198:             return TRUE;
199:         }
200:         return FALSE;
201:     }
202: return FALSE;
203: }
204:
205: //Diese Funktion erhaelt alle messages
206: //des screen save window
207: LRESULT WINAPI ScreenSaverProc(
208:     HWND hwnd, UINT message,
209:     WPARAM wParam, LPARAM lParam)
210: {
211:     PAINTSTRUCT ps;
212:     HDC hdc;
213:     RECT rect;
214:     static int xTimePos, yTimePos;
215:     static int xTimeSpeed;
216:     static int cxWidth, cyHeight;
217:     static idTimer;
218:     CREATESTRUCT FAR* lpCS;
219:     char szTime[10];
220:     static COLORREF lTimeColor;
221:     static BOOL bPassword;
222:
223:     switch(message)
224:     {
225:         case WM_CREATE:
226:             idTimer = SetTimer(
227:             hwnd, ID_TIMER, 200, NULL);
228:             lpCS = (CREATESTRUCT FAR*) lParam;
229:             cxWidth = lpCS->cx;
230:             cyHeight = lpCS->cy;
231:             xTimePos = 0;
232:             yTimePos = cyHeight / 4;
233:             GetIniSetting(
234:             &xTimeSpeed,

```

```

235:     &lTimeColor, &bPassword);
236:     break;
237:     case WM_TIMER:
238:         strtime(szTime);
239:         hdc = GetDC(hwnd);
240:         SetBkColor(hdc, RGB(0,0,0));
241:         TextOut(hdc, xTimePos, yTimePos,
242:         " ", 15);
243:         xTimePos = (xTimePos+xTimeSpeed)
244:         % cxWidth;
245:         SetTextColor(hdc, lTimeColor);
246:         TextOut(hdc, xTimePos, yTimePos,
247:         szTime, strlen(szTime));
248:         ReleaseDC(hwnd, hdc);
249:         break;
250:     case WM_DESTROY:
251:         if (idTimer!=0)
252:             KillTimer(hwnd, ID_TIMER);
253:         break;
254:     case WM_ERASEBKND:
255:         GetClientRect(hwnd, &rect);
256:         FillRect(&rect, wParam, &rect,
257:         (HBRUSH) GetStockObject(
258:         BLACK_BRUSH));
259:         return 0;
260:     default:
261:         break;
262:     }
263:     return DefScreenSaverProc(
264:     hwnd, message, wParam, lParam);
265: }

```

Das Programm »sstime.c« schont Ihren Bildschirm, indem es die Uhrzeit über den Bildschirm wandern läßt.

Listing 2: sstime.h

```

1: #define DLG_CHANGEPASSWORD 2000
2: #define DLG_ENTERPASSWORD 2001
3: #define DLG_INVALIDPASSWORD 2002
4: #define DLG_SCRNSAVECONFIGURE 2003
5:
6: #define ID_OLDTEXT 100
7: #define ID_NEWTEXT 101
8: #define ID_AGAIN 102
9: #define ID_PASSWORD 103
10: #define ID_ETOLD 104
11: #define ID_ETNEW 105
12: #define ID_ETAGAIN 106
13: #define ID_ETPASSWORD 107
14:
15: #define ID_PASSWORDPROTECTED 110
16: #define ID_SETPASSWORD 111
17:
18: #define ID_TIMERED 2010
19: #define ID_TIMEGREEN 2011
20: #define ID_TIMEBLUE 2012
21: #define ID_TIMESLOW 2020
22: #define ID_TIMEMEDIUM 2021
23: #define ID_TIMEFAST 2022

```

In der Headerdatei »sstime.h« legen Sie Konstanten für die verwendeten Dialogboxen fest.

Listing 3: sstime.dlg

```

1: DLGINCLUDE RCDATA DISCARDABLE
2: BEGIN
3:     "SSTIME.HV0"
4: END
5: DLG_SCRNSAVECONFIGURE DIALOG
6,18,220,135
6: STYLE DS_MODALFRAME | WS_POPUP |
7: WS_VISIBLE | WS_CAPTION | WS_SYSMENU
8: CAPTION "Time Screen Saver
Configuration"
9: FONT 8, "MS Sans Serif"
10: BEGIN
11:     GROUPBOX "Speed", 2006,10,10,50,60,
12:     WS_GROUP
13:     CONTROL "Slow", ID_TIMESLOW, "Button",
14:     BS_AUTORADIOBUTTON | WS_GROUP |
15:     WS_TABSTOP, 15, 24, 39, 10
16:     CONTROL "Medium", ID_TIMEMEDIUM,
"Button",
17:     BS_AUTORADIOBUTTON, 15, 40, 39, 10
18:     CONTROL "Fast", ID_TIMEFAST, "Button",
19:     BS_AUTORADIOBUTTON, 15, 55, 39, 10
20:     GROUPBOX "Colors", 2004,70,10,50,60
21:     CONTROL "Red", ID_TIMERED, "Button",
22:     BS_AUTOCHECKBOX | WS_TABSTOP,
23:     75,25,40,10
24:     CONTROL "Green", ID_TIMEGREEN, "Button",
25:     BS_AUTOCHECKBOX | WS_TABSTOP,
26:     75, 40, 40, 10
27:     CONTROL "Blue", ID_TIMEBLUE, "Button",
28:     BS_AUTOCHECKBOX | WS_TABSTOP,
29:     75, 55, 40, 10
30:     GROUPBOX "Password Options",
31:     1, 10, 80, 200, 45
32:     CHECKBOX "Password Protected",
33:     ID_PASSWORDPROTECTED, 20,100,80,14,
34:     WS_GROUP
35:     PUSHBUTTON "&Set Password...",
36:     ID_SETPASSWORD, 115, 100, 80, 14,

```

```

37: WS_DISABLED | WS_GROUP
38: PUSHBUTTON "OK", IDOK,160,30,50,15
39: DEFPUSHBUTTON "Cancel", IDCANCEL,
40: 160, 55, 50, 15
41: END
42:
43: DLG_CHANGEPASSWORD DIALOG 20,100,174,79
44: STYLE DS_MODALFRAME | WS_POPUP |
45: WS_CAPTION | WS_SYSMENU
46: CAPTION "Change Password"
47: FONT 8, "MS Sans Serif"
48: BEGIN
49:     LTEXT "&Old Password:", ID_OLDTEXT,
50:     4, 3, 80, 14
51:     EDITTEXT ID_ETOLD, 84, 3, 80, 14,
52:     ES_PASSWORD
53:     LTEXT "&New Password:", ID_NEWTEXT,
54:     4, 21, 80, 14
55:     EDITTEXT ID_ETNEW, 84, 21, 80, 14,
56:     ES_PASSWORD
57:     LTEXT "&Retype New Password:",
ID_AGAIN,
58:     4, 39, 80, 16
59:     EDITTEXT ID_ETAGAIN, 84, 39, 80,14,
60:     ES_PASSWORD
61:     DEFPUSHBUTTON "OK", IDOK,4,59,40,14
62:     PUSHBUTTON "Cancel", IDCANCEL,
63:     124, 59, 40, 14
64: END
65:
66: DLG_INVALIDPASSWORD DIALOG 50,50,174,79
67: STYLE DS_MODALFRAME | WS_POPUP |
68: WS_CAPTION | WS_SYSMENU
69: CAPTION "Invalid Password"
70: FONT 8, "MS Sans Serif"
71: BEGIN
72:     CTEXT "Incorrect password; Check your
screen saver password and try again.",
73:     -1, 22, 6, 130, 40
74:     DEFPUSHBUTTON "OK", IDOK,71,53,40,14
75: END
76:
77: DLG_ENTERPASSWORD DIALOG 30,71,193,93
78: STYLE DS_MODALFRAME | WS_POPUP |
79: WS_CAPTION | WS_SYSMENU
80: CAPTION "Enter Password"
81: FONT 8, "MS Sans Serif"
82: BEGIN
83:     LTEXT "The screen saver you are using
is password protected. You must type
in the screen saver password to turn
off the screen saver.",
84:     -1, 31, 7, 140, 40
85:     LTEXT "Password:", ID_PASSWORD,31,49,
86:     40, 14
87:     EDITTEXT ID_ETPASSWORD,71,49,80,14,
88:     ES_PASSWORD
89:     DEFPUSHBUTTON "OK", IDOK,31,70,40,14
90:     PUSHBUTTON "Cancel", IDCANCEL,111,70,
91:     40, 14
92: END

```

Die Datei »sstime.dlg« enthält die Definitionen der Dialogboxelemente.

Listing 4: sstime.rc

```

1: #include <windows.h>
2: #include <scrnsave.h>
3: #include "sstime.h"
4:
5: STRINGTABLE PRELOAD
6: BEGIN
7:     idsAppName "Screen Saver.Time"
8: END
9:
10: rcinclude sstime.dlg

```

Die Ressourcendatei »sstime.rc« definiert eine Zeichenkettentabelle und bindet die Dialogboxen mit ein. Wichtig ist hierbei der Eintrag *idsAppName*.

Listing 5: sstime.def

```

1: ; sstime.def
2: NAME SSTIME
3: DESCRIPTION 'SCRNSAVE : Time'
4: EXETYPE WINDOWS
5: STUB 'WINSTUB.EXE'
6: CODE PRELOAD MOVEABLE DISCARDABLE
7: DATA PRELOAD MOVEABLE MULTIPLE
8: HEAPSIZE 1024
9: STACKSIZE 8192
10: EXPORTS ScreenSaverProc
11:     ScreenSaverConfigureDialog
12:     DlgChangePassword
13:     DlgGetPassword
14:     DlgInvalidPassword
15:     HelpMessageFilterHookFunction

```

»sstime.def« exportiert die beiden Callback-Funktionen aus »sstime.c« und die Funktionen der Bibliothek »scrnsave.lib«.

Color Code: Mastermind für Windows

Sie kennen Mastermind? Dann kennen Sie auch Color Code. Dieser Mastermind-Klon ist vollständig in C++ mit Borlands Object Windows Library programmiert und zeigt sehr schön, wie Sie mit weniger als 700 Zeilen Programmtext ein ansprechendes Windows-Programm schreiben.

Bei dem Spiel Color Code (Bild 1) geht es darum, einen fünfstelligen Farbcode zu »knacken«. Sie müssen versuchen, aus den sieben oder neun Farben die fünf Farben des gesuchten Codes herauszufinden und diese in der richtigen Reihenfolge anzuordnen. Dazu wählen Sie jeweils eine Farbe auf dem Farbauswahlbalken aus und setzen einen entsprechenden Stein in die markierte Reihe. Doppelte Farben sind nicht zugelassen. Nachdem die Reihe vollständig besetzt wurde, wird mit dem Prüfen-Button ausgewertet, inwieweit die Reihe mit dem gesuchten Code übereinstimmt. Das Ergebnis wird in den Markierungsfeldern neben der Reihe ange-

zeigt: Ein weißer Markierungsstein steht für eine richtige Farbe an der falschen Position, ein schwarzer Stein für eine richtige Farbe an der richtigen Position. Welche Farbe oder Position genau gemeint ist, wird nicht angezeigt. Sie haben maximal zehn Versuche.

Steine werden gesetzt, indem Sie zunächst mit der linken Maustaste den entsprechenden Farbstein des Farbauswahlbalkens und dann ein Feld der markierten Reihe anklicken. Mit »[Alt-S]piel [N]eu« oder [F2] starten Sie das Spiel neu und ein neuer Rate-Code wird erzeugt. »[Alt-S]piel [A]ufdecken« oder [F3] deckt den Code auf und bricht das Spiel ab. »[Alt-S]piel [7] Farben« und »[Alt-S]piel [9] Farben« bestimmen die Farbanzahl, aus denen die Code-Farben ausgewählt werden.

Color Code (Listings 1 bis 4) wurde mit Borland C++ 3.1 für Windows 3.1 und Object Windows C++ geschrieben. Die verwendeten Grafiken sehen Sie in Bild 2. Zeichnen Sie diese ab, entwerfen Sie neue oder erwerben Sie die Databox — dort sind alle Dateien enthalten.

(Carsten Manshusen/bm)

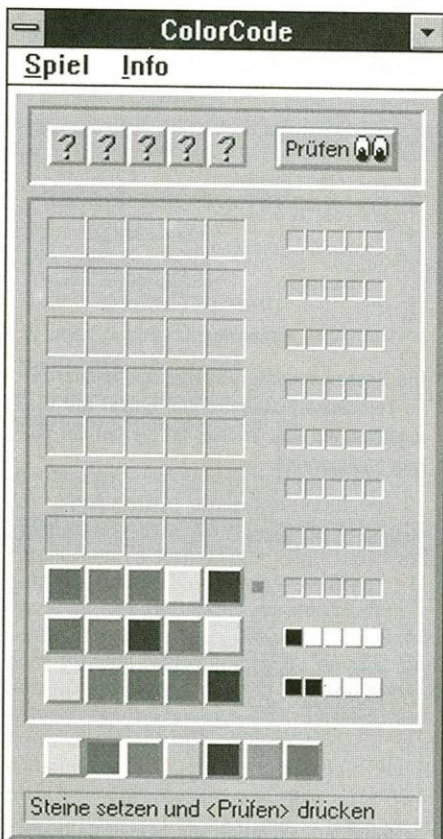


Bild 1. Mastermind im Windows-Gewand: Finden Sie die richtige Farbkombination?

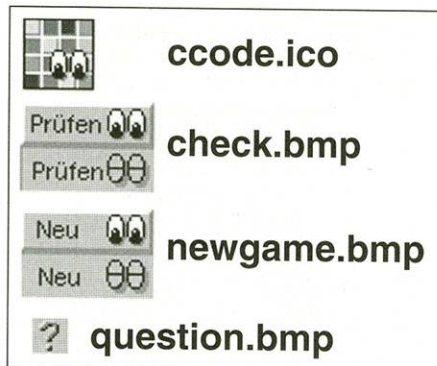


Bild 2. Das Icon und die drei verwendeten Bitmaps

Listing 1: ccode.cpp

```
1: // Ccode.cpp, BC 3.1
2: #include <owl.h>
3: #include <windows.h>
4: #include "ccoderc.h"
5:
6: // -- TColorCodeWin-Klasse --
7:
8: _CLASSDEF(TColorCodeWin)
9: class TColorCodeWin : public TWindow
10: {
11: public:
12:     TColorCodeWin();
13:     virtual ~TColorCodeWin();
14: protected:
15:     virtual LPSTR GetClassName()
16:     { return "ColorCodeWin"; }
17:     virtual void GetWindowClass(
18:         (WNDCLASS FAR &AWndClass);
19:     virtual void Paint(HDC PaintDC,
20:         PAINTSTRUCT FAR &PaintInfo);
21:     virtual void WMLButtonDown(RTMessage
22:         e Msg)
23:     = [WM_FIRST + WM_LBUTTONDOWN];
24:     virtual void WMMouseMove(RTMessage Msg)
25:     = [WM_FIRST + WM_MOUSEMOVE];
26:     virtual void WMLButtonUp(RTMessage Msg)
27:     = [WM_FIRST + WM_LBUTTONUP];
28:     virtual void WMInitMenu(RTMessage Msg)
29:     = [WM_FIRST + WM_INITMENU];
30:     virtual void CMNewGame(RTMessage Msg)
31:     = [CM_FIRST + CM_NEWGAME];
32:     virtual void CMUncover(RTMessage Msg)
33:     = [CM_FIRST + CM_UNCOVER];
34:     virtual void CMHighColor(RTMessage Msg)
35:     = [CM_FIRST + CM_HIGHLIGHT];
36:     virtual void CMInfo(RTMessage Msg)
37:     = [CM_FIRST + CM_INFO];
38: private:
39:     HPEN WhitePen;
40:     HPEN GrayPen;
41:     HBITMAP BMQuestion;
42:     HBITMAP BMCheck;
43:     HBITMAP BMNewGame;
44:     HFONT HelpFont;
45:     int ColorCode[5];
46:     int ColorFields[10][5];
47:     struct TCheck
48:     { int Black;
49:       int White; } Checks[10];
50:     int ColorCount;
51:     int ColorIndex;
52:     int LineIndex;
53:     int HelpIndex;
54:     BOOL ButtonPressed;
55:     static COLORREF CodeColors[9];
56:     static LPSTR HelpStr[6];
57:     void NewGame();
58:     void ButtonSelect();
59:     void SetHelpIndex(int AnIndex);
60:     void DrawButton(HDC DC, int X, int Y);
61:     void DrawCode(HDC DC, int X, int Y);
62:     void DrawColorBar(HDC DC, int X, int Y);
63:     void DrawField(HDC DC, int X, int Y,
64:         BOOL OutShade, int AColorIndex);
65:     void DrawCheckBar(HDC DC, int X, int Y,
66:         TCheck ACheck);
67:     void DrawHelpBox(HDC DC, int X1, int Y1,
68:         int X2, int Y2);
69:     void Box(HDC DC, int X, int Y,
70:         int Extent, int BorderWidth,
71:         BOOL OutShade, COLORREF FAR*Color);
72:     void Frame(HDC DC, int X1, int Y1,
73:         int X2, int Y2,
74:         int BorderWidth, BOOL OutShade);
75:     void Mark(HDC DC, int X, int Y);
76: };
77:
78:
79: // -- TColorCodeApp-Klasse --
80:
81: _CLASSDEF(TColorCodeApp)
82: class TColorCodeApp: public
83:     TApplication
84: {
85: public:
86:     TColorCodeApp(LPSTR AName,
87:         HINSTANCE hInstance,
88:         HINSTANCE hPrevInstance,
89:         LPSTR lpCmdLine, int nCmdShow)
90:         : TApplication(AName, hInstance,
91:             hPrevInstance,
92:             lpCmdLine, nCmdShow) {}
93: protected:
94:     virtual void InitInstance();
95:     virtual void InitMainWindow();
96: };
97:
98: // -- globale Variablen --
99:
100: COLORREF TColorCodeWin::CodeColors[9]=
101: {0x0000FFFF, 0x000000FF, 0x00FF00FF,
102: 0x00FFFF00, 0x00FF0000, 0x0000FF00,
103: 0x00008000, 0x00FF0080, 0x00000080};
104: LPSTR TColorCodeWin::HelpStr[6] =
105: {"Steine setzen und <Prüfen>
106: drücken",
107: "Fehler: Alle Felder d Reihe belegen",
108: "Fehler: J. Farbe nur einmal setzen",
109: "Bravo, Super, Toll, Klasse !",
110: "Nicht weinen..."}
```

»ccode.cpp« ist die Neuauflage des bekannten Mastermind-Spiels.


```

109: "Spielverderber, Spielverderber...";
110:
111: // -TColorCodeWin-Elementfunktionen-
112:
113: TColorCodeWin::TColorCodeWin()
114: {
115:     TWindow(NULL, "ColorCode")
116: {
117:     Attr.X = 50;
118:     Attr.Y = 20;
119:     Attr.W = 222;
120:     Attr.H = 421;
121:     Attr.Style = ~(WS_THICKFRAME |
122:                     WS_MAXIMIZEBOX);
123:     AssignMenu(MEN_COLORCODE);
124:     WhitePen = GetStockPen(WHITE_PEN);
125:     GrayPen = CreatePen(PS_SOLID, 1,
126:                         RGB(128, 128, 128));
127:     BMQuestion =
128: LoadBitmap(GetApplication()->hInstance,
129:             MAKEINTRESOURCE(BM_QUESTION));
130:     BMCheck =
131: LoadBitmap(GetApplication()->hInstance,
132:             MAKEINTRESOURCE(BM_CHECK));
133:     BMNewGame =
134: LoadBitmap(GetApplication()->hInstance,
135:             MAKEINTRESOURCE(BM_NEWGAME));
136:     LOGFONT LogFont;
137:     memset(&LogFont, 0, sizeof(LOGFONT));
138:     LogFont.lfHeight = 8;
139:     LogFont.lfWeight = FW_NORMAL;
140:     lstrcpy((LPSTR)LogFont.lfFaceName,
141:             "MS Sans Serif");
142:     HelpFont = CreateFontIndirect(&LogFont);
143:     ColorCount = 7;
144:     ColorIndex = 0;
145:     ButtonPressed = FALSE;
146:     randomize();
147:     NewGame();
148: }
149: TColorCodeWin::~TColorCodeWin()
150: {
151:     DeleteFont(HelpFont);
152:     DeleteBitmap(BMNewGame);
153:     DeleteBitmap(BMCheck);
154:     DeleteBitmap(BMQuestion);
155:     DeletePen(GrayPen);
156: }
157:
158: void TColorCodeWin::GetWindowClass
159: (WNDCLASS FAR & AwndClass)
160: {
161:     TWindow::GetWindowClass(AwndClass);
162:     AwndClass.hIcon =
163: LoadIcon(GetApplication()->hInstance,
164:           MAKEINTRESOURCE(ICO_COLORCODE));
165:     AwndClass.hbrBackground =
166: GetStockBrush(LTGRAY_BRUSH);
167: }
168:
169: void TColorCodeWin::Paint(HDC PaintDC,
170:                           PAINTSTRUCT FAR & PaintInfo)
171: {
172:     TWindow::Paint(PaintDC, PaintInfo);
173:     RECT ClientRect;
174:     GetClientRect(HWindow, &ClientRect);
175:     Frame(PaintDC, 0, 0, ClientRect.right,
176:           ClientRect.bottom, 4, TRUE);
177:     Frame(PaintDC, 10, 10, 210, 50,
178:           2, FALSE);
179:     DrawCode(PaintDC, 20, 20);
180:     DrawButton(PaintDC, 134, 19);
181:     Frame(PaintDC, 10, 56, 210, 321,
182:           2, FALSE);
183:     for (int Ln = 0; Ln < 10; Ln++)
184:     {
185:         int Y = 66 + Ln * 25;
186:         for (int Fd = 0; Fd < 5; Fd++)
187:         {
188:             int Index = ColorFields[Ln][Fd];
189:             DrawField(PaintDC, 20 + Fd * 20,
190:                       Y, Index >= 0, Index);
191:         }
192:         DrawCheckBox(PaintDC, 140,
193:                       71 + Ln * 25, Checks[Ln]);
194:     }
195:     if (LineIndex >= 0)
196:         Mark(PaintDC, 124,
197:               73 + LineIndex * 25);
198:     DrawColorBar(PaintDC, 20, 328);
199:     DrawHelpBox(PaintDC, 10, 355, 210, 371);
200: }
201:
202: void TColorCodeWin::WMLButtonDown
203: (RTMessage Msg)
204: {
205:     POINT MousePos = MAKEPOINT(Msg.LParam);
206:     RECT ButtonRect;
207:     ButtonRect.left = 134;
208:     ButtonRect.top = 19;
209:     ButtonRect.right = 196;
210:     ButtonRect.bottom = 41;
211:     if (PtInRect(&ButtonRect, MousePos))
212:     {
213:         ButtonPressed = TRUE;
214:         HDC DC = GetDC(HWindow);
215:         DrawButton(DC, 134, 19);
216:         ReleaseDC(HWindow, DC);
217:         SetCapture(HWindow);
218:     }

```

```

219:     else if (MousePos.y >= 328 &&
220:             MousePos.y < 328 + 20)
221:     {
222:         int NewIndex = (MousePos.x - 20) / 20;
223:         if (NewIndex != ColorIndex &&
224:             NewIndex < ColorCount)
225:         {
226:             ColorIndex = NewIndex;
227:             HDC DC = GetDC(HWindow);
228:             DrawColorBar(DC, 20, 328);
229:             ReleaseDC(HWindow, DC);
230:         }
231:     }
232:     else if (LineIndex >= 0)
233:     {
234:         int LineYPos = 66 + LineIndex * 25;
235:         if (MousePos.y >= LineYPos &&
236:             MousePos.y < LineYPos + 20)
237:         {
238:             int FieldIndex =
239: (MousePos.x - 20) / 20;
240:             if (FieldIndex < 5)
241:             {
242:                 ColorFields[LineIndex][FieldIndex]
243:                 = ColorIndex;
244:                 HDC DC = GetDC(HWindow);
245:                 DrawField(DC, 20 + FieldIndex * 20,
246:                           LineYPos, TRUE, ColorIndex);
247:                 ReleaseDC(HWindow, DC);
248:                 SetHelpIndex(0);
249:             }
250:         }
251:     }
252: }
253:
254: void TColorCodeWin::WMMouseMove
255: (RTMessage Msg)
256: {
257:     if (GetCapture() == HWindow)
258:     {
259:         RECT ButtonRect;
260:         ButtonRect.left = 134;
261:         ButtonRect.top = 19;
262:         ButtonRect.right = 196;
263:         ButtonRect.bottom = 41;
264:         BOOL Pressed =
265: PtInRect(&ButtonRect,
266:           MAKEPOINT(Msg.LParam));
267:         if (Pressed != ButtonPressed)
268:         {
269:             ButtonPressed = Pressed;
270:             HDC DC = GetDC(HWindow);
271:             DrawButton(DC, 134, 19);
272:             ReleaseDC(HWindow, DC);
273:         }
274:     }
275: }
276:
277: void TColorCodeWin::WMLButtonUp
278: (RTMessage Msg)
279: {
280:     if (GetCapture() == HWindow)
281:     {
282:         ReleaseCapture();
283:         if (ButtonPressed)
284:         {
285:             ButtonPressed = FALSE;
286:             HDC DC = GetDC(HWindow);
287:             DrawButton(DC, 134, 19);
288:             ReleaseDC(HWindow, DC);
289:             ButtonSelect();
290:         }
291:     }
292: }
293:
294: void TColorCodeWin::WMInitMenu
295: (RTMessage Msg)
296: {
297:     EnableMenuItem((HMENU)Msg.WParam,
298:                   CM_UNCOVER,
299:                   (LineIndex >= 0) ?
300: MF_ENABLED : MF_GRAYED);
301:     CheckMenuItem((HMENU)Msg.WParam,
302:                   CM_LOWCOLOR,
303:                   (ColorCount == 7) ?
304: MF_CHECKED : MF_UNCHECKED);
305:     CheckMenuItem((HMENU)Msg.WParam,
306:                   CM_HIGHCOLOR,
307:                   (ColorCount == 9) ?
308: MF_CHECKED : MF_UNCHECKED);
309: }
310:
311: void TColorCodeWin::CMNewGame
312: (RTMessage Msg)
313: {
314:     NewGame();
315: }
316:
317: void TColorCodeWin::CMUncover
318: (RTMessage Msg)
319: {
320:     if (LineIndex >= 0)
321:     {
322:         HDC DC = GetDC(HWindow);
323:         Mark(DC, 124, 73 + LineIndex * 25);
324:         LineIndex = -1;
325:         DrawCode(DC, 20, 20);
326:         DrawButton(DC, 134, 19);
327:         ReleaseDC(HWindow, DC);
328:         SetHelpIndex(5);

```

```

329:     }
330: }
331:
332: void TColorCodeWin::CMLowColor
333: (RTMessage Msg)
334: {
335:     ColorCount = 7;
336:     ColorIndex = 0;
337:     NewGame();
338: }
339:
340: void TColorCodeWin::CMHighColor
341: (RTMessage Msg)
342: {
343:     ColorCount = 9;
344:     ColorIndex = 0;
345:     NewGame();
346: }
347:
348: void TColorCodeWin::CMInfo
349: (RTMessage Msg)
350: {
351:     GetApplication()->
352: ExecDialog(new TDialog(this,
353:                           DLG_INFO));
354: }
355:
356: void TColorCodeWin::NewGame()
357: {
358:     int NumArray[9] =
359: {0, 1, 2, 3, 4, 5, 6, 7, 8};
360:     int MaxNum = ColorCount;
361:     for (int i = 0; i < 5; i++)
362:     {
363:         int Index = random(MaxNum);
364:         ColorCode[i] = NumArray[Index];
365:         MaxNum--;
366:         for (int j = Index; j < MaxNum; j++)
367:             NumArray[j] = NumArray[j + 1];
368:     }
369:     for (int Ln = 0; Ln < 10; Ln++)
370:         for (int Fd = 0; Fd < 5; Fd++)
371:         {
372:             ColorFields[Ln][Fd] = -1;
373:             Checks[Ln].Black = 0;
374:             Checks[Ln].White = 0;
375:         }
376:     LineIndex = 9;
377:     HelpIndex = 0;
378:     if (HWindow)
379:         InvalidateRect(HWindow, NULL, TRUE);
380: }
381:
382: void TColorCodeWin::ButtonSelect()
383: {
384:     if (LineIndex >= 0)
385:     {
386:         Paint CheckCode =
387: &ColorFields[LineIndex][0];
388:         for (int i = 0; i < 5; i++)
389:             if (CheckCode[i] == -1)
390:             {
391:                 SetHelpIndex(1);
392:                 MessageBeep(0);
393:                 return;
394:             }
395:         int IndexCount[9];
396:         for (i = 0; i < 9; i++)
397:             IndexCount[i] = 0;
398:         for (i = 0; i < 5; i++)
399:             if (++IndexCount[CheckCode[i]] > 1)
400:             {
401:                 SetHelpIndex(2);
402:                 MessageBeep(0);
403:                 return;
404:             }
405:         for (i = 0; i < 5; i++)
406:             for (int j = 0; j < 5; j++)
407:                 if (ColorCode[i] == CheckCode[j])
408:                 {
409:                     if (i == j)
410:                         Checks[LineIndex].Black++;
411:                     else
412:                         Checks[LineIndex].White++;
413:                 }
414:                 break;
415:         HDC DC = GetDC(HWindow);
416:         DrawCheckBox(DC, 140,
417:                       71 + LineIndex * 25,
418:                       Checks[LineIndex]);
419:         Mark(DC, 124, 73 + LineIndex * 25);
420:         if (Checks[LineIndex].Black == 5)
421:         {
422:             LineIndex = -1;
423:             SetHelpIndex(3);
424:         }
425:         else if (--LineIndex >= 0)
426:             Mark(DC, 124, 73 + LineIndex * 25);
427:         else
428:             SetHelpIndex(4);
429:         if (LineIndex < 0)
430:         {
431:             DrawCode(DC, 20, 20);
432:             DrawButton(DC, 134, 19);
433:         }

```

»ccode.cpp« ist die Neuauflage des bekannten Mastermind-Spiels.

WINDOWS-PROGRAMMIERUNG

Denk- und Strategiespiel für Windows

```

434: ReleaseDC(HWindow, DC);
435: }
436: else
437:     NewGame();
438: }
439:
440: void TColorCodeWin::SetHelpIndex
441:     (int AnIndex)
442: {
443:     if (HelpIndex != AnIndex)
444:     {
445:         HelpIndex = AnIndex;
446:         if (HWindow)
447:         {
448:             HDC DC = GetDC(HWindow);
449:             DrawHelpBox(DC, 10, 355, 210, 371);
450:             ReleaseDC(HWindow, DC);
451:         }
452:     }
453: }
454:
455: void TColorCodeWin::DrawButton(HDC DC,
456:     int X, int Y)
457: {
458:     RECT ButtonRect;
459:     ButtonRect.left = X;
460:     ButtonRect.top = Y;
461:     ButtonRect.bottom = X + 62;
462:     ButtonRect.right = Y + 22;
463:     FrameRect(DC, &ButtonRect,
464:         GetStockBrush(GRAY_BRUSH));
465:     HDC MemDC = CreateCompatibleDC(DC);
466:     SelectBitmap(MemDC, LineIndex >= 0 ?
467:         BMCheck : BMNewGame);
468:     BitBlt(DC, X + 1, Y + 1, 60, 20, MemDC,
469:         0, ButtonPressed ? 20 : 0,
470:         SRCCOPY);
471:     DeleteDC(MemDC);
472: }
473:
474: void TColorCodeWin::DrawCode(HDC DC,
475:     int X, int Y)
476: {
477:     HDC MemDC;
478:     if (LineIndex >= 0)
479:     {
480:         MemDC = CreateCompatibleDC(DC);
481:         SelectBitmap(MemDC, BMQuestion);
482:     }
483:     for (int i = 0; i < 5; i++)
484:     {
485:         int XPos = X + i * 20;
486:         Box(DC, XPos, Y, 20, 2, TRUE,
487:             (LineIndex >= 0) ?
488:             NULL : &CodeColors[ColorCode[i]]);
489:         if (LineIndex >= 0)
490:             BitBlt(DC, XPos + 2, Y + 2, 18, 18,
491:                 MemDC, 0, 0, SRCCOPY);
492:     }
493:     if (LineIndex >= 0)
494:         DeleteDC(MemDC);
495: }
496:
497: void TColorCodeWin::DrawColorBar(
498:     HDC DC, int X, int Y)
499: {
500:     for (int i = 0; i < ColorCount; i++)
501:         Box(DC, X + i * 20, Y, 20,
502:             2, i != ColorIndex,
503:             &CodeColors[i]);
504: }
505:
506: void TColorCodeWin::DrawField(HDC DC,
507:     int X, int Y,
508:     BOOL OutShade, int AColorIndex)
509: {
510:     int BorderWidth;
511:     COLORREF_FAR * Color;
512:     if (AColorIndex >= 0)
513:     {
514:         BorderWidth = 2;
515:         Color = &CodeColors[AColorIndex];
516:     }
517:     else
518:     {
519:         BorderWidth = 1;
520:         Color = NULL;
521:     }
522:     Box(DC, X, Y, 20, BorderWidth,
523:         OutShade, Color);
524: }
525:
526: void TColorCodeWin::DrawCheckBar(
527:     HDC DC, int X, int Y, TCheck ACheck)
528: {
529:     for (int i = 0; i < 5; i++)
530:     {
531:         COLORREF CheckColor;
532:         COLORREF_FAR * Color;
533:         if (ACheck.Black)
534:         {
535:             CheckColor = RGB(0, 0, 0);
536:             Color = &CheckColor;
537:             ACheck.Black--;
538:         }
539:         else if (ACheck.White)
540:         {
541:             CheckColor = RGB(255, 255, 255);
542:             Color = &CheckColor;
543:             ACheck.White--;

```

```

544:         }
545:         else
546:             Color = NULL;
547:         Box(DC, X + i * 10, Y, 10, 1,
548:             Color != NULL, Color);
549:     }
550: }
551:
552: void TColorCodeWin::DrawHelpBox(HDC DC,
553:     int X1, int Y1, int X2, int Y2)
554: {
555:     Frame(DC, X1, Y1, X2, Y2, 1, FALSE);
556:     RECT TextRect;
557:     TextRect.left = X1 + 1;
558:     TextRect.top = Y1 + 1;
559:     TextRect.right = X2 - 1;
560:     TextRect.bottom = Y2 - 1;
561:     LPSTR HelpText = HelpStr[HelpIndex];
562:     HFONT OldFont = SelectFont(DC, Help
563:         Font);
564:     COLORREF OldBkColor =
565:         SetBkColor(DC, RGB(192, 192, 192));
566:     ExtTextOut(DC, TextRect.left + 2,
567:         TextRect.top,
568:         ETO_CLIPPED | ETO_OPAQUE,
569:         &TextRect, HelpText,
570:         lstrlen(HelpText), NULL);
571:     SetBkColor(DC, OldBkColor);
572:     SelectFont(DC, OldFont);
573: }
574:
575: void TColorCodeWin::Box(HDC DC, int X,
576:     int Y, int Extent, int BorderWidth,
577:     BOOL OutShade, COLORREF_FAR * Color)
578: {
579:     int X2 = X + Extent;
580:     int Y2 = Y + Extent;
581:     Frame(DC, X, Y, X2, Y2,
582:         BorderWidth, OutShade);
583:     if (Color)
584:     {
585:         RECT ColorRect;
586:         ColorRect.left = X + BorderWidth;
587:         ColorRect.top = Y + BorderWidth;
588:         ColorRect.right = X2 - BorderWidth;
589:         ColorRect.bottom = Y2 - BorderWidth;
590:         HBRUSH Brush =
591:             CreateSolidBrush(*Color);
592:         FillRect(DC, &ColorRect, Brush);
593:         DeleteBrush(Brush);
594:     }
595: }
596:
597: void TColorCodeWin::Frame(HDC DC,
598:     int X1, int Y1, int X2, int Y2,
599:     int BorderWidth, BOOL OutShade)
600: {
601:     X2--;
602:     Y2--;
603:     HPEN OldPen =
604:         SelectPen(DC, OutShade ?
605:             WhitePen : GrayPen);
606:     for (int i = 0; i < BorderWidth; i++)
607:     {
608:         MoveTo(DC, X1 + i, Y2 - i);
609:         LineTo(DC, X1 + i, Y1 + i);
610:         LineTo(DC, X2 - i, Y1 + i);
611:     }
612:     SelectPen(DC, OutShade ?
613:         GrayPen : WhitePen);
614:     for (i = 0; i < BorderWidth; i++)
615:     {
616:         MoveTo(DC, X2 - i, Y1 + i);
617:         LineTo(DC, X2 - i, Y2 - i);
618:         LineTo(DC, X1 + i, Y2 - i);
619:     }
620:     SelectPen(DC, OldPen);
621: }
622:
623: void TColorCodeWin::Mark(HDC DC,
624:     int X, int Y)
625: {
626:     RECT MarkRect;
627:     MarkRect.left = X;
628:     MarkRect.top = Y;
629:     MarkRect.right = X + 6;
630:     MarkRect.bottom = Y + 6;
631:     InvertRect(DC, &MarkRect);
632: }
633:
634: // -TColorCodeApp-Elementfunktionen-
635:
636: void TColorCodeApp::InitInstance()
637: {
638:     TApplication::InitInstance();
639:     HACCTable =
640:         LoadAccelerators(hInstance,
641:             MAKEINTRESOURCE(ACC_COLORCODE));
642: }
643:
644: void TColorCodeApp::InitMainWindow()
645: {
646:     MainWindow = new TColorCodeWin;
647: }
648:
649: // -- WinMain-Funktion --
650:
651: int PASCAL WinMain(HINSTANCE hInstance,
652:     HINSTANCE hPrevInstance,
653:     LPSTR lpCmdLine, int nCmdShow)
654: {

```

```

654: TColorCodeApp ColorCode(NULL,
655:     hInstance, hPrevInstance,
656:     lpCmdLine, nCmdShow);
657: ColorCode.Run();
658: return ColorCode.Status;
659: }

```

»ccode.cpp« ist die Neuauflage des bekannten Mastermind-Spiels.

Listing 2: ccode.rc

```

1: // ccode.rc, BC31
2:
3: #include "ccoderc.h"
4: #include "owlrc.h"
5:
6: // -- Bitmaps --
7:
8: BM_CHECK BITMAP "check.bmp"
9: BM_QUESTION BITMAP "question.bmp"
10: BM_NEWGAME BITMAP "newgame.bmp"
11:
12: // -- Icon --
13:
14: ICO_COLORCODE ICON "ccode.ico"
15:
16: // -- Menü --
17:
18: MEN_COLORCODE MENU
19: BEGIN
20:     POPUP "&Spiel"
21:     BEGIN
22:         MENUITEM "&Neu\&F2", CM_NEWGAME
23:         MENUITEM "&Aufdecken\&F3", CM_UNCOVER
24:         MENUITEM SEPARATOR
25:         MENUITEM "&7 Farben", CM_LOWCOLOR
26:         MENUITEM "&9 Farben", CM_HIGHCOLOR
27:         MENUITEM SEPARATOR
28:         MENUITEM "&Beenden", CM_EXIT
29:     END
30:     POPUP "&Info"
31:     BEGIN
32:         MENUITEM "&Info...", CM_INFO
33:     END
34: END
35:
36: // -- Menü -Beschleuniger --
37:
38: ACC_COLORCODE ACCELERATORS
39: BEGIN
40:     VK_F2, CM_NEWGAME, VIRTKEY
41:     VK_F3, CM_UNCOVER, VIRTKEY
42: END
43:
44: // -- Info-Dialog --
45:
46: DLG_INFO DIALOG 30, 59, 139, 85
47: STYLE DS_MODALFRAME | WS_POPUP |
48:     WS_CAPTION | WS_SYSMENU
49: CAPTION "Info über ColorCode"
50: FONT 8, "MS Sans Serif"
51: BEGIN
52:     ICON ICO_COLORCODE, -1, 37, 8, 18, 20,
53:     WS_CHILD | WS_VISIBLE
54:     LTEXT "ColorCode", -1, 61, 15, 36, 8,
55:     WS_CHILD | WS_VISIBLE | WS_GROUP
56:     CTEXT "von Carsten Manshusen", -1, 28,
57:     35, 83, 8, WS_CHILD | WS_VISIBLE |
58:     WS_GROUP
59:     CONTROL "\251 1993 DMV GmbH & Co."
60:     "KG", -1, "STATIC", SS_CENTER |
61:     SS_NOPREFIX | WS_CHILD | WS_VISIBLE |
62:     WS_GROUP, 6, 45, 127, 8
63:     DEFPUSHBUTTON "OK", IDOK, 50, 62, 38, 14,
64:     WS_CHILD | WS_VISIBLE | WS_TABSTOP
65: END

```

Die Datei »ccode.rc« enthält die Ressourcen für Color Code.

Listing 3: ccode.h

```

1: // ccode.h, BC 31
2: #define BM_CHECK 1000
3: #define BM_NEWGAME 1001
4: #define BM_QUESTION 1002
5: #define ICO_COLORCODE 100
6: #define MEN_COLORCODE 100
7: #define ACC_COLORCODE 100
8: #define CM_NEWGAME 101
9: #define CM_UNCOVER 102
10: #define CM_LOWCOLOR 103
11: #define CM_HIGHCOLOR 104
12: #define CM_INFO 111
13: #define DLG_INFO 100

```

Die Headerdatei »ccode.h« definiert einige Werte.

Kesse Kurven

Windows eignet sich sehr gut, um Zusammenhänge grafisch mit Kurven darzustellen. Deren Programmierung ist eine aufwendige Kleinarbeit, die Sie nur einmal in ein geeignet definiertes Objekt investieren sollten.

Ein solches Objekt ist *TGrafOWnd*: Es gibt Kurven in einem »Universal-Fenster« *TWindow* aus, wobei die Anzahl der Kurven, die Anzahl der Werte pro Kurve, die Kurvenfarben sowie Ort und Größe der Kurven vom übergeordneten Programm beliebig gewählt werden können.

Die Methoden von *TGrafOWnd* sind so flexibel, daß sowohl eine feste Anzahl von Werten (zum Beispiel aus einer Datei) für eine »statische« Darstellung als auch die Ergebnisse einer laufenden Berechnung oder Messung nacheinander in Echtzeit dargestellt werden können (Bild).

Im Hinblick auf den letzteren Fall wurde bei der Programmierung großer Wert auf hohe Ausgabegeschwindigkeit gelegt, insbesondere werden in der laufenden Grafik nur die zuletzt zugefügten Werte nachgetragen, solange das Fenster nicht von außen geändert wurde. Die Art der Bildänderung wird dem übergeordneten Programm mitgeteilt, das damit alle nur die Kurven betreffenden Paint-Botschaften ignorieren kann.

Beispielsweise wird das Demoprogramm durch diese Optimierungen um das knapp Fünffache beschleunigt, wenn jeweils ein Wert pro Kurve eingetragen wird.

Der Objekttyp *TFastAppl*

Wenn Sie *TGrafOWnd* zur Darstellung von Berechnungen nutzen, brauchen Sie einen Timer, der die Berechnungen anstößt. Dabei ist es häufig von Nachteil, daß der kleinste Abstand zwischen zwei Timersignalen 55 Millisekunden beträgt, insbesondere wenn Sie die Rechnung in viele kurze Teile zerlegen wollen, die das »kooperative Multitasking« von Windows verlangt.

TFastAppl ist ein Objekttyp, der die Nachrichtenschleife von *TApplication* überschreibt und bei jedem Schleifendurchlauf eine Timerbotschaft an das Fenster schickt, wenn *NextMsg=True* ist. Durch Überschreiben der Funktion *NextMsg: Bool; virtual*; läßt sich das Tempo feiner den Erfordernissen anpassen, als es mit einem Timer machbar ist.

Zur Unterscheidung von »echten« Timerbotschaften steht im Parameter *wParam* der Timerbotschaft von *TFastAppl* der

Die Felder von *TGrafOWnd*

YMax: Word; Darstellungsbereich der Grafik von 0 bis YMax (Vorbelegung 1000)

CalcNum: LongInt; laufende Nummer des letzten Wertes (1,2,...)

PaintAll: Bool; Im Zustand *False* kann das übergeordnete Programm *wm_Paint*-Meldungen ignorieren, solange es selber das Fenster nicht geändert hat.

AktAll: Bool; Durch Setzen auf *True* wird ein Neuzeichnen aller Kurven erzwungen.

RAI: TRect; Rechteck der Grafik

IndWert: Integer; Zeiger auf den jüngsten Wert in jedem Kurvenringpuffer (1 bis *MaxWert*, 0 bei leerem Puffer)

AnzWert: Integer; Anzahl der Werte in jedem Ringpuffer (1,2,...), bei übergelaufenem Puffer *MaxWert+1*

Globale Variablen (werden vor dem Init-Aufruf gesetzt)

MaxKurv: Word; Anzahl der Kurven (1,2,...)

MaxWert: Word; Anzahl der Werte pro Kurve ohne den Randwert (siehe Bemerkung), muß > 2 sein

PtFarbe: PFBarr; Zeiger auf ein Array mit den Farbwerten der Kurven im übergeordneten Programm

Die Methoden von *TGrafOWnd*

Procedure IniCalc; virtual; initialisiert die Steuergrößen

Procedure SetGrafRect (var NeuRect: TRect); virtual; setzt Ort und Größe der Grafik auf *NeuRect*

Procedure InvalidateGraf; virtual; stößt die Darstellung der zuvor mit *SetNeuWert* übergebenen Werte in der Grafik an.

Procedure SetNeuWert (PtNeuWert: PWDarr); virtual; übergibt die Werte, die in einem *Array[0..MaxKurv-1]* stehen, auf das *PtNeuWert* zeigt, an die Grafik als jeweils letzten Kurvenwert. Eine (beliebig lange!) Folge von *SetNeuWert*-Aufrufen muß mit dem einmaligen Aufruf von *InvalidateGraf* abgeschlossen werden, um die Werte grafisch darzustellen.

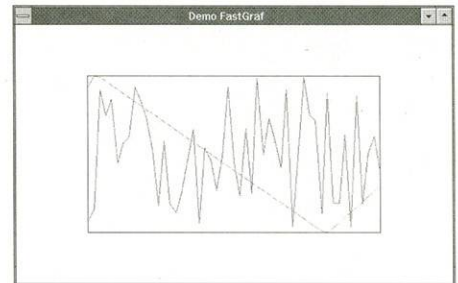
Procedure DrawCurve (PaintDC, ix0,ix1: Integer); zeichnet mit dem übergebenen Displaykontext alle Kurven von *ix0* bis *ix1* (*ix0,ix1=0..MaxWert*)

Bemerkungen:

1. Die Grafik schließt rechts mit dem Wert *MaxWert* ab und beginnt links mit einem Randwert, der zu Anfang gleich dem ersten und nach Pufferüberlauf gleich dem letzten Wert ist.

2. Bei Pufferüberlauf werden die neuen Werte wieder links beginnend dargestellt. Zwischen dem letzten neuen und dem ersten alten Wert wird zur Unterscheidung eine »Lücke« von einem Wert gelassen.

3. Der Speicher für die Kurvenwerte wird dynamisch zugewiesen. Reicht er nicht aus, bricht das Programm mit einer Messagebox ab.



Fix: Diese Kurven flitzen nur so über den Bildschirm.

Wert *VonFastAppl*.

Das Beispielprogramm

Die Unit »fastgraf.pas« (Listing 1) wird von dem Beispielprogramm »fgdemo.pas« (Listing 2) verwendet und setzt die beschriebenen Objekte ein, um zwei Kurven mit je 50 Zufallswerten fortlaufend auszugeben. Mit der rechten Maustaste läßt sich die Grafikausgabe unterbrechen.

Beachten Sie, daß in *Fgdemo* der Rahmen nur gezeichnet werden muß, wenn *PaintAll=True* ist.

Listing 1: fastgraf.pas

```
1: { FastGraf.pas, TPW }
2: Unit FastGraf;
3: {SA+,D-,G+,I-,L-,R-,S-,V-,W-,X+}
4: INTERFACE
5:
6: uses
7:   WObjects, WinTypes, WinProcs;
8:
9: type
10:   WordArr=array[0..32000] of Word;
11:   PointArr=array[0..16000] of TPoint;
12:   FarbArr=array[0..16000] of TColorRef;
13:   PenArr = array[0..32000] of HPen;
14:   PWDarr = ^WordArr;
15:   PPointArr = ^PointArr;
16:   PFBarr = ^FarbArr;
17:   PPnArr = ^PenArr;
18:
19: const
20:   CMaxHeap = 65520;
21:   Reserve = 32*1024;
22:   UnitNam = 'Unit FastGraf'#0;
23:   MemErr =
24:     'Nicht genügend Speicher frei !'#0;
25:   VonFastAppl[Alt 9]= 4713;
26:   MaxKurv: Word = 1;
27:   MaxWert: Word = 10;
28:
29: var
30:   DasWh: HWnd;
31:   PtFarbe: PFBarr;
32:
33: type
34:   PGrafOWnd = ^TGrafOWnd;
35:   TGrafOWnd = object (TWindow)
36:
37:     YMax: Word;
38:     CalcNum: LongInt;
39:     PaintAll,AktAll: Bool;
40:     RAI: TRect;
41:     AnzWert,IndWert: Integer;
42:     PtAllBuf: PWDarr;
43:
44:     KvLen: Word;
45:     MemoryDa: Bool;
46:     IAKtAnf,IAktPrv: Integer;
47:     RAKt: TRect;
48:     dx,dy: Real;
49:     PtPoints: PPointArr;
50:     PtStifte: PPnArr;
51:     SizeAllBuf,SizePoints,
52:     SizeStifte,SizeGesamt: LongInt;
53:
54:     {- geerbte Methoden -}
55:     constructor Init
56:       (AParent: TWindowsObject;
57:        ATitle: PChar);
58:     function Create: Boolean; virtual;
59:     procedure DefWndProc
60:       (var Msg: TMessage); virtual;
```

»fastgraf.pas« enthält die Methoden für eine schnelle Grafikausgabe.


```

61: procedure Paint (PaintDC: HDC; var
62: PaintInfo: TPaintStruct); virtual;
63: destructor Done; virtual;
64:
65: {- neue Methoden -}
66: procedure IniCalc; virtual;
67: procedure SetGrafRect
68: (var NeuRect: TRect); virtual;
69: procedure SetNeuWert
70: (PtNeuWert: PWDArray); virtual;
71: procedure InvalidateGraf; virtual;
72: procedure DrawCurve (PaintDC: HDC;
73: ix0,ix1: Integer);
74: end;
75:
76: type
77: TFastAppl = object (TApplication)
78: procedure MessageLoop; virtual;
79: function NextMsg: Bool; virtual;
80: end;
81:
82: IMPLEMENTATION
83:
84: { METHODEN VON TGrafOWnd }
85:
86: { Initialisierung / Abbruch }
87: constructor TGrafOWnd.Init
88: (AParent: PWindowsObject; ATitle: PChar);
89: var
90: i: Word;
91: begin
92: TWindow.Init (AParent, ATitle);
93:
94:
95: Rall.left := 10;
96: Rall.right := 10;
97: Rall.top := 10;
98: Rall.bottom := 10;
99: Rakt := Rall;
100: YMax := 1000;
101: KvLen := MaxWert + 1;
102:
103: {- Speicher reservieren -}
104: SizeAllBuf := KvLen;
105: SizeAllBuf :=
106: SizeAllBuf * MaxKurv * SizeOf (Word);
107: SizePoints := KvLen * SizeOf (TPoint);
108: SizeStifte := MaxKurv * SizeOf (HPen);
109: SizeGesamt := SizeAllBuf + SizePoints
110: + SizeStifte;
111: MemoryDa := False;
112: if SizeGesamt < CMaxHeap then begin
113: if SizeGesamt < MemAvail - Reserve
114: then begin
115: MemoryDa := True;
116: GetMem (PtAllBuf, Word (SizeAllBuf));
117: GetMem (PtPoints, Word (SizePoints));
118: GetMem (PtStifte, Word (SizeStifte));
119: end;
120: end;
121:
122: {- Abbruch, wenn Speicher fehlt -}
123: if not MemoryDa then begin
124: MessageBox (0, MemErr, UnitNam,
125: mb_ok or mb_IconExclamation);
126: PostQuitMessage (0);
127: end;
128:
129: {- Farbstifte, Rechnung init. -}
130: if MemoryDa then begin
131: for i := 0 to MaxKurv-1 do
132: PtStifte[i] := CreatePen
133: [Alt 9] (ps_Solid, 1, PtFarbe[i]);
134: IniCalc;
135: end;
136: end;
137:
138: procedure TGrafOWnd.IniCalc;
139: var
140: i: Word;
141: begin
142: IndWert := 0;
143: AnzWert := 0;
144: CalcNum := 0;
145: IaktAnf := -1;
146: AktAll := True;
147: PaintAll := True;
148: end;
149:
150: function TGrafOWnd.Create: Boolean;
151: begin
152: Create := TWindow.Create;
153: DasWh := HWindow;
154: end;
155:
156: destructor TGrafOWnd.Done;
157: var
158: i: Word;
159: begin
160: if MemoryDa then begin
161: for i := 0 to MaxKurv-1 do
162: DeleteObject (PtStifte[i]);
163: FreeMem (PtAllBuf, Word (SizeAllBuf));
164: FreeMem (PtPoints, Word (SizePoints));
165: FreeMem (PtStifte, Word (SizeStifte));
166: end;
167: TWindow.Done;
168: end;
169:
170: { Message-Verarbeitung }
171: procedure TGrafOWnd.DefWndProc
172: (var Msg: TMessage);
173: begin
174: case Msg.Message of
175: {- Bild von aussen geandert -}
176: 136, WM_NCPaint:
177: begin
178: AktAll := True;
179: PaintAll := True;
180: InvalidateRect (HWindow, @Rall, TRUE);

```

```

181: end;
182: end;
183: {- Standard-Verarbeitung -}
184: TWindow.DefWndProc (Msg);
185: end;
186:
187: { Paint-Routine }
188: procedure TGrafOWnd.Paint (PaintDC: HDC;
189: var PaintInfo: TPaintStruct);
190: var
191: indl, ind0: Integer;
192: label
193: Ende;
194: begin
195:
196: {- alles neu zu zeichnen ? -}
197: if AktAll then begin
198: {- Skalenteile berechnen -}
199: dx := (Rall.right - Rall.left) / MaxWert;
200: dy := (Rall.top - Rall.bottom) / YMax;
201: {- Aktualisierungsrechteck setzen -}
202: Rakt.top := Rall.top;
203: Rakt.bottom := Rall.bottom;
204: {- ganze Kurve zeichnen -}
205: IaktAnf := IndWert;
206: end;
207:
208: {- Kurve von links bis IndWert -}
209: if AnzWert = 0 then goto Ende;
210: indl := IndWert;
211: if IndWert > IaktAnf then begin
212: {alten Wert mitzeichnen}
213: ind0 := IaktAnf - 1;
214: if ind0 < 0 then ind0 := 0;
215: end else begin
216: ind0 := 0;
217: end;
218: DrawCurve (PaintDC, ind0, indl);
219:
220: {- Kurve bis zum rechten Rand ? -}
221: if AnzWert <= MaxWert then goto Ende;
222: if IndWert > IaktAnf then goto Ende;
223: if AktAll then begin
224: ind0 := IndWert + 1;
225: end else begin
226: ind0 := IaktAnf - 1;
227: if ind0 < 0 then ind0 := MaxWert;
228: end;
229: if ind0 >= MaxWert then goto Ende;
230: indl := MaxWert;
231: DrawCurve (PaintDC, ind0, indl);
232:
233: Ende;
234: IaktAnf := -1;
235: AktAll := False;
236: end;
237:
238: { Grafik-Rechteck uebernehmen }
239: procedure TGrafOWnd.SetGrafRect
240: (var NeuRect: TRect);
241: begin
242: AktAll := True;
243: Rall := NeuRect;
244: end;
245:
246: { neue Werte uebernehmen }
247: procedure TGrafOWnd.SetNeuWert
248: (PtNeuWert: PWDArray);
249: var
250: ind0, indl, k: Integer;
251: Rand: Word;
252: begin
253:
254: { Pufferindex hochsetzen }
255: CalcNum := CalcNum + 1;
256: IndWert := IndWert + 1;
257: if IndWert > MaxWert then
258: IndWert := IndWert - MaxWert;
259: if AnzWert <= MaxWert then
260: AnzWert := AnzWert + 1;
261:
262: {- neue Werte in Puffer eintragen -}
263: for k := 0 to MaxKurv-1 do begin
264: {- Werte ablegen -}
265: PtAllBuf[k * KvLen + IndWert]
266: := PtNeuWert[k];
267: {- Randpunkt am Pufferanfang -}
268: if IndWert = 1 then begin
269: if (AnzWert = 1) then
270: Rand := PtNeuWert[k]
271: else begin
272: Rand := PtAllBuf[k * KvLen + MaxWert];
273: end;
274: PtAllBuf[k * KvLen + 0] := Rand;
275: end;
276: end;
277:
278: {- Grafikaenderung markieren -}
279: if IaktAnf = -1 then begin
280: IaktAnf := IndWert - 1;
281: if IaktAnf = 0 then IaktAnf := MaxWert;
282: IaktPrv := IaktAnf - 1;
283: if IaktPrv = 0 then IaktPrv := MaxWert;
284: end else begin
285: { Pufferueberlauf vor InvalidateGraf }
286: if IndWert = IaktPrv then AktAll := True;
287: end;
288: end;
289:
290: { Grafikerneuerung anstossen }
291: procedure TGrafOWnd.InvalidateGraf;
292: var
293: iEnd: Integer;
294: label
295: Ende;
296: begin
297: {- Icon ? Alles l+schen ? -}
298: if IsIconic (HWindow) then goto Ende;
299: if AktAll then begin
300: InvalidateRect (HWindow, @Rall, TRUE);

```

```

301: goto Ende;
302: end;
303:
304: {alte Werte und 1 daneben loeschen}
305:
306: Rakt.left := Rall.left + Round (dx * IaktAnf);
307: IEnd := IndWert + 1;
308: if IEnd > MaxWert then IEnd := MaxWert;
309: if IaktAnf < IndWert then begin
310: {- nur neuer Bereich zu loeschen -}
311: Rakt.right :=
312: Rall.left + Round (dx * IEnd);
313: InvalidateRect (HWindow, @Rakt, TRUE);
314: end else begin
315: {- ab IaktAnf bis Ende loeschen -}
316: Rakt.right := Rall.right;
317: InvalidateRect (HWindow, @Rakt, TRUE);
318: {- ab Anfang bis IndWert+1 loeschen -}
319: Rakt.left := Rall.left;
320: Rakt.right :=
321: Rall.left + Round (dx * IEnd);
322: InvalidateRect (HWindow, @Rakt, TRUE);
323: end;
324:
325: { Kurven von ix0..ix1 zeichnen }
326: procedure TGrafOWnd.DrawCurve
327: (PaintDC: HDC; ix0,ix1: Integer);
328: var
329: ik, ianz, ix, ind0: Integer;
330: begin
331: for ik := 0 to MaxKurv-1 do begin
332: ianz := 0;
333: for ix := ix0 to ix1 do begin
334: with PtPoints[ianz] do begin
335: [Alt 9] x := Rall.left + Round (dx * ix);
336: [Alt 9] y := Rall.right - 1;
337: y := Rall.bottom + Round (dy *
338: PtAllBuf[ik * KvLen + ix]);
339: if y < Rall.top then y := Rall.top;
340: [Alt 9] if y >= Rall.bottom then
341: [Alt 9] y := Rall.bottom - 1;
342: end;
343: ianz := ianz + 1;
344: end;
345: end;
346:
347: SelectObject (PaintDC,
348: PtStifte[ik]);
349: Polyline (PaintDC, PtPoints,
350: ianz);
351: end;
352:
353: { METHODEN VON TFastAppl }
354:
355: { Message aus Loop schicken }
356: procedure TFastAppl.MessageLoop;
357: var
358: Msg: TMsg;
359: begin
360: while Msg.Message <> WM_Quit do
361: begin
362: if PeekMessage (Msg, 0, 0, 0, PM_REMOVE)
363: then begin
364: {- Windows-Message weitergeben -}
365: TranslateMessage (Msg);
366: DispatchMessage [Alt 9] (Msg);
367: end else begin
368: {- Timer-Msg verschicken -}
369: if NextMsg then
370: [Alt 9] SendMessage
371: (DasWh, WM_TIMER, VonFastAppl, 0);
372: end;
373: end;
374:
375: { Bedingung fuer Msg-Ausgabe }
376: function TFastAppl.NextMsg: Bool;
377: begin
378: NextMsg := True;
379: end;
380: end;

```

»fastgraf.pas« enthält die Methoden für eine schnelle Grafikausgabe.

Listing 2: fgdemo.pas

```

1: { Fgdemo.pas, TPW }
2:
3: program FGDemo;
4:
5: { $A+,D-,G+,I-,L-,R-,S-,V-,W-,X+ }
6: { $M 8192,16000 }
7:
8: uses
9:   WObjects, WinTypes, WinProcs, FastGraf;
10:
11: const
12:   PName = 'Demo FastGraf';
13:
14: { Parameter fuer Grafik }
15: const
16:   MyMaxKurv = 2; { Anzahl Kurven }
17:   MyMaxWert = 50; { Werte pro Kurve }
18:   Farbe = array[0..MyMaxKurv-1]
19:   of TColorRef =
20:   ( $00FF0000, { Kurve 1 blau }

```

»fgdemo.pas« demonstriert die schnelle Grafikausgabe.


```

21: $000000FF ); { Kurve 2 rot }
22: { ueberschriebene MessageLoop }
23: type
24: TgtestAppl = object(TFastAppl)
25:   procedure InitMainWindow; virtual;
26: end;
27: { Allg. Fenster mit Grafikausgabe }
28: type
29: PMyWindow = ^TMyWindow;
30: TMyWindow = object(TGrafOWnd)
31:   istep, idec: Integer;
32:   Aktiv: Bool;
33:   constructor Init
34:   (AParent: PWindowsObject;
35:    ATitle: PChar);
36:   procedure DefWndProc
37:   (var Msg: TMessage); virtual;
38:   procedure Paint (PaintDC: HDC; var
39:   PaintInfo: TPaintStruct); virtual;
40:   procedure IniCalc; virtual;
41:   procedure NextCalc;
42: end;
43: constructor TMyWindow.Init
44: (AParent: PWindowsObject;
45:  ATitle: PChar);
46: begin
47:   {- vor (!!) Init Parameter setzen -}
48:   MaxKurv:=MyMaxKurv; {Anzahl Kurven}
49:   MaxWert:=MyMaxWert; {Werte pro Kurve}
50:   PtFarbe:=@Farbe; {Farben setzen}
51:   TGrafOWnd.Init(AParent, ATitle);
52:   Aktiv:= True;
53: end;
54: { Rechengroessen initialisieren }
55: procedure TMyWindow.IniCalc;
56: begin
57:   TGrafOWnd.IniCalc;
58:   istep:= 0;
59:   idec:= +10;
60:   Randomize;
61: end;
62: { darzustellende Werte bilden }
63: procedure TMyWindow.NextCalc;
64: var
65:   NeuWert:array[0..MyMaxKurv-1] of Word;
66: begin

```

```

73:   {- Wert berechnen -}
74:   istep:= istep +idec;
75:   if istep<0 then begin
76:     istep:= 0;
77:     idec:= Random(100) +1;
78:   end;
79:   if istep>YMax then
80:     idec:=-Random(100) -1;
81:   {- Werte uebergeben -}
82:   NeuWert[0]:= Random(YMax+1);
83:   NeuWert[1]:= istep;
84:   SetNeuWert(@NeuWert);
85: end;
86: { Messages verarbeiten }
87: procedure TMyWindow.DefWndProc
88: (var Msg: TMessage);
89: var
90:   i: Word;
91: begin
92:   case Msg.Message of
93:     {- Ausgabe anhalten/fortsetzen -}
94:     WM_RBUTTONDOWN:
95:       if Aktiv= False
96:       then
97:         Aktiv:= True;
98:     {- Werte bilden und darstellen -}
99:     WM_TIMER:
100:       if Aktiv then begin
101:         for i:=1 to 10 do NextCalc;
102:         InvalidateGraf;
103:       end;
104:     {- Standardverarbeitung -}
105:     TGrafOWnd.DefWndProc (Msg);
106:   end;
107:   { Paint-Routine }
108:   procedure TMyWindow.Paint(PaintDC:HDC;
109:   var PaintInfo: TPaintStruct);
110:   var
111:     Rahmen: array[0..4] of TPoint;
112:     NeuRect: TRect;
113:     ib, ih: Integer;
114:   begin
115:     { alles neu zeichnen ? }

```

```

125:   if PaintAll then begin
126:     MessageBeep(0);
127:     PaintAll:= False;
128:   { Grafik fensterabhaengig setzen }
129:   GetClientRect (HWindow, NeuRect);
130:   with NeuRect do begin
131:     ib:= (right -left) div 6;
132:     ih:= (bottom -top) div 5;
133:     left := left +ib;
134:     right := right -ib;
135:     top := top +ih;
136:     bottom:= bottom -ih;
137:   end;
138:   SetGrafRect (NeuRect);
139:   {- Rahmen um Kurven -}
140:   Rahmen[0].x:= NeuRect.left -1;
141:   Rahmen[0].y:= NeuRect.top -1;
142:   Rahmen[1].x:= NeuRect.left -1;
143:   Rahmen[1].y:= NeuRect.bottom;
144:   Rahmen[2].x:= NeuRect.right;
145:   Rahmen[2].y:= NeuRect.bottom;
146:   Rahmen[3].x:= NeuRect.right;
147:   Rahmen[3].y:= NeuRect.top -1;
148:   Rahmen[4].x:= Rahmen[0].x;
149:   Rahmen[4].y:= Rahmen[0].y;
150:   Polyline (PaintDC, Rahmen, 5);
151: end;
152: { Kurven zeichnen }
153: TGrafOWnd.Paint(PaintDC, PaintInfo);
154: end;
155: procedure TgtestAppl.InitMainWindow;
156: begin
157:   MainWindow:= New(PMyWindow,
158:   Init(nil, PName));
159: end;
160: var
161:   MyApp: TgtestAppl;
162: begin
163:   MyApp.Init(PName);
164:   MyApp.Run;
165:   MyApp.Done;
166: end.

```

»fgdemo.pas« demonstriert die schnelle Grafikausgabe.

**Zeigen Sie sich doch einmal von Ihrer besten Seite.
Zum Beispiel hier!**

**Anzeigenschluß für die Ausgabe Nr. 2/94 ist der
20.04.94**

**Wenden Sie sich einfach
gleich an Ihren nächsten Mediaberater:**

DMV Daten- und Medienverlag
Gruberstraße 46a • 85586 Poing
☎ 08121/769-300

DMV Daten- und Medienverlag
Fuldaer Straße 6 • 37269 Eschwege
☎ 05651/809-375

DMV Daten- und Medienverlag
Postfach 130309 • 20103 Hamburg
☎ 040/40 44 12 05

Der PC spielt auf

Die Soundprogrammierung gehört mit Sicherheit zu den faszinierendsten Anwendungen auf dem PC, leider aber nicht gerade zu den einfachsten. Auf den folgenden Seiten führen wir Sie Schritt für Schritt in die Programmierung der musikalischen Welt Ihres Computers ein.



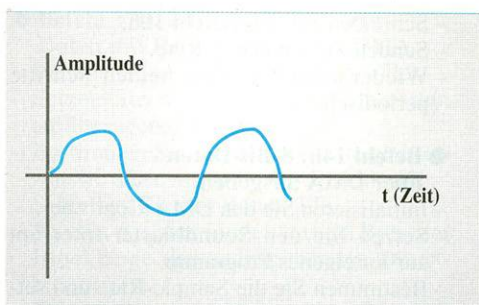


Bild 1. Geräusch, Ton und Klang: drei unterschiedliche Arten von Schwingungen

Lesen Sie die notwendigen Grundlagen, sofern Sie Ihnen noch nicht oder nicht völlig bekannt sind. Auch der folgende Abschnitt über die Techniken digitaler Musik wird leider recht theoretisch bleiben. Danach jedoch wenden wir uns der Programmierung der Soundblasterkarte zu, die zur Zeit wohl den Standard in der PC-Musikwelt darstellt. Dieser sehr praxisnahe Teil sollte die noch nicht so erfahrenen Programmierer unter Ihnen jedoch nicht abschrecken, da Sie mit der in einem eigenen Beitrag behandelten Unit »modplay.pas« eine einfach zu handhabende Hilfe zur Einbindung von Musik und Soundeffekten in Ihre eigenen Programme erhalten.

Grundlagen der Soundprogrammierung

Um die Soundprogrammierung richtig zu verstehen, ist zunächst ein Verständnis des Prinzips der Tonerzeugung notwendig. Wie kommt es, daß wir einen Ton hören? Kurzgefaßt sendet eine Quelle Schwingungen aus, die sich durch ein Medium (meistens die Luft, aber auch beispielsweise Wasser) als Welle fortsetzen, bis sie schließlich auf unser Ohr treffen, von wo aus sie an das Gehirn weitergeleitet und interpretiert werden.

Man unterscheidet drei Arten eintreffender Schwingungen (Bild 1):

- ein Geräusch, welches sich dadurch auszeichnet, daß das Medium unregelmäßig erschüttert wird
- ein Ton, der aus rein sinusförmigen Schwingungen (Bild 2) besteht
- Klänge, die beim Spielen von Instrumenten, zum Beispiel einem Klavier oder einer Gitarre, entstehen

Eine Schwingung besteht aus Amplitude (Lautstärke) und Frequenz (Tonhöhe). Die Größe der Amplitude entspricht dabei der empfundenen Lautstärke der Schwingung. Mit steigender Frequenz – je öfter die Kurve die x-Achse schneidet – erscheint uns der Ton zunehmend höher.

Wenn Sie sich diesen Sachverhalt verdeutlicht haben, werden Sie sicherlich verstehen, warum es sinnvoll ist, eine möglichst hohe Sampling-Rate sowohl zum Aufnehmen als auch zum Abspielen zu wählen. Die Sample-Daten bezeichnen jeweils einen Punkt auf der Schwingungs-

kurve. Je mehr Punkte pro Zeiteinheit zur Verfügung stehen, desto genauer läßt sich die Kurve zeichnerisch beziehungsweise ausgeben. Bei abnehmender Informationsmenge für einen zeitlichen Bereich entstehen mehr und mehr wenige unpassende Zwischenwerte, welche die Qualität trüben (Bild 3).

Der ADPCM-Packalgorithmus

Auf dem zuvor beschriebenen Prinzip basiert der ADPCM-Packalgorithmus, der in den Voc-Dateien Anwendung findet, und der vom Soundblaster direkt durch die Hardware unterstützt wird. ADPCM steht für »Adaptive Delta Pulse Code Modulation«.

Er speichert nicht jede einzelne Amplitude, sondern nur deren Differenzen. Dies ergäbe an sich noch keine Platzersparnis, wenn der Algorithmus die Differenzen in

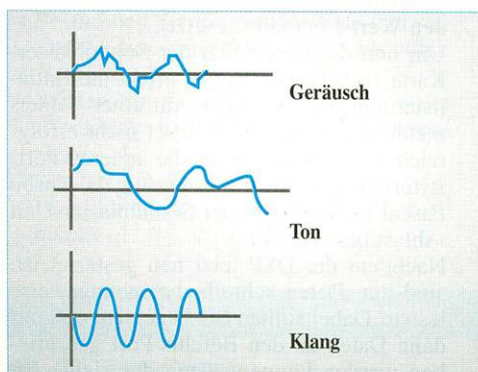


Bild 2. Ein Ton besteht lediglich aus sinusförmigen Schwingungen.

jeweils einem Byte speichern würde. Das dabei zugrunde gelegte Verfahren geht nun aber davon aus, daß zwei Werte »relativ« dicht beieinander liegen. Darum verwendet es zur Speicherung der Informationen bei der 4-Bit-Komprimierung vier Bit, bei der 2,6-Bit-Komprimierung drei Bit und bei der 2-Bit-Komprimierung zwei Bit. Das erste Bit dient zur Speicherung des Vorzeichens, die restlichen zur Speicherung der Differenz, woraus sich bei der 4-Bit-Komprimierung ein zulässiger Wertebereich von -7 bis +7 ergibt. Alle Differenzen, die diese Maximalwerte über- oder unterschreiten, werden automatisch als Maximalwert interpretiert. Dies ist auch die Ursache für den Qualitätsverlust bei gepackten Daten.

Ein weiterer Nachteil dieser Methode liegt darin, daß die Informationen nur via DMA ausgabbar sind, da wir uns

sonst selbst um das Entpacken kümmern müßten. Deshalb haben Samples für die digitale Musik fast keine Bedeutung, weil es hier auf maximale Klangqualität bei möglichst geringer Verarbeitungszeit ankommt. Komprimierte Samples sind demgegenüber ideal zur Sprachausgabe geeignet oder wenn nur sehr wenig Platz für die Daten zur Verfügung steht.

Multimedia im Griff

In den folgenden Beiträgen zum Thema Multimedia-Programmierung gehen wir unter anderem anhand der beiden Units »mixer.pas« und »sblast.pas« auf die Soundblaster-Programmierung näher ein und zeigen, wie man den Timer-Interrupt für eigene Programme manipuliert.

Im Bereich MIDI- und MOD-Dateien schlüsseln wir das MOD-Dateiformat auf und stellen einen MOD-Editor (»editmod.pas«) sowie eine Unit zum Abspielen dieser Dateien im Hintergrund vor. Der Einsatz externer MIDI-Geräte kommt ebenfalls nicht zu kurz. Ein Beispielprogramm verdeutlicht die Entgegennahme von MIDI-Daten, beispielsweise eines Synthesizer- oder Drum-Keyboards.

Einen tiefen Blick hinter die Kulissen der Programmierung von animierten Demos mit Laufschrift, skalierbaren Sprites und Tonuntermalung gestattet ein Beitrag (gleich mit kompletter Unit und Anwendungsbeispiel), der unter anderem auch die Herzen von Spieleprogrammierern höher schlagen läßt. Trotz vielfältiger Bewegung von Sprites vor einem als GIF-Grafik geladenen Hintergrund bleibt das Bild dabei störungsfrei und flüssig in der Bewegung.

Betreten Sie also mit uns die faszinierende Welt der Sound- und Multimedia-Programmierung.

(Boris Bertelsohn/hs/bj)

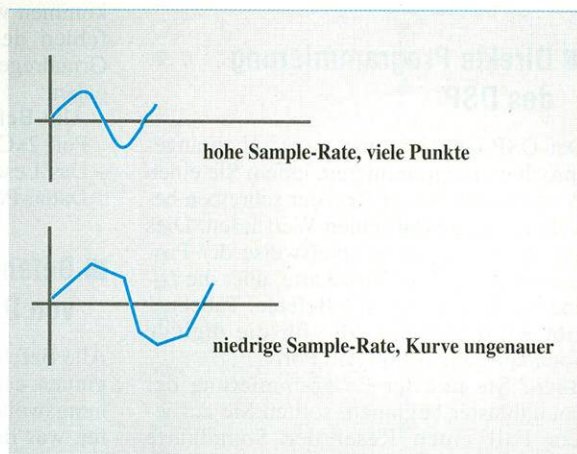


Bild 3. Die Sample-Daten bezeichnen jeweils einen Punkt auf der Schwingungskurve. Je genauer die Kurve, desto besser die Klangqualität.

DSP, das Tor zur Tonwelt

DSP steht für »Digital Soundprocessor« (digitaler Soundprozessor). Er ist gewissermaßen die CPU der Soundblaster und regelt die Ein- und Ausgabe von Sound- und MIDI-Daten.

Der in der Soundblaster-Serie zu findende DSP trägt die Bezeichnung CT-DSP-1321. Sie programmieren ihn, indem Sie einen Wert in eines seiner Register schreiben. Standardmäßig findet man ihn an der Adresse 220h, die Entwickler haben den Soundblaster-Karten aber aus Flexibilitätsgründen die Fähigkeit mitgegeben, die Basisadresse zwischen 210h und 280h frei (genauer gesagt in Schritten von 10h) festzulegen. Aufgrund dieser Flexibilität setzen Sie an den Stellen, wo zum Beispiel von Port 22Ch die Rede ist, gegebenenfalls Ihre eigene Basisadresse ein (zum Beispiel Port 24Ch, wenn Sie die Base auf 240h eingestellt haben).

Neben der Basisadresse ist auch der von der Soundblaster benötigte Hardware-Interrupt in gewissen Grenzen flexibel wählbar. Hier stehen IRQ2, IRQ3, IRQ5, IRQ7 und IRQ10 zur Verfügung, der Vorgabewert auf Soundblaster und Soundblaster Pro ist IRQ7, auf der Soundblaster 16 (ASP) ist es der IRQ5. Der IRQ wird auf den Soundblaster- und Soundblaster-Pro-Karten über Jumper, auf der Soundblaster 16 (ASP) über die Software eingestellt. Ab der Soundblaster Pro ist es sogar erlaubt, den von der Karte benutzten DMA-Kanal zu wählen (0, 1 oder 3). In den Demos und Units zur Soundprogrammierung dieser Ausgabe setzen wir den DMA-Kanal 1 als Standard voraus.

■ Direkte Programmierung des DSP

Der DSP läßt sich, wie zuvor schon angesprochen, programmieren, indem Sie einen Wert in eines seiner Register schreiben beziehungsweise dort einen Wert lesen. Dies geschieht, analog beispielsweise der Programmierung der VGA-Karte, über die *IN*- und *OUT*- sowie *PORT*-Befehle. Tabelle 1 gibt Aufschluß über die für die digitale Soundausgabe benötigten Ports.

Bevor Sie mit der Programmierung der Soundblaster beginnen, sollten Sie auf jeden Fall einen Reset der Soundkarte durchführen. Dafür sendet man zuerst den Wert 1 an den Port 2x6h (x steht dabei stellvertretend für den zweiten hexadezimalen Wert der individuell genutzten Ba-

sisadresse), läßt das Programm 3 ms warten und schickt schließlich den Wert 0 an den Port 2x6h. Ob der Reset erfolgreich verlief, ist am Daten-Port 2xAh zu erkennen. Liest das Programm dort den Wert AAh, so war die Initialisierung erfolgreich. Der Reset selbst dauert zirka 100 ms. Des weiteren liegen am Daten-Port nur dann Informationen vor, wenn das Bit 7 (das höchstwertige Bit) des Ports 2xEh den Wert 1 enthält (gesetzt).

Um nun die Flexibilität der Soundblaster-Karte zu berücksichtigen, bricht die Initialisierungsprozedur nicht mit einer Fehlermeldung ab, wenn der Reset nicht erfolgreich war, sondern testet die anderen Port-Alternativen. Eine Umsetzung in Turbo Pascal finden Sie in der Soundblaster-Unit »sblast.pas« (Listing).

Nachdem der DSP jetzt neu gestartet ist, sind nun Daten schreib- beziehungsweise lesbar. Dabei sollten Sie beachten, daß nur dann Daten an den Befehls-Port geschrieben werden können, wenn das siebte Bit an Port 2xCh nicht gesetzt ist. Genauso lassen sich nur dann Daten vom Daten-Port (2xAh) lesen, wenn er einen Inhalt ungleich AAh aufweist. Eine Prozedur zum Schreiben von Daten und eine Funktion zum Lesen finden Sie ebenfalls im Listing verwirklicht.

Da Sie nun wissen, wie Sie die Soundblaster initialisieren und wie Sie Daten schreiben beziehungsweise lesen können, kommen wir jetzt zu den eigentlichen Befehlen des DSP (Tabelle 2). Folgende Grundregeln gelten für alle Ein- und Ausgaben:

- Alle Befehle werden über den Befehls-Port 2xCh ausgegeben.
- Das Lesen der Informationen erfolgt am Daten-Port 2xAh.

■ Befehle zum Abspielen von Daten

Alle Befehle zum Abspielen von Daten beginnen entweder mit 1 oder mit 7 beziehungsweise mit 9 für High-Speed-Transfer, was Ihnen die Einordnung der Befehle etwas erleichtern dürfte.

● Befehl 10h: 8-Bit-Direktausgabe von Daten

- Schreiben Sie den Befehl 10h.
- Senden Sie ein Daten-Byte.
- Wiederholen Sie diese beiden Schritte periodisch.

● Befehl 14h: 8-Bit-Daten über DMA ausgeben

- Initialisieren Sie den DMA-Controller.
- Setzen Sie den Soundblaster-Interrupt auf Ihr eigenes Programm.
- Bestimmen Sie die Sample-Rate und setzen Sie diese mit Hilfe des Befehls 40h.
- Schreiben Sie den Befehl 14h.
- Senden Sie das niederwertige Byte der Blocklänge des auszugebenden Datenblocks -1.
- Senden Sie das höherwertige Byte der Blocklänge des auszugebenden Datenblocks -1.

Die Sample-Ausgabe startet sofort nach Ausgabe des höherwertigen Byte (letzter Schritt). Zu beachten ist, daß der DSP am Ende der Blockübertragung den anfangs angesprochenen Interrupt erzeugt. Wollen Sie Blöcke größer als 64 KByte ausgeben, sollten Sie Ihre Interrupt-Routine so anlegen, daß sie das nächste Teilstück der Daten ausgibt. Die maximale Größe von 64 KByte begründet sich darin, daß der DMA-Controller jeweils nur Daten innerhalb eines Segments adressieren kann.

● Befehl 91h: 8-Bit-Daten im High-Speed-Modus über DMA ausgeben

- Initialisieren Sie den DMA-Controller.
- Setzen Sie den Soundblaster-Interrupt auf Ihr eigenes Programm.
- Bestimmen Sie die Sample-Rate und setzen Sie diese mit Hilfe des Befehls 40h.
- Schreiben Sie den Befehl 91h.
- Senden Sie das niederwertige Byte der Blocklänge des auszugebenden Datenblocks -1.
- Senden Sie das höherwertige Byte der Blocklänge des auszugebenden Datenblocks -1.
- Schreiben Sie den Befehl 91h.

Wichtig: Die Befehle 48h und 91h sind erst ab der Soundblaster Pro unterstützt und dienen zur High-Speed-Ausgabe von Sample-Daten. Beachten Sie auch die Anmerkungen zu Befehl 14h.

● Befehl 16h: 2-Bit-komprimierte Daten über DMA ausgeben

- Initialisieren Sie den DMA-Controller.
- Setzen Sie den Soundblaster-Interrupt auf Ihr eigenes Programm.
- Bestimmen Sie die Sample-Rate und setzen Sie diese mit Hilfe des Befehls 40h.
- Schreiben Sie den Befehl 16h.
- Senden Sie das niederwertige Byte der Blocklänge des auszugebenden Datenblocks -1.
- Senden Sie das höherwertige Byte der Blocklänge des auszugebenden Datenblocks -1.

Beachten Sie auch die Anmerkungen zu Befehl 14h.

● Befehl 17h: 2-Bit-komprimierte Daten mit Referenz-Byte via DMA

- Initialisieren Sie den DMA-Controller.
- Setzen Sie den Soundblaster-Interrupt auf Ihr eigenes Programm.
- Bestimmen Sie die Sample-Rate und setzen Sie diese mit Hilfe des Befehls 40h.
- Schreiben Sie den Befehl 17h.
- Senden Sie das niederwertige Byte der Blocklänge des auszugebenden Datenblocks –1.
- Senden Sie das höherwertige Byte der Blocklänge des auszugebenden Datenblocks –1.

Beachten Sie auch die Anmerkungen zu Befehl 14h.

Bei dieser Methode wird das erste Byte als Referenz-Byte interpretiert. Wollen Sie mehrere Sample-Blöcke hintereinander abspielen, so verwenden Sie für die folgenden Blöcke den Befehl 16h.

● Befehl 74h: 4-Bit-komprimierte Daten über DMA ausgeben

Siehe Befehl 16h.

● Befehl 75h: 4-Bit-komprimierte Daten mit Referenz-Byte via DMA

Siehe Befehl 17h.

● Befehl 76h: 2,6-Bit-komprimierte Daten über DMA ausgeben

Siehe Befehl 16h.

● Befehl 77h: 2,6-Bit-komprimierte Daten mit Referenz-Byte via DMA

Siehe Befehl 17h.

■ Befehle zum Aufnehmen von Daten

Alle Befehle zum Aufnehmen von Daten beginnen entweder mit 2 oder mit 9 (für High-Speed-Transfer).

● Befehl 20h: direkte Aufnahme von 8-Bit-Daten

- Schreiben Sie den Befehl 20h.
 - Lesen Sie ein Daten-Byte.
 - Speichern Sie das gelesene Byte.
- Diese Schritte sind periodisch zu wiederholen.

● Befehl 24h: Aufnahme von 8-Bit-Daten über DMA

- Initialisieren Sie den DMA-Controller.
- Setzen Sie den Soundblaster-Interrupt auf Ihr eigenes Programm.
- Bestimmen Sie die Sample-Rate und setzen Sie diese mit Hilfe des Befehls 40h.
- Schreiben Sie den Befehl 24h.
- Senden Sie das niederwertige Byte der Blocklänge des einzugebenden Datenblocks –1.
- Senden Sie das höherwertige Byte der Blocklänge des einzugebenden Datenblocks –1.

Die Sample-Aufnahme startet sofort nach Ausgabe des höherwertigen Byte (letzter Schritt). Zu beachten ist, daß der DSP am Ende der Blockübertragung den bereits angesprochenen Interrupt erzeugt. Wollen Sie Blöcke größer als 64 KByte aufnehmen, sollten Sie Ihre Interrupt-Routine so anlegen, daß sie das nächste Teilstück der Daten ausgibt. Die maximale Größe von 64 KByte begründet sich darin, daß der DMA-Controller jeweils nur Daten innerhalb eines Segments adressieren kann.

● Befehl 99h: Aufnahme von 8-Bit-Daten im High-Speed-Modus via DMA

- Initialisieren Sie den DMA-Controller.
- Setzen Sie den Soundblaster-Interrupt auf Ihr eigenes Programm.
- Bestimmen Sie die Sample-Rate und setzen Sie diese mit Hilfe des Befehls 40h.
- Schreiben Sie den Befehl 48h.
- Senden Sie das niederwertige Byte der Blocklänge des einzugebenden Datenblocks –1.
- Senden Sie das höherwertige Byte der Blocklänge des einzugebenden Datenblocks –1.
- Schreiben Sie den Befehl 99h.

Wichtig: Die Befehle 48h und 99h sind erst ab der Soundblaster Pro unterstützt und dienen zur High-Speed-Aufnahme von Sample-Daten. Beachten Sie auch die Anmerkungen zu Befehl 24h.

Port	Name	Status	Soundblaster	Soundblaster Pro	Soundblaster 16
Base+00h	linkes FM-Register, Auswahl und Status	lesen/schreiben	○	●	●
Base+01h	linkes FM-Register, Daten	schreiben	○	●	●
Base+02h	rechtes FM-Register, Auswahl und Status	lesen/schreiben	○	●	●
Base+03h	rechtes FM-Register, Daten	schreiben	○	●	●
Base+04h	Mixer-Chip, Indexregister	schreiben	○	●	●
Base+05h	Mixer-Chip, Datenregister	lesen/schreiben	○	●	●
Base+06h	DSP-Reset	schreiben	●	●	●
Base+08h	beide FM-Register, Auswahl und Status	lesen/schreiben	●	●	●
Base+09h	beide FM-Register, Daten	schreiben	●	●	●
Base+0Ah	DSP-Daten (Read Data)	lesen	●	●	●
Base+0Ch	DSP-Daten oder Befehl und Pufferstatus	lesen/schreiben	●	●	●
Base+0Dh	DSP-Timer-Interrupt Clear Port	lesen	○	○	●
Base+0Eh	DSP-Status vorhandener Daten	lesen	●	●	●
Base+0Fh	16-Bit Voice-Interrupt Clear Port	lesen	○	○	●
Base+10h	reserviert	–	–	–	–
Base+11h	reserviert	–	–	–	–
Base+12h	reserviert	–	–	–	–
Base+13h	reserviert	–	–	–	–
200h-207h	Game-Port	lesen/schreiben	●	●	●
MPUBase+0	MPU-401 Daten-Port	lesen/schreiben	●	●	●
MPUBase+1	MPU-401 Status-Port	lesen	●	●	●
MPUBase+1	MPU-401 Adress-Port	schreiben	●	●	●
338h	FM-Register Status-Port	lesen	●	●	●
339h	FM-Auswahlregister	schreiben	●	●	●
389h	FM-Register Daten-Port	schreiben	●	●	●
38Bh	Advanced FM-Register Auswahl	schreiben	●	●	●
38Bh	Advanced FM-Register Daten-Port	schreiben	●	●	●

● Funktion vorhanden ○ Funktion nicht vorhanden

Tabelle 1. Registerbelegung der Soundblaster-Karten

■ Sonstige Befehle

Kommen wir nun zu den Befehlen, die diverse Aufgaben erledigen.

● Befehl 40h: Programmieren der Sample-Rate

- Schreiben Sie den Befehl 40h.
- Senden Sie das nach der folgenden Formel ermittelte Frequenz-Byte:

$$\text{Frequenz-Byte} = 256 - (1000000 / \text{Sample-Rate})$$

● Befehl 48h: Einstellen der Blockgröße

Die Benutzung des Befehls ist im Zusammenhang mit den Befehlen 91h und 99h erklärt worden. Dieser Befehl wird erst seit der Soundblaster Pro unterstützt.

● Befehl 80h: Definition eines Stille-Blocks

- Setzen Sie den Soundblaster-Interrupt auf Ihr eigenes Programm.
- Bestimmen Sie die Sample-Rate und setzen Sie diese mit Hilfe des Befehls 40h.
- Schreiben Sie den Befehl 80h.

- Senden Sie das niederwertige Byte der Länge des Stille-Blocks –1.
 - Senden Sie das höherwertige Byte der Länge des Stille-Blocks –1.
- Am Ende des Stille-Blocks erzeugt die Soundblaster den angesprochenen Interrupt.

● Befehl D0h: Unterbrechen der DMA-Ausgabe

- Schreiben Sie den Befehl D0h.
- Der Befehl unterbricht die Sample-Ausgabe, die sich mit dem Befehl D4h fortsetzen läßt.

● Befehl D1h: Einschalten des Lautsprechers

- Schreiben Sie den Befehl D1h.
- Die Ausführung des Befehls nimmt zirka 112 ms in Anspruch. Er schaltet die Verbindung zwischen DSP und Verstärker ein.

● Befehl D3h: Ausschalten des Lautsprechers

- Schreiben Sie den Befehl D3h.
- Die Ausführung des Befehls nimmt zirka 220 ms in Anspruch. Er unterbricht die Verbindung zwischen DSP und Verstärker.

● Befehl D4h: Fortsetzen der DMA-Ausgabe

- Schreiben Sie den Befehl D4h.
- Der Befehl setzt die unterbrochene Sample-Ausgabe fort. Sie muß allerdings zuvor mit dem Befehl D0h angehalten worden sein.

● Befehl D8h: Abfrage des Lautsprecher-Status

- Schreiben Sie den Befehl D8h.
 - Lesen Sie das Status-Byte vom Daten-Port.
- Besitzt das Status-Byte den Wert 0, dann ist der Lautsprecher ausgeschaltet, bei 255 ist er eingeschaltet. Dieser Befehl wird erst ab der Soundblaster Pro unterstützt.

● Befehl E1h: Abfrage der Versionsnummer der Soundblaster

- Senden Sie den Befehl E1h.
 - Lesen Sie das höherwertige Byte der Nummer der Version vom Daten-Port.
 - Lesen Sie das niederwertige Byte der Nummer der Version vom Daten-Port.
- (Boris Bertelsens/bj)

Befehl	Funktion
10h	8-Bit-Ausgabe direkt
14h	8-Bit-Ausgabe über DMA
16h	Ausgabe von 2-Bit-komprimierten Samples über DMA
17h	Ausgabe von 2-Bit-komprimierten Samples mit Referenz-Byte über DMA
20h	direkte Aufnahme
24h	Aufnahme über DMA
30h	direkte MIDI-Eingabe
31h	MIDI-Eingabe über Interrupt
32h	direkte MIDI-Eingabe mit Timestamp
33h	MIDI-Eingabe über Interrupt mit Timestamp
34h	MIDI-UART-Modus, direkt
35h	MIDI-UART-Modus über Interrupt
37h	MIDI-UART über Interrupt mit Timestamp
38h	MIDI-Code senden
40h	Setzen der Sample-Rate
48h	Blockgröße einstellen
74h	Ausgabe von 4-Bit-komprimierten Samples über DMA
75h	Ausgabe von 4-Bit-komprimierten Samples mit Referenz-Byte über DMA
76h	Ausgabe von 2,6-Bit-komprimierten Samples über DMA
77h	Ausgabe von 2,6-Bit-komprimierten Samples mit Referenz-Byte über DMA
80h	Stille-Block definieren
91h	8-Bit-Ausgabe über DMA in High-Speed (ab Soundblaster Pro)
99h	8-Bit-Eingabe über DMA in High-Speed (ab Soundblaster Pro)
D0h	DMA beenden
D1h	Lautsprecher einschalten
D3h	Lautsprecher ausschalten
D4h	DMA fortsetzen
D8h	Abfrage der Lautsprechereinstellung (ab Soundblaster Pro)
E1h	Versionsabfrage

Tabelle 2. DSP-Befehlsübersicht

Listing: sbblast.pas

```

1: {
2:   Programm: Unit SBlast
3:   Funktion: Bereitstellung und Demonstration grundlegender
4:             Soundblaster-Funktionen
5:   Sprache: Turbo Pascal ab 6.0
6:   Autor:   Boris Bertelsens
7:   (c) 1993 DMV GmbH & Co. KG
8: }
9:
10: UNIT SBlast;
11:
12: interface
13:   Uses Crt,dos;
14:
15:   Const dsp_adr      : word = $220;
16:         dsp_irq      : byte = $5;
17:         SbRegDetected : BOOLEAN = FALSE;
18:
19:   var SbVersMaj : byte;
20:       SbVersMin : byte;
21:       SbVersStr : string[5];
22:
23:   procedure init_sb;
24:   procedure wr_dsp(v : byte);
25:   function  SbReadByte : byte;
26:   procedure SB_Befehl10h(v : byte);
27:   procedure Set_Timeconst(tc : byte);
28:   procedure Sende_Stilleblock(slaenge
29:                               : word);
30:   procedure Lautsprecher_Ein;
31:   procedure Lautsprecher_Aus;
32:   procedure Stopp_DMA;
33:   procedure Continue_DMA;
34:   function  Lautsprecher_Aktiv:boolean;
35:   procedure GetDSPVersion;
36:
37: implementation
38:
39:   function wrt_dsp_adr : string;
40:   {
41:     Liefert die Base-Adresse des SB als
42:     String zurück
43:   }

```

Die Unit »sbblast.pas« stellt Ihnen grundlegende Anweisungen zur Programmierung der Soundblaster-Karte bereit.


```

44: begin;
45:   case dsp_adr of
46:     $210 : wrt_dsp_adr := '210';
47:     $220 : wrt_dsp_adr := '220';
48:     $230 : wrt_dsp_adr := '230';
49:     $240 : wrt_dsp_adr := '240';
50:     $250 : wrt_dsp_adr := '250';
51:     $260 : wrt_dsp_adr := '260';
52:     $270 : wrt_dsp_adr := '270';
53:     $280 : wrt_dsp_adr := '280';
54:   END;
55: end;
56:
57: function Reset_sb : boolean;
58: {
59:   Die Funktion resetet den DSP. War
60:   der Reset erfolgreich, wird TRUE zu-
61:   rückgeliefert, ansonsten FALSE
62: }
63: const ready = $AA;
64: var ct, stat : BYTE;
65: BEGIN
66:   port[dsp_adr+$6] := 1;
67:   { dsp_adr+$6 = Resetfunktion }
68:   for ct := 1 to 100 do;
69:     port[dsp_adr+$6] := 0;
70:     stat := 0;
71:     ct := 0;
72:     while (stat <> ready)
73:       and (ct < 100) do begin
74:         { Der Vergleich ct < 100, da die
75:           Initialisierung ca.100ms dauert }
76:         stat := PORT[dsp_adr+$E];
77:         stat := PORT[dsp_adr+$A];
78:         inc(ct);
79:       end;
80:       Reset_sb := (stat = ready);
81: END;
82:
83: function Detect_Reg_sb : boolean;
84: {
85:   Die Funktion liefert TRUE zurück,
86:   wenn ein Soundblaster initialisiert
87:   werden konnte, ansonsten FALSE. Die
88:   Variable dsp_adr wird auf die Base-
89:   Adresse des SB gesetzt.
90: }
91: VAR Port, Lst : word;
92: BEGIN
93:   Detect_Reg_sb := SbRegDetected;
94:   IF SbRegDetected THEN EXIT;
95:   { Exit, wenn initialisiert }
96:   Port := $210;
97:   Lst := $280;
98:   { Mögliche SB-Adressen zwischen $210
99:     und $280 ! }
100:
101:   while (not SbRegDetected)
102:     and (Port <= Lst) do begin
103:     dsp_adr := Port;
104:     SbRegDetected := Reset_sb;
105:     if not SbRegDetected then
106:       inc(Port, $10);
107:     end;
108:     Detect_Reg_sb := SbRegDetected;
109:   END;
110:
111: procedure init_sb;
112: {
113:   Die Procedure prüft, ob ein Sound-
114:   blaster vorhanden ist, und
115:   initialisiert diesen mit der
116:   richtigen BASE - Adresse
117: }
118: BEGIN;
119:   clrscr;
120:   if Detect_Reg_Sb then begin;
121:     writeln('Soundblaster an Base ',
122:       wrt_dsp_adr, 'h gefunden !');
123:     writeln('Karte erfolgreich ',
124:       'initialisiert');
125:   end else begin;

```

```

126:     writeln('Keinen Soundblaster oder',
127:       'kompatible Karte gefunden');
128:     writeln('Prüfen Sie ihre Karte !');
129:   end;
130: END;
131:
132: procedure wr_dsp(v : byte);
133: {
134:   Wartet, bis der DSP zum Schreiben
135:   bereit ist, und schreibt dann das
136:   in "v" übergebene Byte in den DSP
137: }
138: begin;
139:   while port[dsp_adr+$c] >= 128 do ;
140:     port[dsp_adr+$c] := v;
141:   end;
142:
143: function SbReadByte : byte;
144: {
145:   Die Funktion wartet, bis der DSP
146:   gelesen werden kann und liefert
147:   den gelesenen Wert zurück
148: }
149: begin;
150:   while port[dsp_adr+$a] = $AA do ;
151:     { warten, bis DSP ready }
152:     SbReadByte := port[dsp_adr+$a];
153:     { Wert schreiben }
154:   end;
155:
156: procedure SB_Befehl10h(v : byte);
157: {
158:   Direkte Ausgabe von 8-Bit Daten am SB
159: }
160: begin;
161:   wr_dsp($10);
162:   wr_dsp(v);
163: end;
164:
165: procedure Set_Timeconst(tc : byte);
166: {
167:   Timeconstant setzten, die nach der
168:   Formel
169:   tc = 256 - (1000000/Real_Frequenz)
170:   berechnet wird.
171: }
172: begin;
173:   wr_dsp($40);
174:   wr_dsp(tc);
175: end;
176:
177: procedure Meine_Routine; interrupt;
178: begin;
179:   end;
180:
181: procedure Sende_Stilleblock(slaenge
182:   : word);
183: {
184:   Sendet einen Stilleblock, für
185:   Ruheblocks im VOC-Format verwenden
186: }
187: const Frequenz : word = 22000;
188: begin;
189:   setintvec(dsp_irq, @Meine_Routine);
190:   Set_Timeconst(256 - (1000000
191:     div Frequenz));
192:   wr_dsp($80);
193:   wr_dsp(Lo(slaenge-1));
194:   wr_dsp(Hi(slaenge-1));
195: end;
196:
197: procedure Lautsprecher_Ein;
198: {
199:   Schaltet den Lautsprecher ein
200: }
201: begin;
202:   wr_dsp($D1);
203:

```

Die Unit »sbblast.pas« stellt Ihnen grundlegende Anweisungen zur Programmierung der Soundblaster-Karte bereit.


```

204: end;
205:
206: procedure Lautsprecher_Aus;
207: {
208:   Schaltet den Lautsprecher aus
209: }
210: begin;
211:   wr_dsp($D3);
212: end;
213:
214: procedure Stopp_DMA;
215: {
216:   Unterbricht die DMA-Ausgabe
217: }
218: begin;
219:   wr_dsp($D0);
220: end;
221:
222: procedure Continue_DMA;
223: {
224:   Setzt die mit Stopp_DMA angehaltene
225:   Ausgabe fort
226: }
227: begin;
228:   wr_dsp($D4);
229: end;
230:
231: function Lautsprecher_Aktiv:boolean;
232: {
233:   Liefert TRUE, wenn der Lautsprecher
234:   eingeschaltet ist, sonst FALSE
235:   Erst ab SB Pro !
236: }
237: begin;
238:   wr_dsp($D8);

```

```

239:   if SbReadByte = 0 then
240:     Lautsprecher_Aktiv := false
241:   else
242:     Lautsprecher_Aktiv := true;
243: end;
244:
245: procedure GetDSPVersion;
246: {
247:   Ermittelt die Version des DSP und
248:   gibt diese aus
249: }
250: var i : word;
251:     t : WORD;
252:     s : STRING[2];
253: begin
254:   wr_dsp($E1);
255:   SbVersMaj := SbReadByte;
256:   SbVersMin := SbReadByte;
257:   str(SbVersMaj, SbVersStr);
258:   SbVersStr := SbVersStr + '.';
259:   str(SbVersMin, s);
260:   if SbVersMin > 9 then
261:     SbVersStr := SbVersStr + s
262:   else
263:     SbVersStr := SbVersStr + '0' + s;
264:   writeln('Sie haben die Soundblast',
265:     'er-Version ', SbVersStr);
266: end;
267:
268: begin;
269: end.

```

Die Unit »sblast.pas« stellt Ihnen grundlegende Anweisungen zur Programmierung der Soundblaster-Karte bereit.

**Zeigen Sie sich doch einmal von Ihrer besten Seite.
Zum Beispiel hier!**

**Anzeigenschluß für die Ausgabe Nr. 2/94 ist der
20.04.94**

**Wenden Sie sich einfach
gleich an Ihren nächsten Mediaberater:**

**DMV Daten- und Medienverlag
Gruberstraße 46a • 85586 Poing
☎ 08121/769-300**

**DMV Daten- und Medienverlag
Fuldaer Straße 6 • 37269 Eschwege
☎ 05651/809-375**

**DMV Daten- und Medienverlag
Postfach 130309 • 20103 Hamburg
☎ 040/40 44 12 05**

Das MOD-Dateiformat unter der Lupe

Das Format der MOD-Dateien stammt ursprünglich vom Commodore Amiga, fand mit der Verbreitung der Soundblaster-Karte aber auch seinen Weg auf den PC. Mit ihm läßt sich digitale Musik ohne großen Speicheraufwand realisieren.

Die den MOD-Dateien zugrundegelegte Technik verwendet »Naturklänge« und nutzt sie als Instrumente. Diese klingen natürlich viel realistischer, als die FM-Synthese der Soundblaster. Das als Sampled-Wave-Technologie bezeichnete Verfahren findet zunehmend auch Unterstützung durch die Hardware, so zum Beispiel im Wave Blaster von Creative Labs oder in der neuen Gravis Ultrasound. Betrachten wir jedoch den zur Verfügung stehenden Sample-Kanal der Soundblaster genauer. Wie ist also eine MOD-Datei aufgebaut? Sie läßt sich in drei Teile untergliedern:

Der Header

Die Dateikopfinformationen sind 1084 Byte lang und enthalten alle für das Stück erforderlichen Informationen:

● Byte 0 bis 19 – Songname

In den ersten 20 Bytes des Songs steht der Songname als C-Zeichenkette, das letzte Byte muß also 0 sein.

● Byte 20 bis 41 – Name des ersten Instruments

In diesen 22 Bytes befindet sich der Name des ersten Instruments als C-String im ASCII-Format, das letzte Zeichen muß somit wiederum 0 sein.

● Byte 42 bis 43 – Länge des Instruments

In diesen beiden Bytes befindet sich die Länge des ersten Instruments. Die Angabe bezieht sich auf Worte; um die Bytezahl zu erhalten, muß der Wert noch mit 2 multipliziert werden. Des weiteren ist zu beachten, daß es sich hier um eine ursprünglich vom Amiga stammende Datei handelt. Der 68000-Prozessor speichert einen Wert in der Reihenfolge Hi-Byte/Lo-Byte und nicht wie die Prozessoren der 80x86-Reihe als Lo-Byte/Hi-Byte. Darum sind Hi- und Lo-Byte zu vertauschen, um den korrekten Wert zu erhalten.

● Byte 44 – Fine Tune

Nur die unteren vier Bits sind hier von Bedeutung. Sie geben einen Wert zwischen

-8 und 8 an, der es erlaubt, eine Feinabstimmung des Instruments vorzunehmen. Dieser Parameter findet allerdings nur sehr selten Verwendung.

● Byte 45 – Volume

Dieses Byte gibt die voreingestellte Lautstärke des Instruments an. Zulässige Werte liegen zwischen 0 und 64.

● Byte 46 bis 47 – Loop-Start

Das MOD-Format gestattet es, ein Sample zu wiederholen. Die beiden Bytes bezeichnen den Start des Loop. Auch hier ist wieder darauf zu achten, daß es sich um eine Amiga-Word-Angabe handelt, also Hi- und Lo-Byte zu tauschen sind, und das Ergebnis mit 2 zu multiplizieren ist.

● Byte 48 bis 49 – Loop-Länge

In diesen beiden Bytes steht die Länge des Loop. Auch hier handelt es sich um eine Amiga-Word-Angabe.

Die Informationen der Bytes 20 bis 49 wiederholen sich jetzt für die restlichen 30 Instrumente im neuen beziehungsweise 14 Instrumente im alten MOD-Format. Wir setzen hier das verbreitete neue MOD-Format voraus.

● Byte 950 – Liedlänge

Dieses Byte bezeichnet die Länge des Liedes in Pattern. Es enthält die tatsächliche Länge des Liedes und nicht die Anzahl der definierten Pattern.

● Byte 951 – CIAA-Speed

Dieses Byte ist Amiga-spezifisch und somit für den PC-Einsatz ohne Bedeutung.

● Byte 952 bis 1079 – Liedarrangement

Jedes der 128 Bytes darf einen Wert zwischen 0 und 63 annehmen, der das zu spielende Pattern bezeichnet. Aus diesen 128 Bytes und der Liedlänge ergibt sich das Arrangement des Liedes.

● Byte 1080 bis 1083 – Kennung

In diesen vier Bytes steht die Kennung der MOD-Datei. Findet sich der Wert M.K. oder FLT4, so handelt es sich um eine vierstimmige MOD-Datei mit 31 Instrumenten. Auf dem PC sind auch noch neue Formate entstanden, so steht zum Beispiel

Effekte bei MOD-Sample-Daten auf einen Blick

Effekt	Name	Parameter
00	Arpeggio	2
01	Portamento Up	1
02	Portamento Down	1
03	Portamento to Note	1
04	Vibrato	2
10	Volume Sliding	1
11	Position Jump	1
12	Set Note Volume	1
13	Pattern Break	0
15	Set Speed	1

6CHN für eine sechsstimmige und 8CHN für eine achsstimmige MOD-Datei.

Die Pattern

● Byte 1084 bis 2107

In diesem Byte befindet sich das erste Pattern. Eine Note eines Patterns besteht aus jeweils vier Bytes, vier Noten sind pro Zeile definiert. Jedes Pattern enthält 64 Zeilen. Daraus ergibt sich die Gesamtlänge eines Patterns von 1024 Byte. Die Anzahl der Pattern zu bestimmen ist etwas schwierig, sie muß berechnet werden. Dazu zieht man zunächst von der Gesamtlänge der MOD-Datei die Länge des Headers ab. Anschließend ist die Länge der einzelnen Samples aufzuaddieren und diese ebenfalls von der Länge der Datei abzuziehen. Den verbleibenden Rest dividieren Sie dann durch die Pattern-Größe (1024), woraus die Anzahl der Samples resultiert. Die Pattern lassen sich nun problemlos einlesen.

Die Sample-Daten

● Byte n1-nx

An die Pattern anschließend befinden sich die Sample-Daten im Format »signed Raw-Data«. Das heißt, daß es sich um reine 8-Bit-Sample-Daten handelt, die Werte zwischen -127 und 128 einnehmen. Auf dem PC sind die Daten allerdings zwischen 0 und 255 definiert, die Sample-Daten müssen deshalb vor der Ausgabe noch konvertiert werden.

Das Format einer MOD-Note

● Byte 1

Die oberen vier Bits (7 bis 4) sind als Hi-Bits der Instrumentennummer zu interpretieren. Die unteren vier Bits (3 bis 0) gehören mit zur Tonhöhenangabe.

● Byte 2

Byte 2 enthält zusammen mit den Bits 3 bis 0 des ersten Bytes die Angabe der Tonhöhe, die in Samples pro Minute gemessen wird.

● Byte 3

In den oberen vier Bits (7 bis 4) finden wir die Lo-Bits der Instrumentennummer. In den unteren vier Bits ist der Effekt-Befehl untergebracht. Eine Übersicht aller Effekte geht aus der Tabelle hervor.

● Byte 4

Byte 4 enthält die Operanden des Effekt-Befehls.

Effekte

● Effekt 00 –

Arpeggio (zwei Operanden)

Der Arpeggio-Effekt bewirkt, daß die Note nicht durchgängig in einer Tonhöhe gespielt wird, sondern in drei verschiedenen. Dabei bezeichnet der erste Parameter die Differenz zwischen den ersten beiden und der zweite Parameter die Differenz zwischen den zweiten beiden Tönen. Als Maß der Differenz dienen Halbtonschritte.

● Effekt 01 –

Portamento Up (ein Operand)

Dieser Effekt steigert die Tonhöhe bestän-

dig. Die Geschwindigkeit der Erhöhung bestimmt der Operand.

● Effekt 02 –

Portamento Down (ein Operand)

Portamento Down verringert die Tonhöhe kontinuierlich. Die Geschwindigkeit der Verringerung legt der Operand fest.

● Effekt 03 –

Portamento to Note (ein Operand)

Anders als bei den Effekten 1 und 2 kündigt sich Portamento to Note um die Angabe der Zieltonhöhe. Der Operand bestimmt dabei wiederum die Geschwindigkeit.

● Effekt 04 –

Vibrato (zwei Operanden)

Mit diesem Effekt läßt sich ein Vibrato erzielen. Dabei gibt der erste Operand die Geschwindigkeit und der zweite die Tiefe des Vibratos an.

● Effekt 10 –

Volume Sliding (ein Operand)

Dieser Effekt verringert oder erhöht die

Lautstärke eines Instruments gemäß der Vorgabe des Operanden.

● Effekt 11 –

Position Jump (ein Operand)

Die Ausgabe des aktuellen Patterns wird abgebrochen und in dem im Operanden angegebenen Pattern fortgesetzt.

● Effekt 12 –

Set Note Volume (ein Operand)

Mit diesem Effekt läßt sich die Lautstärke einer Note wählen. Der dem Operanden übergebene Wert darf zwischen 0 und 63 liegen.

● Effekt 13 –

Pattern Break (kein Operand)

Pattern Break veranlaßt den Abbruch der Ausgabe des aktuellen Patterns und fährt mit dem nächsten fort.

● Effekt 15 – Set Speed (ein Operand)

Dieser Effekt setzt die Geschwindigkeit, mit der Sie die MOD-Datei abspielen. Die erlaubten Werte für den Operanden liegen zwischen 0 und 31.

(Boris Bertelsohn/bj)

Impressum

Herausgeber: Michael Scharfenberger

Redaktion

Chefredakteur: Ulrich Eike (ue)

Chefin vom Dienst: Gabi Börger (gb)

Leitender Redakteur: Wolfgang Hofner (wh)

Redaktion:

Herbert Buckel (bj), Burkhard Müller (bm),

Helmut Stult (hs), Kerrin Uhlmann (uk)

Textredaktion: Anne Brückner (br), Vera Eich (ve)

Redaktionsassistent:

Kathrin Nagy, Andrea Rutzmoser, Indra Schreiber

Gestaltung und DTP-Layout:

Marcus Geppert, Michael Grebenstein, Andrea Kloss,

Cristiana Seiser, Martina Siebert

Leonhard Büttner (Grafik), Heinz Harcuba (Fotografie)

Titelgestaltung: Arno Krämer, Erich Schulze

Anschrift der Redaktion:

Redaktion DOS International,

Gruber Str. 46a, 85586 Poing bei München,

Telefon: (08121) 769-0, Telefax: (08121) 79046

Sie erreichen die Redaktion werktags telefonisch von 15 bis 17 Uhr, freitags bis 16.30 Uhr.

Manuskripte, Programme bitte an die Redaktion DOS International, Gruber Str. 46a, 85586 Poing bei München, senden.

Für unverlangt eingesandte Manuskripte und Datenträger sowie Fotos übernehmen der Verlag und die Redaktion keine Haftung. Die Zustimmung zum Abdruck wird vorausgesetzt. Das Urheberrecht für veröffentlichte Manuskripte liegt ausschließlich beim Verlag. Nachdruck sowie Vervielfältigung oder sonstige Verwertung von Texten nur mit schriftlicher Genehmigung des Verlages. Für veröffentlichte Programme und Basteleien übernehmen wir weder Gewähr noch Haftung. Schaltungen und verwendete Bezeichnungen müssen nicht frei von gewerblichen Schutzrechten sein. Namentlich gekennzeichnete Fremdbeiträge geben nicht in jedem Fall die Meinung der Redaktion wieder.

Anzeigenverkauf

Gesamtanzeigenleitung: Stefan Grajer

Anzeigenverkauf für PLZ 6, 7, 8, 9, A, CH:

DMV Daten- und Medienverlag GmbH & Co. KG,

Gruber Str. 46a, 85586 Poing bei München,

Telefon: (0 81 21) 769-300, Telefax: (0 81 21) 769-399

Leitung: Britta Fiebig (0 81 21) 769-375

Anzeigenverkauf: Katharina Küspert (0 81 21) 769-374,

Alan Markovic (0 81 21) 769-378, Helmut Jäger (0 81 21) 769-379,

Christian Buck (0 81 21) 769-307

Anzeigenverkauf für PLZ 0, 1, 3, 4, 5:

DMV-Verlagsbüro Eschwege, Fuldaer Str. 6, 37269 Eschwege

Leitung: Thomas Goldmann (0 56 51) 809-327

Anzeigenverkauf: Karina Ehrlich (0 56 51) 809-371,

Bernd Heckmann (0 56 51) 809-381, Christina Wabra

(0 56 51) 809-372, Sylvia Stephani (0 56 51) 809-380,

Telefax: (0 56 51) 809-444

Anzeigenverkauf für PLZ 2:

DMV-Verlagsbüro Hamburg, Postfach 130309, 20103 Hamburg,

Telefon: (040) 44 12 05, Fax: (040) 44 12 06

Anzeigenverkauf: Sandra Neubäumer

International Advertising Manager:

Sarah A. Money, Phone: (0 81 21) 769-350, Fax: (0 81 21) 769-377

Coordination Manager Asia:

Ralph Streng, Phone: (040) 5 23 10 63, Fax: (040) 5 28 31 82

Repräsentanten im Ausland:

USA: M+T International Marketing, Ms. Sharon Kazan, 411 Borel

Avenue, Suite 1000, San Mateo CA 94402-3522, Phone: 0415-

358-9500, Fax: 0415-358-9739

Great Britain: Huson European Media, Gerry Rhoades-Brown,

10-11 The Green Business Centre, The Causeway, Staines, Midd-

lesex, TW18 3AL, Phone: (0784) 46 99 00, Fax: (0784) 46 99 96

Taiwan: PRISCO, Mr. Parson Lee, 2F No. 17, Alley 7, Lane

217, Chung Hsiao E. Rd. Sec. 3, Taipei, Phone: (02) 711 48 33-

6, Fax: (02) 741-51 10

Hong Kong: ELECTRONIC (HK) Publ. Co., Mr. Herman Lau,

1301 Po Sang Bank Building, 33 Argyle Street, Kin, Hong

Kong, Phone: 391-2977, 391-5088, Fax: 789-8027

Singapore: Major Media Private Ltd., Mr. Hoo Siew-Sai, 6th Fl.,

52 Chin Swee Road, Singapore, Phone: 738-0122, Fax: 738-2108

Anzeigenposition:

Leitung: Katharina Pickel (Litg.) (0 81 21) 769-340

Claudia Gosse (0 81 21) 769-343

Anzeigenpreise:

Es gilt die Anzeigenpreisliste Nr. 8 vom 01.01.1993

Anzeigenrundpreise:

1/1 Seite s/w DM 4.000,-, Farbzuschlag: erste Zusatzfarbe aus

Europaskala DM 300,- und zweite Zusatzfarbe DM 600,-, Vier-

farbzuschlag DM 1.000,-.

Verlag

Anschrift Verlag: DMV Daten- und Medienverlag

GmbH & Co. KG, Gruber Str. 46a, 85586 Poing,

Telefon: (08121) 769-0, Fax: (08121) 79046

Geschäftsführung: Michael Scharfenberger

Vertriebsleitung: Helmut Grünfeldt

Vertrieb: MZV Moderner Zeitschriften Vertrieb GmbH & Co.

KG, Breslauer Str. 5, Postfach 1123, 85386 Eching,

Tel.: (089) 3 19 00 60

Herstellungsleitung: Otto Albrecht

Lithographie und Belichtung:

Journalatz GmbH, Zittelstr. 6, 80796 München

Druck: Druckerei Jungfer, 37412 Herzberg

So erreichen Sie uns:

Abonnementverwaltung:

DMV-Verlag, DOS Extra, Abonnementverwaltung CSJ,

Postfach 1400220, 80452 München

Tel.: (089) 24 01 32 22, Fax: (089) 24 01 32 15

Abonnementpreise:

Inland: 4 Ausgaben DM 60,-

Studentenpreis: 4 Ausgaben DM 48,-

Europäisches Ausland: 4 Ausgaben DM 70,-

Außereuropäisches Ausland auf Anfrage

Bankverbindungen:

Postscheck Frankfurt/M.: Kontonr. 23043-608

Bayerische Vereinsbank Markt Schwaben,

BLZ 700 202 70, Kontonr. 478 344 49

Abonnementbestellung Österreich:

Alpha Buchhandels GmbH, Rembrandtstraße 9, A-1020 Wien,

Tel.: (02 22) 33 09 754, Fax: (02 22) 33 09 754/20

Preis pro Ausgabe: ÖS 145,- (Einzelheftfaktura)

Einzelheftbestellung:

DMV-Verlag Leserservice, CSJ, Postfach 14 02 20,

80452 München, Tel.: 089/2 40 13-2 22, Fax: 089/2 40 13-2 15

Bestellungen nur mit Angabe des Bankeinzugs oder gegen

Verrechnungsscheck möglich

Diskettenbestellung:

Erdem Development, Postfach 10 05 18, 80079 München

Tel: 089/4 27 10 39, Fax: 089/42 36 08

Bestellungen nur mit Angabe des Bankeinzugs oder gegen

Verrechnungsscheck möglich

Soundblaster unter Kontrolle

Neben dem Digital-Sound-Prozessor, kurz DSP genannt, besitzen die Soundblaster Pro und die Soundblaster 16 einen sogenannten Mixerchip. Dieser Chip erlaubt diverse Einstellungen, die das Verhalten der Soundkarte beeinflussen, und koordiniert die ein- und ausgehenden Signale.

Die Register des Mixerchips liegen an den Port-Adressen 2x4h (Indexregister) und 2x5h (Datenregister). Der Mixerchip wird programmiert, indem man zuerst einen Wert an sein Indexregister sendet und dann über das Datenregister die entsprechenden Daten liest oder schreibt. Prozeduren hierzu finden Sie in der Unit »mixer.pas« (Listing).

Bei der Besprechung des Mixers werden wir nur auf die schon seit der Soundblaster Pro vorhandenen Funktionen eingehen und gegebenenfalls Abweichungen der Soundblaster 16 (ASP) erläutern. Für die Programmierung des Mixers stehen folgende Indexregister zur Verfügung:

● Register 0h – Reset

Dieses Register dient dazu, den Mixerchip in seinen Standardzustand zu versetzen. Gehen Sie dazu folgendermaßen vor:

- Der Wert 0 wird in das Indexregister geschrieben.
- Circa 100 ms Pause benötigt der Reset.
- Schreiben Sie den Wert 0 in das Datenregister.

● Register 02h – DSP-Lautstärke

Dieses Register wird zur Einstellung der Lautstärke des DSP verwendet. Dabei stehen bei der Soundblaster Pro die Bits 7 bis 5, bei der Soundblaster 16 (ASP) die Bits 7 bis 4 für die Lautstärke des linken Kanals. Die Lautstärke kann also auf der Soundblaster Pro einen Wert von 0 bis 7 und auf der Soundblaster 16 (ASP) einen Wert von 0 bis 15 annehmen. Entsprechend dienen die Bits 3 bis 1 auf der Soundblaster Pro beziehungsweise die Bits 3 bis 0 auf der Soundblaster 16 (ASP) für die Einstellung der Lautstärke des rechten Kanals. Zum Setzen der DSP-Lautstärke verwenden Sie folgendes Schema:

- Das Indexregister erhält den Wert 02h.
- Berechnen Sie die Lautstärke für die Soundblaster Pro

$$\text{Lautstärke} = (\text{left sh1 5}) + (\text{right sh1 1})$$
 beziehungsweise für die Soundblaster 16 (ASP)

$$\text{Lautstärke} = (\text{left sh1 4}) + \text{right}$$

wobei *left* für den linken und *right* für den rechten Kanal steht.

- Schreiben Sie nun die Lautstärke in das Datenregister.

● Register 0Ah – Mikrofonlautstärke

Dieses Register ziehen Sie zur Einstellung der Lautstärke des Mikrofons heran. Dabei sind auf der Soundblaster Pro Werte von 0 bis 3, auf der Soundblaster 16 (ASP) von 0 bis 7 erlaubt. Das Setzen der Lautstärke ist folgendermaßen realisierbar:

- Schreiben Sie den Wert 0Ah in das Indexregister.
- Die Lautstärke für die Soundblaster Pro berechnet sich aus

$$\text{Lautstärke} = (\text{Wert sh1 1})$$

beziehungsweise für die Soundblaster 16 (ASP)

- $\text{Lautstärke} = \text{Wert}$
- Der errechnete Wert für die Lautstärke wird in das Datenregister geschrieben.

● Register 0Ch – Eingabe- und Filter-Einstellungen

Dieses Register regelt Einstellungen für die Aufnahme. Hiermit lassen sich ein Filter und die Eingabequelle wählen. Ist Bit 5 nicht gesetzt, ist der Filter eingeschaltet, ansonsten bleibt er inaktiv. Bit 3 regelt die Durchlaßhöhe des Filters; ist Bit 3 gesetzt, so ist sie tief, sonst hoch. Die Bits 2 und 1 bestimmen die Eingabequelle. Dabei steht ein Wert von 0 für Mikrofoneingabe, 1 für Eingabe von CD-ROM und 3 für Line-Input. Zum Setzen des Registers gehen Sie wie folgt vor:

- Das Indexregister enthält den Wert 00C
- Berechnen Sie den Einstellungswert EW folgendermaßen:
Zum Filter einschalten:
 $\text{EW} := (1 \text{ sh1 } 5)$
Für Durchlaßhöhe *Tief*:
 $\text{EW} := \text{EW} + (1 \text{ sh1 } 3)$
und schließlich zur Wahl der Eingabequelle *Quelle*:
 $\text{EW} := \text{EW} + (\text{Quelle sh1 } 1)$
- Den berechneten Wert der gewünschten Einstellung schreiben Sie nun ins Datenregister.

● Register 0Eh – Ausgabefilter und Stereowahl

Dieses Register besitzt zwei Funktionen: Zum einen dient es dazu, festzulegen, ob ein Ausgabefilter gewählt werden soll. Zum anderen schaltet es zwischen Mono- und Stereoausgabe um.

Ist Bit 5 nicht gesetzt, ist der Filter aktiv, ansonsten bleibt er ausgeschaltet. Ist Bit 1 gesetzt, erfolgt die Ausgabe in Stereo, andernfalls in Mono. Zum Setzen der Einstellungen verwenden Sie folgendes Schema:

- Initialisieren Sie das Indexregister mit dem Wert 00E.
- Zur Aktivierung eines Ausgabefilters:
 $\text{Reg-Wert} := (1 \text{ sh1 } 5)$
Sofern die Ausgabe in Stereo erfolgen soll:
 $\text{Reg-Wert} := \text{Reg-Wert} + (1 \text{ sh1 } 1)$
- Schreiben Sie den berechneten Registerwert nun in das Datenregister.

● Register 22h – Masterlautstärke

Dieses Register wird zur Einstellung der allgemeinen Lautstärke herangezogen. Dabei stehen bei der Soundblaster Pro die Bits 7 bis 5, bei der Soundblaster 16 (ASP) die Bits 7 bis 4 für die Lautstärke des linken Kanals. Die Lautstärke darf also auf der Soundblaster Pro einen Wertebereich von 0 bis 7 und auf der Soundblaster 16 (ASP) Werte von 0 bis 15 annehmen. Entsprechend stehen die Bits 3 bis 1 auf der Soundblaster Pro und die Bits 3 bis 0 auf der Soundblaster 16 (ASP) für die Einstellung der Lautstärke des rechten Kanals. Zum Setzen der Masterlautstärke gehen Sie folgendermaßen vor:

- Das Indexregister erhält den Wert 22h.
- Berechnen Sie die Lautstärke für die Soundblaster Pro

$$\text{Lautstärke} = (\text{left sh1 5}) + (\text{right sh1 1})$$

beziehungsweise für die Soundblaster 16 (ASP)

$$\text{Lautstärke} = (\text{left sh1 4}) + \text{right}$$

wobei *left* für den linken und *right* für den rechten Kanal steht.

- Schreiben Sie nun die Lautstärke in das Datenregister.

● Register 26h – FM-Lautstärke

Dieses Register regelt die Einstellung der Lautstärke des FM-Kanals. Ansonsten gilt dasselbe wie für Register 22h (Masterlautstärke). Das Setzen der Lautstärke wird wie folgt realisiert:

- Das Indexregister erhält den Wert 26h.
- Berechnen Sie die Lautstärke für die Soundblaster Pro

$$\text{Lautstärke} = (\text{left sh1 5}) + (\text{right sh1 1})$$

beziehungsweise für die Soundblaster 16 (ASP)

$$\text{Lautstärke} = (\text{left sh1 4}) + \text{right}$$

wobei *left* für den linken und *right* für den rechten Kanal steht.

- Schreiben Sie nun die Lautstärke in das Datenregister.

● Register 28h – CD-Lautstärke

Dieses Register dient zur Einstellung der Lautstärke des CD-Kanals. Ansonsten gilt dasselbe wie für Register 22h (Masterlautstärke). Zum Setzen der CD-Lautstärke verwendet man folgende Technik:

- Das Indexregister erhält den Wert 28h.
- Berechnen Sie die Lautstärke für die Soundblaster Pro

```
Lautstärke = (left shl 5) +
              (right shl 1)
```

beziehungsweise für die Soundblaster 16 (ASP)

```
Lautstärke = (left shl 4) +
              right
```

wobei *left* für den linken und *right* für den rechten Kanal steht.

- Schreiben Sie nun die Lautstärke in das Datenregister.

● Register 2Eh – Line-Lautstärke

Dieses Register beeinflusst die Lautstärke des Line-Kanals. Ansonsten gilt dasselbe wie für Register 22h (Masterlautstärke). Zum Setzen der Line-Lautstärke benötigen Sie folgende Schritte:

- Das Indexregister erhält den Wert 2Eh.
- Berechnen Sie die Lautstärke für die Soundblaster Pro

```
Lautstärke = (left shl 5) +
              (right shl 1)
```

beziehungsweise für die Soundblaster 16 (ASP)

```
Lautstärke = (left shl 4) +
              right
```

wobei *left* für den linken und *right* für den rechten Kanal steht.

- Schreiben Sie nun die Lautstärke in das Datenregister.

(Boris Bertelsons/hs/bj)

Listing : mixer.pas

```
1: {
2: Programm: Unit Mixer
3: Funktion: Bereitstellung und Demon-
4: stration der Mixer-Funktionen
5: Sprache: Turbo Pascal ab 6.0
6: Autor: Boris Bertelsons
7: (c) 1993 DMV Widuch GmbH & Co. KG
8: }
9:
10: UNIT MIXER;
11:
12: interface
13:
14: CONST SB16 : boolean = false;
15:       Hoch = False;
16:       Tief = True;
17:       Mikro = 0;
18:       CD = 1;
19:       Line = 3;
20:
21: VAR m_adr : word;
22:
23: Procedure Set_Mixervalues
24: (dsp_adr:word;
25: Sbttyp : boolean);
26: PROCEDURE Write_Mixer(Reg, Val: BYTE);
27: FUNCTION Read_Mixer(Reg: BYTE): BYTE;
28: procedure reset_Mixer;
29: procedure Set_DSPVolume
30: (left,right:byte);
31: procedure Set_MicVolume(vol : byte);
32: procedure Set_Inputfilter
33: (Filter,Durchlass
34: : boolean; Quelle : byte);
35: procedure Set_Outputfilter(Filter
36: : boolean);
```

```
34: Procedure Set_Stereo(Stereo: boolean);
35: procedure Set_Volume(left,right:
36: byte);
37: procedure SetFmVolume
38: (left,right: byte);
39: procedure SetCDVolume
40: (left,right: byte);
41: procedure SetLineVolume
42: (left,right: byte);
43: implementation
44: 41:
45: Procedure Set_Mixervalues
46: (dsp_adr: word;
47: Sbttyp: boolean);
48: begin;
49: m_adr:= dsp_adr;
50: Sb16 := Sbttyp;
51: end;
52: 48:
53: PROCEDURE Write_Mixer(Reg, Val: BYTE);
54: {
55: Schreibt den in Val übergebenen
56: Wertan
57: das in Reg angegebene Register des
58: Mixer - Chips
59: }
60: begin;
61: Port[m_adr+$4]:= Reg;
62: Port[m_adr+$5]:= Val;
63: END;
64: 62: FUNCTION Read_Mixer(Reg: BYTE): BYTE;
65: {
66: Die Function liefert den Inhalt des
67: über
68: Reg indizierten Registers des Mixer-
69: Chips
70: }
71: begin;
72: Port[m_adr+$4]:= Reg;
73: Read_Mixer:= Port[m_adr+$5];
74: end;
75: 72: procedure reset_Mixer; assembler;
76: {
77: Resetet den Mixer Chip auf seine
78: Default - Werte
79: }
80: asm
81: mov dx,m_adr+$4
82: mov al,0
83: out dx,al
84: mov cx,50
85: @loop:
86: loop @loop
87: inc dx
88: out dx,al
89: end;
90: 88: procedure Set_DSPVolume
91: (left,right: byte);
92: Var Hilfe: byte;
93: begin;
94: if SB16 then begin;
95: Hilfe:= (left shl 4)+right;
96: Write_Mixer($02,Hilfe);
97: end else begin;
98: Hilfe:= (left shl 5)+(right shl 1);
99: Write_Mixer($02,Hilfe);
100: end;
101: 100: procedure Set_MicVolume(vol: byte);
102: Var Hilfe: byte;
103: begin;
104: if SB16 then
105: Hilfe:= vol shl 1;
106: Write_Mixer($0A,Hilfe);
107: end;
108: 110: procedure Set_Inputfilter
109: (Filter,Durchlass
110: : boolean; Quelle: byte);
111: {
112: Filter = TRUE für on und FALSE für
113: off
114: Durchlass = HOCH oder TIEF
115: Quelle = MIKRO,CD oder LINE
116: }
```

```
117: var Hilfe: byte;
118: begin;
119: Filter:= NOT Filter;
120: (Vertauschen wgn. Default-Wert=E in);
121: Hilfe:= byte(Filter) shl 5;
122: Hilfe:= Hilfe +
123: (byte(Durchlass) shl 3);
124: Hilfe:= Hilfe + (Quelle shl 1);
125: write_Mixer($0C,Hilfe);
126: end;
127:
128: Procedure Set_Outputfilter(Filter
129: : boolean);
130: var Hilfe,Oldv: byte;
131: begin;
132: Filter:= NOT Filter;
133: Oldv:= Read_Mixer($0E);
134: Oldv:= Oldv AND 2;
135: { Alles bis auf BIT1 raus }
136: Hilfe:= byte(Filter) shl 5;
137: Hilfe:= Hilfe OR Oldv;
138: Write_Mixer($0E,Hilfe);
139: end;
140:
141: Procedure Set_Stereo(Stereo: boolean);
142: var Hilfe,Oldv: byte;
143: begin;
144: Stereo:= NOT Stereo;
145: Oldv:= Read_Mixer($0E);
146: Oldv:= Oldv AND $20;
147: { Alles bis auf BIT5 = 0 }
148: Hilfe:= byte(Stereo) shl 1;
149: Hilfe:= Hilfe OR Oldv;
150: Write_Mixer($0E,Hilfe);
151: end;
152:
153: procedure Set_Volume
154: (left,right: byte);
155: {
156: Zulässige Werte zwischen 0..7 auf SB
157: Pro
158: und 0..15 auf SB 16
159: }
160: Var Hilfe: byte;
161: begin;
162: if SB16 then
163: Hilfe:= (left shl 4)+right
164: else
165: Hilfe:= (left shl 5)+(right shl 1);
166: Write_Mixer($22,Hilfe);
167: end;
168: 167: procedure SetFmVolume
169: (left,right: byte);
170: var Hilfe: byte;
171: begin;
172: if SB16 then
173: Hilfe:= (left shl 4)+right
174: else
175: Hilfe:= (left shl 5)+(right shl 1);
176: Write_Mixer($26,Hilfe);
177: end;
178: 177: procedure SetCDVolume
179: (left,right: byte);
180: var Hilfe: byte;
181: begin;
182: if SB16 then
183: Hilfe:= (left shl 4)+right
184: else
185: Hilfe:= (left shl 5)+(right shl 1);
186: Write_Mixer($28,Hilfe);
187: end;
188: 187: procedure SetLineVolume
189: (left,right: byte);
190: var Hilfe: byte;
191: begin;
192: if SB16 then
193: Hilfe:= (left shl 4)+right
194: else
195: Hilfe:= (left shl 5)+(right shl 1);
196: Write_Mixer($2E,Hilfe);
197: end;
198: 197: begin;
199: end.
```

Die Unit »mixer.pas« enthält Prozeduren, um die Grundeinstellungen von Soundblaster-Karten zu beeinflussen.

Sounds aus dem Hintergrund

Die hier vorgestellte Turbo-Pascal-Unit »Modplay« erlaubt es, MOD-Dateien im Hintergrund abzuspielen und gleichzeitig Soundeffekte auszulösen. Dies ist besonders bei selbstgeschriebenen Spielen von großem Interesse, denn von Tönen und Geräuschen hängt unter anderem auch die Spielbegeisterung des Anwenders ab.

Die durch die Unit »modplay.pas« (Listing 1) abgespielte MOD-Datei könnte beispielsweise parallel zum Programmablauf eines Adventure-Programms an entsprechender Stelle das Knarren einer Tür ausgeben. Auch das Rauschen eines Baches, das Zwitschern von Vögeln oder das Hämmern einer Schmiede wären denkbar.

Als Beispiel für den Einsatz der Unit dient das Programm »mod_demo.pas« (Listing 2), das einen einfachen MOD-Player darstellt. Sie starten dieses Demoprogramm mit Angabe einer MOD-Datei, also beispielsweise

```
mod_demo hurrica2.mod
```

Anhand des Programms wird deutlich, wie Sie vorgehen, um eine bestehende MOD-Datei abzuspielen. Zuerst sind einige grundlegende Prozeduren aufzurufen, insbesondere *mod_Setloopflag*, die festlegt, ob die MOD-Datei nach der Wiedergabe erneut gespielt wird, *init_sb* zum Initialisieren des Soundblasters und *mod_Samplefreq* zum Festlegen der Sample-Frequenz in kHz.

Das Abspielen einer MOD-Datei

Nachdem Sie die Grundeinstellungen getroffen haben, steht dem Laden einer MOD-Datei in den Speicher nichts mehr im Wege. Dies geschieht über die Funktion *lade_moddatei*, die neben dem DOS-Namen der Datei auch noch wahlweise zwei Parameter zur Geschwindigkeitsbestimmung akzeptiert. Im Normalfall sollten jedoch beide Werte auf »auto« gesetzt bleiben.

Anschließend lassen sich noch, je nach zur Verfügung stehendem Speicher, mit der Funktion *lade_Soundeffekt* beliebig viele Soundeffekte laden. Das Abspielen der MOD-Datei startet die Prozedur *periodisch_on*. Nach Aufruf dieses Moduls sollte folgendes beachtet werden:

1. Unter keinen Umständen den Befehl *CLI* benutzen. Die Mischprozedur wird periodisch vom Timer-Interrupt aufgerufen, weswegen die Interrupts auf keinen Fall gelöscht werden dürfen.
2. Lange Diskettenzugriffe sollten Sie vermeiden

und Daten vorher gegebenenfalls zwischenpuffern.

3. Zum Warten auf den »Vertical Retrace« unbedingt die Prozedur *mod_waitretrace* verwenden, da nur sie eine problemlose Synchronisation gestattet.

Um die Ausgabe der MOD-Datei anzuhalten, setzt man den Befehl *periodisch_off* ein. Die MOD-Datei verbleibt im Speicher, und die Ausgabe läßt sich mit *periodisch_on* zu einem späteren Zeitpunkt fortsetzen.

Zum Beenden einer MOD-Session gehen Sie folgendermaßen vor:

1. Schalten Sie das Abspielen durch *periodisch_off* aus.

2. Entfernen Sie die MOD-Datei mit *ende_mod*.

3. Sofern Soundeffekte geladen sind, entfernen Sie diese über die Prozedur *dispose_Soundeffekt* wieder aus dem Speicher.

4. Sollte die Soundkarte eine Version vor Soundblaster 16 sein, so muß noch der Befehl

```
port[dsp_adr+$c] := 1;  
erfolgen.
```

Soundeffekte

Soundeffekte steigern die Atmosphäre und sind damit sicherlich für viele Anwendungen interessant. Um einen Effekt zu verwenden, müssen Sie diesen zuvor mit *Lade_Soundeffekt* geladen haben. Zum Starten des Effekts benutzen Sie die Prozedur *Starte_Soundeffekt*. Diesem Modul wird zuerst das Handle des Soundeffekts und anschließend die Sample-Rate in Hertz (Hz) übergeben. Der dritte Parameter bezeichnet die Lautstärke und darf einen Wert von 0 bis 63 annehmen. Der vierte Parameter gibt schließlich den Typ des Samples an, zulässig sind »PC« oder »Amiga«. Dabei steht »PC« für unsigned Sample-Daten und »Amiga« für signed Sample-Daten.

Dies ist alles, was Sie zum Einsatz der Unit wissen müssen. Der Verlust an System-Performance liegt bei zirka 20 bis 30 Prozent auf allen Computern ab 386DX und ist abhängig von der gewählten Sample-Rate (je kleiner, desto weniger Rechenzeit).

Um eine Synchronisation zwischen dem Anschlag der Stimmen und den Effekten zu erreichen, können Sie auf die globalen Variablen *Noten_Anschlag[Stimme]* zugreifen. Zum Zeitpunkt des Anschlags enthält die Variable einen Wert von 500, der von der Mischprozedur bis auf 0 herabgezählt wird.

Zentrale Prozeduren und Variablen

Wenn der Einsatz von *CLI* (Interrupts löschen) unverzichtbar ist, so läßt sich das Vermischen der Stimmen mit einem »ständigen« Aufruf (Schleife) von *calculate_Music* simulieren. Die Gefahr, daß es hierbei zu Störungen kommt, ist jedoch recht hoch. Das Laden eines Patterns erfolgt in den Prozeduren *nmw_all_4* beziehungsweise *nmw_all_8*. Wenn Sie das Abspielen von sechsstimmigen MOD-Dateien integrieren möchten, wäre hier ein entsprechende eigene Programmsequenz einzufügen.

Dem Vermischen der Stimmen widmen sich die Prozeduren *Innen_Schleife_4* beziehungsweise *Innen_Schleife_8*. Diese beiden Module füllen den Puffer *blk* mit Daten, welche die Routine *bsp_block_sb16* später ausgibt.

Für die Verwendung der Unit in eigenen Applikationen ist es wichtig zu wissen, daß »modplay.pas« bei der Ausgabe den Soundblaster-Interrupt »abfängt«. Dies ist erforderlich, da es sonst zum Soundblaster-typischen Knacken kommen kann. Leider ist jedoch auch der hier benutzte Ansatz, nur 90 Prozent der Sample-Daten zu spielen und dann die nächste Ausgabe zu starten, nicht ganz unproblematisch. In diesem Fall geht nämlich ein Bruchteil der Informationen verloren, was sich beim Looping von Strings jedoch bemerkbar macht. Da dies aber bei weitem angenehmer ist als das ständige durch den Soundblaster-Interrupt verursachte Knacken, fiel die Entscheidung zugunsten des Datenverzichtes.

Bei der vorliegenden Unit handelt es sich um ein dynamisch gewachsenes Unterprogramm, das weitgehend geschwindigkeitsoptimiert arbeitet. Die gute Klangqualität ist der gemischtsprachigen Programmierung (Assembler/Turbo Pascal) zu verdanken. Sie ließe sich sogar noch etwas steigern, wenn man auf Soundeffekte verzichtet und der Wert *Output* dann nur um 2 verschoben wird. An diese und ähnliche sicherlich noch umsetzbare Optimierungen sollten sich allerdings nur »gestandene« Programmierer heranwagen, da die Unit bei unsachgemäßen Änderungen sonst leicht instabil werden könnte. Es ist empfehlenswert, beim Einsatz der Unit und damit auch des Beispielprogramms »mod_demo.pas« für ausreichend vorhandenen Hauptspeicher zu sorgen. Um Inkompatibilitäten bezüglich des Timings mit Speichermanagement-Software oder Treibern auszuschließen, sollten Sie für den Unit-Einsatz weitgehend auf Programme dieser Art verzichten.

(Boris Bertelsohn/hs/bj)

Listing 1: modplay.pas

```

1: {Programm: modplay.pas
2: Funktion: MOD-Dateien im Hintergrund
3: abspielen
4: Sprache : Turbo Pascal ab 6.0
5: Autor : Boris Bertelons
6: (c) 1993 DMV Daten- u. Medienverlag}
7:
8: {SA+,B-,D+,E+,F-,G+,I-,L+,N-,O-,P-,}
9: {SQ-,R-,S-,T-,V-,X-,Y+,}
10: {SM 16384,0,655360}
11: unit modplay;
12: interface uses crt,dos;
13:
14: {define Typ1}
15: TYPE
16:   pt = record
17:     ofs,sgm : word;
18:   end;
19: { ermöglicht die einfache Behandlung
20: von Pointern }
21: Effect_Type = record
22:   p : pointer;
23:   l : longint;
24: end;
25: { Typ für Soundeffekte }
26: Param_Table = record
27:   mult : word;
28:   Speed : word;
29:   bgr : word;
30:   Ab : integer;
31: end;
32: { Für Speed-Änderungen }
33:
34: CONST
35:   frames = 37;
36:   waitret_flag : byte = 0;
37:   PC : word = 0;
38:   AMIGA : 1;
39:   Choose_lower_freq : boolean =
40:     false;
41: {Wird auf True gesetzt, wenn der PC
42: für die engestellte Samplingrate zu
43: langsam arbeitet !}
44:   timer_per_second : word = 1000;
45: {Anzahl Interrupts per Sec.}
46:   Sampling_Frequenz : word = 10000;
47: {Die Samplingfreq.; Default}
48:   in_retrace : boolean =
49:     false;
50: {Gerade in Retracing-Proc ?}
51:   dsp_irq : byte = $5;
52: {Interrupt des SB, Wert wird durch
53: die Init-Routine geändert}
54:   dma_ch : byte = 1;
55: {DMA Chanel, standardmäßig = 1, auf
56: SB 16 ASP auch andere Werte
57: möglich}
58:   dsp_adr : word = $220;
59: {Die Base-Adresse des DSP. Wert wird
60: durch die Init Routine geändert}
61:   SbVersMin : BYTE = 0;
62: {Die Versions - Kennung}
63:   SbVersMaj : BYTE = 0;
64: {STEREO : BOOLEAN = FALSE;
65: {In Stereo abspielen}
66:   SbRegDetected : BOOLEAN = FALSE;
67: {normale SB vorhanden ?}
68:   IRQDetected : BOOLEAN = FALSE;
69:   SbRegInitd : BOOLEAN = FALSE;
70:   SbProDetected : BOOLEAN = FALSE;
71: {SB Pro vorhanden ?}
72:   SbProInitd : BOOLEAN = FALSE;
73:   Sb16Detected : BOOLEAN = FALSE;
74: {SB 16 ASP vorhanden ?}
75:   Sb16Initd : BOOLEAN = FALSE;
76:   MixerDetected : BOOLEAN = FALSE;
77: {Wenn ja, Karte >= SB Pro}
78:   OldTimerInt : $71;
79: {Original Int-Routine des verbogenen
80: Timer - Int.}
81:   Stimmen : integer = 4;
82: {Anzahl der Stimmen im MOD-File}
83:   Modoktave : array[1..60] of word =
84:   (
85:     1712,1616,1525,1440,1359,1283,1211,
86:     1143,1078,961,907,856,808,763,720,
87:     679,641,605,571,539,509,480,453,428,
88:     404,381,360,340,321,303,286,270,254,
89:     240,227,214,202,191,180,170,160,151,
90:     143,135,127,120,113,107,101,95,90,
91:     85,80,76,71,67,64,60,57,0);
92: {Die Werte in Modoktave entsprechen
93: den im MOD-File als Tonnöhen
94: gespeicherten
95: Werten}
96:   Vkt : array[1..84] of word =
97:   (
98:     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
99:     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
100:    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
101:    0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
102:    1,1,1,2,2,2,2,2,2,2,2,3,3,3,3,3,
103:    3,3,3,3,4,4,4,4,4,5,5,5,5,5,5,5,
104:    6,6,6,7,7,8,8,8,9,10,10,10,10,
105:    11,11,12,13,14,15,16);
106:
107: {Die Vorkomma - Stellen der Frequenz-
108: abhängigen Sprünge}
109:   Nkt : array[1..84] of byte =
110:   (
111:     12,13,14,15,16,17,18,19,21,22,24,
112:     25,26,28,30,31,33,35,37,40,42,45,
113:     47,50,53,56,59,63,67,71,75,79,84,
114:     89,94,00,06,12,19,26,33,41,50,59,
115:     68,78,89,00,12,24,38,52,67,83,00,
116:     17,36,56,78,00,24,49,76,04,34,66,
117:     00,35,73,13,55,00,48,98,51,08,68,

```

```

118: 31,99,70,45,25,10,00
119: );
120: { Die Nachkomma - Stellen der
121: Frequenz-abhängigen Sprünge }
122:   Ndfff : array[1..84] of byte =
123:   (
124:     1,1,1,1,1,1,1,2,1,2,1,
125:     1,2,2,1,2,2,2,3,2,3,2,
126:     3,3,3,3,4,4,4,4,4,5,5,
127:     5,6,6,6,7,7,7,8,9,9,9,
128:     10,11,11,12,12,14,15,16,17,17,
129:     19,20,22,22,24,25,27,28,30,32,34,
130:     35,38,40,42,45,48,50,53,57,60,63,
131:     68,71,75,80,85,90,0
132:   );
133: { Differenzen zwischen den Nachkomma-
134: Stellen }
135:   PermUp_1 : byte = 0;
136:   PermUp_2 : byte = 0;
137:   PermUp_3 : byte = 0;
138:   PermUp_4 : byte = 0;
139:   PermUp_5 : byte = 0;
140:   PermUp_6 : byte = 0;
141:   PermUp_7 : byte = 0;
142:   PermUp_8 : byte = 0;
143: { Portamento Up der Stimme ? }
144:   PermDo_1 : byte = 0;
145:   PermDo_2 : byte = 0;
146:   PermDo_3 : byte = 0;
147:   PermDo_4 : byte = 0;
148:   PermDo_5 : byte = 0;
149:   PermDo_6 : byte = 0;
150:   PermDo_7 : byte = 0;
151:   PermDo_8 : byte = 0;
152: { Portamento Down der Stimme ? }
153:   PermNk1 : word = 0;
154:   PermNk2 : word = 0;
155:   PermNk3 : word = 0;
156:   PermNk4 : word = 0;
157:   PermNk5 : word = 0;
158:   PermNk6 : word = 0;
159:   PermNk7 : word = 0;
160:   PermNk8 : word = 0;
161:   AUTO = 9999;
162: { Kennung für Auto detection }
163:   ON = true;
164:   OFF = false;
165:   playeffect : boolean = false;
166:   effectvolume : byte = 64;
167:   converteff : byte = 0;
168:   Aktspeed : byte = 6;
169:   VAR
170:     efi : file;
171:     ModPara : array[1..31] of
172:       Param_Table;
173:     blockgroesse : word;
174:     { Größe des Sound-Puffers }
175:     dsp_rdy_sb16 : boolean;
176:     { Flag für Ende der Übertragung der
177:       Daten via DMA }
178:     SbVersStr : string[5];
179:     { Die SB-Version als String }
180:     Speed : word;
181:     { Abspielgeschwindigkeit }
182:     oldint : pointer;
183:     { Zur Sicherung des vom SB zum DMA-
184:       Transfer benötigten Interrupts }
185:     irqmsk : byte;
186:     { zur Interruptbehandlung }
187:     Vermische_proc : pointer;
188:     nmw_proc : pointer;
189:     innen_proc : pointer;
190:     { Zeiger auf Routinen, die je nach
191:       Anzahl der vorhandenen Stimmen
192:       ausgeführt werden }
193:     Noten_Anschlag : array[1..8]
194:       of integer;
195:     { Zeit seid letztem Anschlag }
196:     Rm_Song : array[1..64,1..8,
197:       1..4] of Byte;
198:     { Ein Pattern }
199:     rm : array[0..128]
200:       of pointer;
201:     { Die einzelnen Pattern }
202:     Lied : array[1..128]
203:       of byte;
204:     { Arrangement des Liedes }
205:     blk : pointer;
206:     { Pointer auf Daten - Puffer }
207:     inst1,inst2 : pointer;
208:     inst3,inst4 : pointer;
209:     inst5,inst6 : pointer;
210:     inst7,inst8 : pointer;
211:     { Pointer auf die in der jeweiligen
212:       Stimme aktiven Sampels }
213:     Sampeldata : array[1..31]
214:       of pointer;
215:     Samp : array[1..31]
216:       of pointer;
217:     { Feld mit Zeigern auf Sampels }
218:     Sam_l : array[1..31]
219:       of word;
220:     Sam_vol : array[1..31]
221:       of byte;
222:     { Die Länge der Sampels }
223:     loop_s : array[0..31]
224:       of word;
225:     { Loop-Start der Sampels }
226:     loop_l : array[0..31]
227:       of word;
228:     { Loop-Länge der Sampels }
229:     li,li2,li3,li4,
230:     li5,li6,li7,li8 : pt;
231:     { Pointer auf
232:       aktive Sampels in "pt"-Form }
233:     in11,in21,in31,
234:     in41,in51,in61,
235:     in71,in81 : word;
236:     { Länge der Sampeldaten der einzelnen
237:       Stimmen }

```

```

238:   in1p,in2p,in3p,
239:   in4p,in5p,in6p,
240:   in7p,in8p : word;
241: { Position in den jeweiligen
242: Sampeldaten }
243:   i : word;
244:   mlj : word;
245: { Schleifenzähler, aktuelles
246: Pattern }
247:   mli : word;
248: { Schleifenzähler, Aktuelle Zeile im
249: Pattern }
250:   Vkl,Vk2,Vk3,
251:   Vk4,Vk5,Vk6,
252:   Vk7,Vk8 : word;
253: { Vorkommawert des Faktors, um den
254: die Pos. in den Sampeldaten
255: erhöht werden muß }
256:   Nk1,Nk2,Nk3,
257:   Nk4,Nk5,Nk6,
258:   Nk7,Nk8 : byte;
259: { Vorkommawert des Faktors, um den
260: die Pos. in den Sampeldaten
261: erhöht werden muß }
262:   Dif1,Dif2,Dif3,
263:   Dif4,Dif5,Dif6,
264:   Dif7,Dif8 : byte;
265:   Difb1,Difb2,Difb3,
266:   Difb4,Difb5,Difb6,
267:   Difb7,Difb8 : byte;
268:   Inst1vk : word;
269:   Inst2vk : word;
270:   Inst3vk : word;
271:   Inst4vk : word;
272:   Inst5vk : word;
273:   Inst6vk : word;
274:   Inst7vk : word;
275:   Inst8vk : word;
276: { Zeigt auf aktuelles Samplebyte in
277: den Daten }
278:   Inst1nk : byte;
279:   Inst2nk : byte;
280:   Inst3nk : byte;
281:   Inst4nk : byte;
282:   Inst5nk : byte;
283:   Inst6nk : byte;
284:   Inst7nk : byte;
285:   Inst8nk : byte;
286: { Nachkommateil des Samplebytes }
287:   In_St1,In_St2,
288:   In_St3,In_St4,
289:   In_St5,In_St6,
290:   In_St7,In_St8 : byte;
291: { Aktives Instrument der Stimme }
292:   sam_anz : byte;
293: { Anzahl der Sempel }
294:   pat_anz : byte;
295: { Anzahl der Pattern }
296:   m_played : boolean;
297: { Musik gespielt worden ??? }
298:   Sound_Schleifen : word;
299: { Anzahl der Durchläufe der
300: Misch-Proc }
301:   Sampling_Rate : byte;
302: { dem DSP übergebene Frequenz-Wert }
303:   mod_name : string;
304: { DOS-Name der MOD-Datei }
305:   tpw : integer;
306: { Transposer - Wert }
307:   loop_pos : word;
308: { Laufvar., von 0 bis Speed }
309:   phase_1,phase_2 : boolean;
310: { Die zwei Phasen der Interrupt-
311: Mischprozedur }
312:   modmultiply : word;
313: { Speed-Angabe im Song * Modmultiply
314: = Speed }
315:   mloop : boolean;
316: { Wenn TRUE beginnt das MOD nach dem
317: Abspielen von vorn }
318:   periodisch_anhalten : pointer;
319: { Pointer auf Stop_Prozedur für
320: Ausgabe }
321:   music_aus : boolean;
322: { Wenn TRUE wird keine Musik
323: gespielt }
324:   Notvol1,Notvol2,
325:   Notvol3,Notvol4,
326:   Notvol5,Notvol6,
327:   Notvol7,Notvol8 : byte;
328: { Lautstärke der einzelnen Kanäle }
329:   Pnk1,Pnk2,Pnk3,
330:   Pnk4,Pnk5,Pnk6,
331:   Pnk7,Pnk8 : byte;
332: { Nachkomma des Portamento }
333:   Old_TZaeher : word;
334: { Zur Synchronisation des alten
335: Timerinterrupts }
336:   Dma_Zaeher : integer;
337: { Zum abfangen des DMA_Ready
338: Interrupts }
339:   dma_abbruch : integer;
340: { Abbruchwert für Dma_Zaeher }
341:   mod_terminated : boolean;
342: { Mod-Ausgabe beendet }
343:   ls1,ls2,ls3,ls4,
344:   ls5,ls6,ls7,ls8 : word;
345: { Start der Loop im Sempel }
346:   li1,li2,li3,li4,
347:   li5,li6,li7,li8 : word;
348: { Länge der Loop im Sempel }
349:   Songname : string[20];
350: { Der Name des Modfiles }

```

Listing 1. Die Unit »modplay.pas« gestattet die Ausgabe von MOD-Dateien während des Ablaufs von Applikationen.


```

351: Instnamen      : array[1..31]
352:               of string[22];
353: { Namen der Instrumente }
354: Liedlaenge     : byte;
355: { Länge des Liedes }
356: Seczaehler     : word;
357: { zur Zeitermittlung }
358: Timerzaehler   : word;
359: { für alten Timer-Interrupt }
360: Laufsec, Laufmin : byte;
361: { Laufzeit des Liedes }
362: Pattgroesse    : integer;
363: { Die dem Modtyp entspr.
364:   Patterngroße }
365: notel, note2,
366: note3, note4,
367: note5, note6,
368: note7, note8 : byte;
369: { Aktives Instrument der Stimme }
370: effektgroesse  : word;
371: effektposi     : word;
372: Effvk         : word;
373: Effnk         : byte;
374: Efflistvk     : word;
375: Efflistnk     : byte;
376: Effekt        : pt;
377: { Zur Effekt-Behandlung }
378: mycli : byte;
379: { Flag, ob Soundberechnung aktiv }
380:
381: procedure vermische_start_8;
382: procedure nmw_all_8;
383: procedure innen_schleife_8n;
384: procedure vermische_start_4;
385: procedure nmw_all_4;
386: procedure innen_schleife_4;
387: procedure Set_Timeconst_sb16(tc :
388:   byte);
389: procedure initialisiere_vermischen;
390:
391: procedure init_sb;
392: { Die Procedure init_sb initialisiert
393:   die Soundblaster-Karte. Sie erkennt
394:   automatisch Base-Adress und IRQ,
395:   prüft, um welche Soundblasterversion
396:   es sich handelt und setzt
397:   entsprechende globale Variablen,
398:   die z.B. mittels write_sb-
399:   Config ausgegeben werden können
400: }
401: procedure dsp_block_sb16;
402: { Spielt den über blk adressierten
403:   Block via DMA ab
404: }
405: procedure mod_waitretrace;
406: { Das Warten auf einen Bildschirm-
407:   Retrace sollte mit dieser
408:   Procedure erfolgen,
409:   wenn MOD's abgespielt werden
410: }
411: procedure mod_SetSpeed(msp : word);
412: { Setzt die Variable "Speed"
413: }
414: procedure mod_SetMultiply(msm :
415:   word);
416: { Setzt den Multiply-Faktor.
417: }
418: procedure mod_SetLoop(msl : word);
419: { Setzt die Variable
420:   "Sound-Schleifen"
421: }
422: procedure mod_SetLoopflag(loopen :
423:   boolean);
424: { Setzt ON/OFF, ob eine zu Ende abge-
425:   spielte Moddatei wieder von Anfang
426:   an abgespielt wird
427: }
428: procedure mod_transpose(
429:   transposerwert : integer);
430: { Setzt den Transposer-Wert.
431: }
432: procedure mod_Samplefreq(Rate :
433:   integer);
434: { Setzt die Samplefrequenz für die
435:   Ausgabe.
436:   Wert*1000 = Frequenz.
437:   Zulässige Werte sind 8,10,16,22
438: }
439: function lade_moddatei(modname :
440:   string;
441:   ispeed, iloop : integer) : integer;
442: { Lädt die unter modname angegebene
443:   Mod-Datei. Mittels ispeed und
444:   iloop können die Variablen "Speed"
445:   und "Sound-Schleifen" vorgegeben
446:   werden, sinnvoller
447:   ist jedoch die Angabe AUTO !
448: }
449: procedure ende_mod;
450: { Beendet das Abspielen einer Mod-Datei
451:   und entfernt sie aus dem Speicher
452: }
453: procedure periodisch_on;
454: { Schaltet das Abspielen einer MOD-
455:   Datei ein. Muß zum Starten
456:   aufgerufen werden
457: }
458: procedure periodisch_off;

```

```

471: {
472:   Hält das Abspielen einer MOD-Datei
473:   an. Die MOD-Datei bleibt im Speicher
474:   und kann über periodisch_on
475:   wieder gestartet werden
476: }
477: procedure write_sbConfig;
478: {
479:   Gibt die gefundene Konfiguration aus
480:   (Textmodus !). Alternativ: Direkter
481:   Zugriff auf die entsprechenden
482:   Variablen
483: }
484: function Lade_Soundeffekt(s :
485:   string;
486:   ef : effect_type) : integer;
487: { Lädt den in s übergebenen Soundeffekt
488:   von Disk in den Speicher
489: }
490: {
491:   procedure Starte_Soundeffekt(ef
492:     : effect_type; frequenz, Vol, styp :
493:       word);
494:   { Startet den geladenen Soundeffekt.
495:     Frequenz = Samplfreq. (z.B. 13000)
496:     Vol darf zwischen 0 und 64 liegen
497:     styp = PC oder AMIGA für signed und
498:     unsigned Sample Data
499:   }
500: }
501: procedure dispose_Soundeffekt(ef
502:   : effect_type);
503: {
504:   Gibt den belegten Speicher wieder
505:   frei
506: }
507: procedure wr_dsp_sb16(v : byte);
508: procedure calculate_music;
509: { Intern! Berechnet die Musik, wird
510:   vom Timer-Int. aufgerufen
511: }
512: {
513:   implementation
514:   const
515:     Speed3 : word = 58;
516:     { Zum modifizieren von Speed }
517:     Loop3 : word = 42;
518:     Var
519:       SaveExitProc : pointer;
520:       { Nötig, da eigene Exitproc }
521:       music_played : boolean;
522:       { Flag, TRUE wenn Musik gespielt
523:         wurde }
524:       tonhoehe : word;
525:       { Der Wert der Tonhöhe, wie er in der
526:         MOD-Datei steht }
527:       ziel : pt;
528:       { Abspielpuffer im pt-Format }
529:       Modp : pointer;
530:       { Pointer auf Rm_Song }
531:       altx, alty : integer;
532:       lax, lbx, lcx, ldx,
533:       lsi, ldi, les, lds : word;
534:       yax, ybx, ycx, ydx,
535:       ysi, ydi, yes, yds : word;
536:       gax, gbx, gcx, gdx,
537:       gsi, gdi, ges, gds : word;
538:       { Hilfsvariablen zur Sicherung der
539:         Register }
540:       procedure wr_dsp_sb16(v : byte);
541:       {
542:         Wartet, bis der DSP zum Schreiben
543:         bereit ist, und schreibt dann das
544:         in "v" übergebene Byte in den DSP
545:       }
546:       begin;
547:         while port[dsp_adr+$c] >= 128 do ;
548:           port[dsp_adr+$c] := v;
549:         end;
550:       FUNCTION SbReadByte : BYTE;
551:       {
552:         Die Funktion wartet, bis der DSP
553:         gelesen werden kann und liefert den
554:         gelesenen Wert zurück
555:       }
556:       begin;
557:         while port[dsp_adr+$a] = $AA do ;
558:           { warten, bis DSP ready }
559:           SbReadByte := port[dsp_adr+$a];
560:           { Wert schreiben }
561:         end;
562:       FUNCTION Reset_sb16 : BOOLEAN;
563:       {
564:         Die Funktion resetet den DSP. War
565:         das Resetten erfolgreich, wird TRUE
566:         zurückgeliefert, ansonsten FALSE
567:       }
568:       const ready = $AA;
569:       VAR ct, stat : BYTE;
570:       BEGIN
571:         PORT[dsp_adr+$6] := 1;
572:         { dsp_adr+$6 = Resetfunktion }
573:         FOR ct := 1 TO 100 DO
574:           PORT[dsp_adr+$6] := 0;
575:           stat := 0;
576:           ct := 0;
577:           WHILE (stat <> ready) DO BEGIN
578:             AND (ct < 100) DO BEGIN
579:               stat := PORT[dsp_adr+$E];
580:               stat := PORT[dsp_adr+$a];
581:               INC(ct);
582:             END;
583:           { Der Vergleich ct < 100, da die
584:             Initialisierung ca.100ms dauert }
585:           Reset_sb16 := (stat = ready);
586:         END;
587:       FUNCTION Detect_Reg_sb16 : BOOLEAN;
588:       {
589:         Die Funktion liefert TRUE zurück,

```

```

591: wenn ein Soundblaster initialisiert
592: werden konnte, ansonsten FALSE.
593: Die Variable dsp_adr wird auf die
594: Base-Adresse des SB gesetzt.
595: }
596: VAR
597:   Port, Lst : WORD;
598: BEGIN
599:   Detect_Reg_sb16 := SbRegDetected;
600:   IF SbRegDetected THEN EXIT;
601:   { Exit, wenn initialisiert }
602:   Port := $210;
603:   Lst := $280;
604:   { Mögliche SB-Adressen zwischen $210
605:     und $280 ! }
606:   WHILE (NOT SbRegDetected)
607:     AND (Port <= Lst) DO BEGIN
608:     dsp_adr := Port;
609:     SbRegDetected := Reset_sb16;
610:     IF NOT SbRegDetected THEN
611:       INC(Port, $10);
612:   END;
613:   Detect_Reg_sb16 := SbRegDetected;
614: END;
615: PROCEDURE Write_MixerReg_sb16(Reg,
616:   Val : BYTE);
617: {
618:   Schreibt den in Val übergebenen
619:   Wert an das in Reg angegebene
620:   Register des Mixer - Chips
621: }
622: begin;
623:   Port[dsp_adr+$4] := Reg;
624:   Port[dsp_adr+$5] := Val;
625: END;
626: FUNCTION Read_MixerReg_sb16(Reg :
627:   BYTE) :
628:   BYTE;
629: {
630:   Die Function liefert den Inhalt des
631:   über Reg indizierten Registers des
632:   Mixer-Chips
633: }
634: begin;
635:   Port[dsp_adr+$4] := Reg;
636:   Read_MixerReg_sb16 :=
637:     Port[dsp_adr+$5];
638: end;
639: procedure reset_Mixer; assembler;
640: {
641:   Resetet den Mixer Chip auf seine
642:   Default - Werte
643: }
644: asm
645:   mov dx, dsp_adr+$4
646:   mov al, 0
647:   out dx, al
648:   mov cx, 50
649: @loop:
650:   loop @loop
651:   inc dx
652:   inc al
653:   out dx, al
654: end;
655: FUNCTION Detect_Mixer_sb16 : BOOLEAN;
656: {
657:   Function zu Erkennung des Mixer-
658:   Chips. TRUE, wenn der Mixer gefunden
659:   wurde, ansonsten FALSE
660: }
661: VAR SaveReg : WORD;
662:   NewReg : WORD;
663: BEGIN
664:   Detect_Mixer_sb16 := MixerDetected;
665:   IF (NOT SbRegDetected)
666:     OR MixerDetected THEN EXIT;
667:   { Abbruch, wenn keine Soundblaster-
668:     Karte vorhanden oder Mixer-Chip
669:     schon initialisiert }
670:   Reset_Mixer;
671:   SaveReg := Read_MixerReg_sb16($22);
672:   { Register sichern }
673:   Write_MixerReg_sb16($22, 243);
674:   NewReg := Read_MixerReg_sb16($22);
675:   { Wenn der geschriebene wert mit dem
676:     zurückgelesenen übereinstimmt, so
677:     ist ein Zugriff möglich und somit
678:     ein Mixer vorhanden }
679:   IF NewReg = 243 THEN begin;
680:     MixerDetected := TRUE;
681:     STEREO := True;
682:   end;
683:   Write_MixerReg_sb16($22, SaveReg);
684:   { Altes Register zurück }
685:   Detect_Mixer_sb16 := MixerDetected;
686: END;
687: PROCEDURE SbGetDSPVersion;
688: VAR i : WORD;
689:   t : WORD;
690:   s : STRING[2];
691: BEGIN
692:   wr_dsp_sb16($E1);
693:   { $E1 = Versionsabfrage }
694:   SbVersMaj := SbReadByte;
695:   SbVersMin := SbReadByte;
696:   str(SbVersMaj, SbVersStr);
697:   SbVersStr := SbVersStr + '.';
698:   str(SbVersMin, s);
699:   if SbVersMin > 9 then
700:     SbVersStr := SbVersStr + s
701:   else
702:     SbVersStr := SbVersStr + '0' + s;

```

Listing 1. Die Unit »modplay.pas« gestattet die Ausgabe von MOD-Dateien während des Ablaufs von Applikationen.


```

703: END;
704: procedure dsp_int_sb16; interrupt;
705: {
706:   Diese Procedure wird bei verbogenem
707:   SB-Interrupt nach Beendigung einer
708:   Sound- Ausgabe via DMA angesprungen.
709:   Die Prozedur sollte aber NICHT
710:   angesprungen werden, wenn der
711:   Computer schnell genug arbeitet, da
712:   wir ja gerade dies über die
713:   Timer-Programmierung abfangen. Desha
  lb
714:   erfolgt die Ausgabe einer Warnung.
715: }
716: var h : byte;
717: begin;
718:   h := port[dsp_adr+$E];
719:   dsp_rdy_sb16 := true;
720:   Port[$20] := $20;
721:   Choose_lower_freq := true;
722: end;
723: function wrt_dsp_adr_sb16 : string;
724: {
725:   Liefert die Base-Adresse des SB als
726:   String zurück
727: }
728: begin;
729:   case dsp_adr of
730:     $210 : wrt_dsp_adr_sb16 := '210';
731:     $220 : wrt_dsp_adr_sb16 := '220';
732:     $230 : wrt_dsp_adr_sb16 := '230';
733:     $240 : wrt_dsp_adr_sb16 := '240';
734:     $250 : wrt_dsp_adr_sb16 := '250';
735:     $260 : wrt_dsp_adr_sb16 := '260';
736:     $270 : wrt_dsp_adr_sb16 := '270';
737:     $280 : wrt_dsp_adr_sb16 := '280';
738:   END;
739: end;
740: function wrt_dsp_irq : string;
741: {
742:   Liefert den IRQ des SB als String
743:   zurück
744: }
745: begin;
746:   case dsp_irq of
747:     $2 : wrt_dsp_irq := '2 h';
748:     $3 : wrt_dsp_irq := '3 h';
749:     $5 : wrt_dsp_irq := '5 h';
750:     $7 : wrt_dsp_irq := '7 h';
751:     $10 : wrt_dsp_irq := '10 h';
752:   END;
753: end;
754: procedure Set_Timeconst_sb16(tc :
755:   byte);
756: {
757:   Procedure zum Setzen der Time-
758:   Konstanten. Sie berechnet sich nach
759:   der Formel:
760:   tc := 256 - (1000000 / Frequenz).
761: }
762: begin;
763:   wr_dsp_sb16($40);
764:   { $40 = Setze Sample Rate }
765:   wr_dsp_sb16(tc);
766: end;
767: procedure testIRQ; interrupt;
768: {
769:   Diese Routine wird zur Erkennung
770:   des IRQ benötigt
771: }
772: begin;
773:   IRQDetected := TRUE;
774: end;
775: procedure detect_sbIRQ;
776: {
777:   Diese Routine erkennt den IRQ der
778:   Soundblaster-Karte. Es werden dazu
779:   alle möglichen Interrupts durchge-
780:   testet. Dazu werden kurze Blöcke
781:   via DMA ausgegeben. Wenn am Ende
782:   der Ausgabe der eingestellte
783:   Interrupt angesprungen wird, so ist
784:   der richtige gefunden.
785: }
786: const moegliche_irqs : array[1..5]
787:   of byte
788:   = ($2,$3,$5,$7,$10);
789: var i : integer;
790: begin;
791:   getmem(blk,12);
792:   set_timeconst_sb16(211);
793:   wr_dsp_sb16($D3);
794:   { Lautsprecher aus }
795:   i := 1;
796:   while (i <= 5)
797:   and (not IRQDetected) do
798:   begin;
799:     dsp_irq := moegliche_irqs[i];
800:     { zu Testender IRQ }
801:     getintvec($8+dsp_irq,oldint);
802:     { Interrupt Verbiegen }
803:     setintvec($8+dsp_irq,@testIRQ);
804:     irqmsk := 1 shl dsp_irq;
805:     port[$21] := port[$21] and
806:       not irqmsk;
807:     blockgroesse := 12;
808:     dsp_block_sb16;
809:     { testweise Ausgabe }
810:     delay(30);
811:     setintvec($8+dsp_irq,oldint);
812:     { Interrupt wieder zurück }
813:     port[$21] := port[$21]
814:       or irqmsk;
815:     port[dsp_adr+$C] := $D3;
816:     Port[$20] := $20;
817:     inc(i);
818:   end;
819:   wr_dsp_sb16($D1);
820:   { Lautsprecher wieder ein }
821:   freemem(blk,12);

```

```

822: end;
823: function Init_Sb16 : boolean;
824: {
825:   Diese Function initialisiert den
826:   Soundblaster. Sie liefert TRUE
827:   zurück, wenn die Initialisierung
828:   erfolgreich war, ansonsten FALSE.
829:   Der Lautsprecher für Sampling-
830:   Ausgabe wird eingeschaltet. Der DMA-
831:   Ready Interrupt wird auf eine eigene
832:   Routine verbogen.
833: }
834: begin;
835:   if not detect_Reg_sb16 then begin;
836:     Init_Sb16 := false;
837:     exit;
838:   end else Init_Sb16 := true;
839:   { Soundblaster gefunden }
840:   detect_sbIRQ;
841:   { IRQ auto-detection }
842:   if Detect_Mixer_sb16 then begin;
843:     SbProDetected := TRUE;
844:     { SB Pro gefunden }
845:   end;
846:   SbGetDspVersion;
847:   if SbVersMaj > 4 then
848:   { SB 16 ASP gefunden }
849:   wr_dsp_sb16($D1);
850:   { Lautsprecher ein }
851:   getintvec($8+dsp_irq,oldint);
852:   { Alten Interrupt sichern, }
853:   setintvec($8+dsp_irq,@dsp_int_sb16);
854:   { auf eigene Routine setzen }
855:   irqmsk := 1 shl dsp_irq;
856:   { Interrupt einmaskieren }
857:   port[$21] := port[$21]
858:     and not irqmsk;
859:   end;
860:   procedure write_sbConfig;
861:   {
862:     Die Procedure gibt die gefundene
863:     Konfiguration auf dem Bildschirm
864:     aus. Sie dient vornehmlich als
865:     Beispiel, wie die Informationen
866:     verwendet werden können.
867:   }
868:   begin;
869:     if SbRegDetected then begin;
870:       writeln('Soundkarte an Base ',
871:         wrt_dsp_adr_sb16,'h mit IRQ ',
872:         wrt_dsp_irq,' gefunden. ');
873:     end else begin;
874:       writeln('Keine Soundblaster-',
875:         'kompatible Karte gefunden !');
876:     end;
877:     if MixerDetected then begin;
878:       writeln('Mixer - Chip gefunden');
879:       if SbVersMaj < 4 then
880:         writeln('Die gefundene Karte ist',
881:           ' ein Soundblaster Pro oder ',
882:           'kompatibel');
883:       else
884:         writeln('Die gefundene Karte ist',
885:           ' ein Soundblaster 16 ASP oder ',
886:           'kompatibel');
887:     end else begin;
888:       writeln('Die gefundene Karte ist',
889:         ' ein Soundblaster oder kompatibel');
890:     end;
891:     writeln('Die Versionsnummer lautet ',
892:       SbVersStr);
893:   end;
894:   procedure Exit_Sb16;
895:   {
896:     Diese Prozedur wird beim Beenden des
897:     Programms aufgerufen und setzt den
898:     verbogenen DMA-Interrupt auf seinen
899:     Ausgangswert
900:   }
901:   begin;
902:     setintvec($8+dsp_irq,oldint);
903:     port[$21] := port[$21] or irqmsk;
904:     { Alten Interrupt wieder herstellen
905:       und Maskierung auf alten Wert
906:       zurück }
907:     port[dsp_adr+$C] := $D3;
908:     { Lautsprecher aus }
909:     Port[$20] := $20;
910:   end;
911:   procedure Spiele_Sb16(Segm,Offs
912:     ,dsiz : word);
913:   {
914:     Diese Procedure spielt den über
915:     Segm:Offs adressierten Block mit der
916:     Größe dsiz ab. Es ist darauf zu
917:     achten, daß der DMA-Controller
918:     NICHT seitenübergreifend arbeiten
919:     kann ...
920:   }
921:   begin;
922:     port[$0A] := dma_ch+4;
923:     { DMA-Kanal sperren }
924:     Port[$0B] := $49;
925:     { für Soundausgabe }
926:     Port[$0C] := 0;
927:     Port[$02] := Lo(offs);
928:     Port[$02] := Hi(offs);
929:     { Adresse des Puffers (blk) an
930:       DMA-Controller }
931:     Port[$83] := Segm;
932:     Port[$03] := Lo(dsiz-1);
933:     Port[$03] := Hi(dsiz-1);
934:     { Größe des Blockes (blockgroesse)
935:       an DMA-Controller }
936:     Port[$0A] := dma_ch;
937:     { DMA-Kanal freigeben }
938:     wr_dsp_sb16($14);
939:     { DSP-Befehl 8-Bit über DMA }
940:     wr_dsp_sb16($D0);

```

```

942:   wr_dsp_sb16($14);
943:   wr_dsp_sb16(Lo(dsiz-1));
944:   wr_dsp_sb16(Hi(dsiz-1));
945:   { Größe des Blockes an den DSP }
946: end;
947: procedure dsp_block_sb16;
948: {
949:   Diese Procedure startet die Ausgabe
950:   des Daten-Blockes blk mit der Größe
951:   blockgroesse über DMA
952: }
953: var l : longint;
954:   pn,ofs : word;
955:   hbyte : byte;
956:   a : word;
957:   OldV,NewV,Hilfe : byte;
958: begin;
959:   dsp_rdy_sb16 := false;
960:   l := 16*longint(pt(blk).sgm)+
961:     pt(blk).ofs;
962:   pn := pt(l).sgm;
963:   ofs := pt(l).ofs;
964:   Spiele_Sb16(pn,ofs,blockgroesse);
965: end;
966: procedure memmove(q,d : pt;anzahl
967:   : word); assembler;
968: {
969:   Procedure zum schnellen kopieren von
970:   zwei Speicherbereichen. Über-
971:   lappungen werden nicht berück-
972:   sichtigt
973: }
974: asm
975:   push ds
976:   mov si,q.sgm
977:   mov ds,si
978:   mov si,q.ofs
979:   mov di,d.sgm
980:   mov es,di
981:   mov di,d.ofs
982:   mov cx,anzahl
983:   rep movsw
984:   pop ds
985: end;
986: procedure sreg; assembler;
987: {
988:   Sichert die Register in die
989:   Variablen lax bis ldi
990: }
991: asm
992:   mov lax,ax
993:   mov lbx,bx
994:   mov lcx,cx
995:   mov ldx,dx
996:   mov lsi,si
997:   mov ldi,di
998:   mov ax,es
999:   mov les,ax
1000:   mov ax,ds
1001:   mov lds,ax
1002: end;
1003: procedure rreg; assembler;
1004: {
1005:   Stellt die Register mit den Werten
1006:   aus lax bis ldi wieder her
1007: }
1008: asm
1009:   mov bx,lbx
1010:   mov cx,lcx
1011:   mov dx,ldx
1012:   mov si,lsi
1013:   mov di,ldi
1014:   mov ax,les
1015:   mov es,ax
1016:   mov ax,lds
1017:   mov ds,ax
1018:   mov ax,lax
1019: end;
1020: procedure syreg; assembler;
1021: {
1022:   Sichert die Register in die
1023:   Variablen yax bis ydi
1024: }
1025: asm
1026:   mov yax,ax
1027:   mov ybx,bx
1028:   mov ycx,cx
1029:   mov ydx,dx
1030:   mov ysi,si
1031:   mov ydi,di
1032:   mov ax,es
1033:   mov yes,ax
1034:   mov ax,ds
1035:   mov yds,ax
1036: end;
1037: procedure ryreg; assembler;
1038: {
1039:   Stellt die Register mit den Werten
1040:   aus yax bis ydi wieder her
1041: }
1042: asm
1043:   mov bx,ybx
1044:   mov cx,ycx
1045:   mov dx,ydx
1046:   mov si,ydi
1047:   mov di,ydi
1048:   mov ax,yes
1049:   mov es,ax
1050:   mov ax,yds
1051:   mov ds,ax
1052:   mov ax,yax
1053: end;

```

Listing 1. Die Unit »modplay.pas« gestattet die Ausgabe von MOD-Dateien während des Ablaufs von Applikationen.


```

1054: procedure get_ptune(hoehe
1055:   : word;Var vk
1056:   : word;VAR nk,dif : byte);
1057: {
1058:   Die Procedure ermittelt aus der
1059:   übergebenen Tonhöhe (so wie sie in
1060:   der MOD-Datei steht) die für die
1061:   Frequenz-Manipulation benötigten
1062:   Vor- und Nachkommastellen.
1063: }
1064: var nct : byte;
1065:   gefunden : boolean;
1066: begin;
1067:   nct := 1;
1068:   gefunden := false;
1069:   while (nct < 60) and not gefunden do
1070:   { Bis gefunden oder letzter Wert in
1071:   Tabelle }
1072:   begin;
1073:     if hoehe > Modoktave[nct] then
1074:       gefunden := true;
1075:     inc(nct);
1076:   end;
1077:   if gefunden then begin;
1078:     dif := ndiff[nct-tpw+12];
1079:     vk := vkt[nct-tpw+12];
1080:     nk := nkt[nct-tpw+12];
1081:   { Werte aus Tabelle holen. Tuning
1082:   über die tpw }
1083:   end;
1084: end;
1085: procedure innen_schleife_4;
1086:   assembler;
1087: {
1088:   Hier erfolgt die eigentliche
1089:   Vermischung der Daten. Der Puffer
1090:   blk wird dabei mit
1091:   den berechneten Daten gefüllt.
1092: }
1093: Var Output : word;
1094: asm
1095:   mov cx,Sound_Schleifen
1096: @Die_Schleife:
1097:   push cx
1098:   xor ax,ax
1099: { Register zurücksetzen }
1100:   xor dx,dx
1101:   mov Output,ax
1102:   mov bx,in11
1103:   sub bx,in1p
1104:   cmp bx,10
1105:   ja @Nochn_label1
1106: { Ton holen, da alles normal }
1107:   cmp 111,10
1108:   jae @Ton_St1_loop
1109: { Wenn Loopen, dann Loopproc und
1110:   weiter }
1111:   mov Vkl,0
1112:   mov Nkl,0
1113: { Ende erreicht, stehenbleiben }
1114: @Nochn_label1:
1115:   mov bx,i1.sgm
1116: { Instr. 1 laden }
1117:   mov es,bx
1118:   mov bx,Inst1vk
1119:   mov al,es:[bx]
1120:   sub al,128
1121:   mul Notvol1
1122: { Lautstärke berechnen }
1123:   shr ax,6
1124:   jmp @Weiter_Stimme_1
1125: @Ton_St1_loop:
1126:   mov ax,ls1
1127: { Auf Loopstart setzen }
1128:   add ax,5
1129: { + 5 wgn. Knacken }
1130:   mov in1p,ax
1131:   mov Inst1vk,ax
1132:   jmp @Nochn_label1
1133: @Weiter_Stimme_1:
1134:   mov bx,Vkl
1135: { Zeiger weitersetzen }
1136:   add Inst1vk,bx
1137:   add in1p,bx
1138:   mov bl,Inst1nk
1139:   mov dl,Nkl
1140: { Nachkomma Behandlung }
1141:   add bl,dl
1142: @CheckPermUp_1:
1143:   cmp PermUp_1,0
1144:   je @CheckPermDo_1
1145:   inc nk1
1146:   dec Dif1
1147: { Dec Dif }
1148:   jnz @ueberlauf_1
1149:   mov PermUp_1,0
1150: { Wenn Dif = 0 => PerUp = 0 }
1151:   jmp @ueberlauf_1
1152: @CheckPermDo_1:
1153:   cmp PermDo_1,0
1154:   je @ueberlauf_1
1155:   dec nk1
1156:   dec Dif1
1157: { Dec Dif }
1158:   jnz @ueberlauf_1
1159:   mov PermDo_1,0
1160: { Wenn Dif = 0 => PerDo = 0 }
1161:   jmp @ueberlauf_1
1162: { Nachkomma - Korrektur }
1163:   cmp bl,100
1164:   jna @Inst1_weiter
1165:   sub bl,100
1166:   inc Inst1vk
1167:   inc in1p
1168:   jmp @ueberlauf_1
1169: @Inst1_weiter:
1170:   mov inst1nk,bl
1171: @Stimme_2:
1172:   mov Output,ax
1173:   mov bx,in21

```

```

1174:   sub bx,in2p
1175:   cmp bx,10
1176:   ja @Nochn_label2
1177:   cmp 112,10
1178:   jae @Ton_St2_loop
1179:   mov VK2,0
1180:   mov NK2,0
1181: @Nochn_label2:
1182:   mov bx,i2.sgm
1183:   mov es,bx
1184:   mov bx,Inst2vk
1185:   mov al,es:[bx]
1186:   sub al,128
1187:   mul Notvol2
1188:   shr ax,6
1189:   add Output,ax
1190:   jmp @Weiter_Stimme_2
1191: @Ton_St2_loop:
1192:   mov bx,ls2
1193:   add bx,5
1194:   mov in2p,bx
1195:   mov Inst2vk,bx
1196:   jmp @Nochn_label2
1197: @Weiter_Stimme_2:
1198:   mov bx,VK2
1199:   add Inst2vk,bx
1200:   add in2p,bx
1201:   mov bl,Inst2nk
1202:   mov dl,NK2
1203:   add bl,dl
1204: @CheckPermUp_2:
1205:   cmp PermUp_2,0
1206:   je @CheckPermDo_2
1207:   inc nk2
1208:   dec Dif2
1209:   jnz @ueberlauf_2
1210:   mov PermUp_2,0
1211:   jmp @ueberlauf_2
1212: @CheckPermDo_2:
1213:   cmp PermDo_2,0
1214:   je @ueberlauf_2
1215:   dec nk2
1216:   dec Dif2
1217:   jnz @ueberlauf_2
1218:   mov PermDo_2,0
1219: @ueberlauf_2:
1220:   cmp bl,100
1221:   jna @Inst2_weiter
1222:   sub bl,100
1223:   inc Inst2vk
1224:   inc in2p
1225:   jmp @ueberlauf_2
1226: @Inst2_weiter:
1227:   mov Inst2nk,bl
1228: @Stimme_3:
1229:   mov bx,in31
1230:   sub bx,in3p
1231:   cmp bx,10
1232:   ja @Nochn_label3
1233:   cmp 113,10
1234:   jae @Ton_St3_loop
1235:   mov VK3,0
1236:   mov NK3,0
1237: @Nochn_label3:
1238:   mov bx,i3.sgm
1239:   mov es,bx
1240:   mov bx,Inst3vk
1241:   mov al,es:[bx]
1242:   sub al,128
1243:   mul Notvol3
1244:   shr ax,6
1245:   add Output,ax
1246:   jmp @Weiter_Stimme_3
1247: @Ton_St3_loop:
1248:   mov bx,ls3
1249:   add bx,5
1250:   mov in3p,bx
1251:   mov Inst3vk,bx
1252:   jmp @Nochn_label3
1253: @Weiter_Stimme_3:
1254:   mov bx,VK3
1255:   add Inst3vk,bx
1256:   add in3p,bx
1257:   mov bl,Inst3nk
1258:   mov dl,NK3
1259:   add bl,dl
1260: @CheckPermUp_3:
1261:   cmp PermUp_3,0
1262:   je @CheckPermDo_3
1263:   inc nk3
1264:   dec Dif3
1265:   jnz @ueberlauf_3
1266:   mov PermUp_3,0
1267:   jmp @ueberlauf_3
1268: @CheckPermDo_3:
1269:   cmp PermDo_3,0
1270:   je @ueberlauf_3
1271:   dec nk3
1272:   dec Dif3
1273:   jnz @ueberlauf_3
1274:   mov PermDo_3,0
1275: @ueberlauf_3:
1276:   cmp bl,100
1277:   jna @Inst3_weiter
1278:   sub bl,100
1279:   inc Inst3vk
1280:   inc in3p
1281:   jmp @ueberlauf_3
1282: @Inst3_weiter:
1283:   mov Inst3nk,bl
1284: @Stimme_4:
1285:   mov bx,in41
1286:   sub bx,in4p
1287:   cmp bx,10
1288:   ja @Nochn_label4
1289:   cmp 114,10
1290:   jae @Ton_St4_loop
1291:   mov VK4,0
1292:   mov NK4,0
1293: @Nochn_label4:

```

```

1294:   mov bx,i4.sgm
1295:   mov es,bx
1296:   mov bx,Inst4vk
1297:   mov al,es:[bx]
1298:   sub al,128
1299:   mul Notvol4
1300:   shr ax,6
1301:   add Output,ax
1302:   jmp @Weiter_Stimme_4
1303: @Ton_St4_loop:
1304:   mov bx,ls4
1305:   add bx,5
1306:   mov in4p,bx
1307:   mov Inst4vk,bx
1308:   jmp @Nochn_label4
1309: @Weiter_Stimme_4:
1310:   mov bx,VK4
1311:   add Inst4vk,bx
1312:   add in4p,bx
1313:   mov bl,Inst4nk
1314:   mov dl,NK4
1315:   add bl,dl
1316: @CheckPermUp_4:
1317:   cmp PermUp_4,0
1318:   je @CheckPermDo_4
1319:   inc nk4
1320:   dec Dif4
1321:   jnz @ueberlauf_4
1322:   mov PermUp_4,0
1323:   jmp @ueberlauf_4
1324: @CheckPermDo_4:
1325:   cmp PermDo_4,0
1326:   je @ueberlauf_4
1327:   dec nk4
1328:   dec Dif4
1329:   jnz @ueberlauf_4
1330:   mov PermDo_4,0
1331: @ueberlauf_4:
1332:   cmp bl,100
1333:   jna @Inst4_weiter
1334:   sub bl,100
1335:   inc Inst4vk
1336:   inc in4p
1337:   jmp @ueberlauf_4
1338: @Inst4_weiter:
1339:   mov Inst4nk,bl
1340: @Weiter_Ziel:
1341:   cmp playeffect,1
1342: { Effekt abspielen ? }
1343:   jne @effekt_ende
1344:   mov bx,Effekt.sgm
1345:   mov es,bx
1346:   mov bx,Effistvk
1347:   mov ah,0
1348:   mov al,es:[bx]
1349:   cmp convertff,1
1350:   jne @PC_Format
1351: { Für PC/AMIGA Format }
1352:   sub al,128
1353: @PC_Format:
1354:   mul effectvolume
1355:   shr ax,6
1356: { Lautstärke berechnen }
1357:   add Output,ax
1358:   add Output,ax
1359:   add Output,ax
1360:   add Output,ax
1361:   shr Output,3
1362:   mov bx,effektgroesse
1363:   sub bx,effektposi
1364:   cmp bx,5
1365:   ja @Sound_moves
1366:   jmp @Sound_raus
1367: @Sound_moves:
1368:   mov bx,Effvk
1369:   add Effistvk,bx
1370:   add Effektposi,bx
1371:   mov bl,Effistnk
1372:   mov dl,Effnk
1373:   add bl,dl
1374: @ueberlauf_E:
1375:   cmp bl,100
1376: { Nachkomma - Behandlung }
1377:   jna @Effekt_weiter
1378:   sub bl,100
1379:   inc Effistvk
1380:   inc Effektposi
1381:   jmp @ueberlauf_E
1382: @effekt_weiter:
1383:   mov Effistnk,bl
1384:   mov bx,effektgroesse
1385:   sub bx,effektposi
1386:   cmp bx,0
1387:   ja @Sound_raus
1388:   mov playeffect,0
1389:   jmp @Sound_raus
1390: @effekt_ende:
1391:   shr Output,3
1392: @Sound_raus:
1393:   mov bx,ziel.sgm
1394:   mov es,bx
1395:   mov bx,ziel ofs
1396:   mov ax,Output
1397:   mov es:[bx],al
1398: {Byte ins Ziel schreiben}
1399:   inc ziel ofs
1400:   pop cx
1401:   dec cx
1402:   cmp cx,0
1403:   ja @Die_Schleife
1404: end;
1405: procedure innen_schleife_8n;

```

Listing 1. Die Unit »modplay.pas« gestattet die Ausgabe von MOD-Dateien während des Ablaufs von Applikationen.


```

1406:      assembler;
1407: {
1408:   Hier erfolgt die eigentliche Ver-
1409:   mischung der Daten. Der Puffer blk
1410:   wird dabei mit den berechneten Daten
1411:   gefüllt.
1412: }
1413: Var Output : word;
1414: asm
1415:   mov cx,Sound_Schleifen
1416: @Die_Schleife:
1417:   push cx
1418:   xor ax,ax
1419: { Register zurücksetzen }
1420:   xor dx,dx
1421:   mov Output,ax
1422:   mov bx,in1l
1423:   sub bx,in1p
1424:   cmp bx,10
1425:   ja @Nochn_label1
1426: { Ton holen, da alles normal }
1427:   cmp 111,10
1428:   jae @Ton_St1_loop
1429: { Wenn Loopen, dann Loopproc und
1430:   weiter }
1431:   mov Vkl,0
1432:   mov Nkl,0
1433: { Ende erreicht, stehenbleiben }
1434: @Nochn_label1:
1435:   mov bx,i1.sgm
1436: { Instr. 1 laden }
1437:   mov es,bx
1438:   mov bx,Inst1vk
1439:   mov al,es:[bx]
1440:   sub al,128
1441:   mul Notvol1
1442: { Lautstärke berechnen }
1443:   shr ax,6
1444:   jmp @Weiter_Stimme_1
1445: @Ton_St1_loop:
1446:   mov ax,ls1
1447: { Auf Loopstart setzen }
1448:   add ax,5
1449: { + 5 wgn. Knacken }
1450:   mov in1p,ax
1451:   mov Inst1vk,ax
1452:   jmp @Nochn_label1
1453: @Weiter_Stimme_1:
1454:   mov bx,Vkl
1455: { Zeiger weitersetzen }
1456:   add Inst1vk,bx
1457:   add in1p,bx
1458:   mov bl,Inst1nk
1459:   mov dl,Nkl
1460: { Nachkomma Behandlung }
1461:   add bl,dl
1462: @CheckPermUp_1:
1463:   cmp PermUp_1,0
1464:   je @CheckPermDo_1
1465:   inc nk1
1466:   dec Dif1
1467: { Dec Dif }
1468:   jnz @ueberlauf_1
1469:   mov PermUp_1,0
1470: { Wenn Dif = 0 => PerUp = 0 }
1471:   jmp @ueberlauf_1
1472: @CheckPermDo_1:
1473:   cmp PermDo_1,0
1474:   je @ueberlauf_1
1475:   dec nk1
1476:   dec Dif1
1477: { Dec Dif }
1478:   jnz @ueberlauf_1
1479:   mov PermDo_1,0
1480: { Wenn Dif = 0 => PerDo = 0 }
1481: @ueberlauf_1:
1482: { Nachkomma - Korrektur }
1483:   cmp bl,100
1484:   jna @Inst1_weiter
1485:   sub bl,100
1486:   inc Inst1vk
1487:   inc in1p
1488:   jmp @ueberlauf_1
1489: @Inst1_weiter:
1490:   mov Inst1nk,bl
1491: @Stimme_2:
1492:   mov Output,ax
1493:   mov bx,in2l
1494:   sub bx,in2p
1495:   cmp bx,10
1496:   ja @Nochn_label2
1497:   cmp 112,10
1498:   jae @Ton_St2_loop
1499:   mov Vkl,0
1500:   mov Nkl,0
1501: @Nochn_label2:
1502:   mov bx,i2.sgm
1503:   mov es,bx
1504:   mov bx,Inst2vk
1505:   mov al,es:[bx]
1506:   sub al,128
1507:   mul Notvol2
1508:   shr ax,6
1509:   add Output,ax
1510:   jmp @Weiter_Stimme_2
1511: @Ton_St2_loop:
1512:   mov bx,ls2
1513:   add bx,5
1514:   mov in2p,bx
1515:   mov Inst2vk,bx
1516:   jmp @Nochn_label2
1517: @Weiter_Stimme_2:
1518:   mov bx,Vkl
1519:   add Inst2vk,bx
1520:   add in2p,bx
1521:   mov bl,Inst2nk
1522:   mov dl,Nkl
1523:   add bl,dl
1524: @CheckPermUp_2:
1525:   cmp PermUp_2,0

```

```

1526:   je @CheckPermDo_2
1527:   inc nk2
1528:   dec Dif2
1529:   jnz @ueberlauf_2
1530:   mov PermUp_2,0
1531:   jmp @ueberlauf_2
1532: @CheckPermDo_2:
1533:   cmp PermDo_2,0
1534:   je @ueberlauf_2
1535:   dec nk2
1536:   dec Dif2
1537:   jnz @ueberlauf_2
1538:   mov PermDo_2,0
1539: @ueberlauf_2:
1540:   cmp bl,100
1541:   jna @Inst2_weiter
1542:   sub bl,100
1543:   inc Inst2vk
1544:   inc in2p
1545:   jmp @ueberlauf_2
1546: @Inst2_weiter:
1547:   mov Inst2nk,bl
1548: @Stimme_3:
1549:   mov bx,in3l
1550:   sub bx,in3p
1551:   cmp bx,10
1552:   ja @Nochn_label3
1553:   cmp 113,10
1554:   jae @Ton_St3_loop
1555:   mov Vkl,0
1556:   mov Nkl,0
1557: @Nochn_label3:
1558:   mov bx,i3.sgm
1559:   mov es,bx
1560:   mov bx,Inst3vk
1561:   mov al,es:[bx]
1562:   sub al,128
1563:   mul Notvol3
1564:   shr ax,6
1565:   add Output,ax
1566:   jmp @Weiter_Stimme_3
1567: @Ton_St3_loop:
1568:   mov bx,ls3
1569:   add bx,5
1570:   mov in3p,bx
1571:   mov Inst3vk,bx
1572:   jmp @Nochn_label3
1573: @Weiter_Stimme_3:
1574:   mov bx,Vkl
1575:   add Inst3vk,bx
1576:   add in3p,bx
1577:   mov bl,Inst3nk
1578:   mov dl,Nkl
1579:   add bl,dl
1580: @CheckPermUp_3:
1581:   cmp PermUp_3,0
1582:   je @CheckPermDo_3
1583:   inc nk3
1584:   dec Dif3
1585:   jnz @ueberlauf_3
1586:   mov PermUp_3,0
1587:   jmp @ueberlauf_3
1588: @CheckPermDo_3:
1589:   cmp PermDo_3,0
1590:   je @ueberlauf_3
1591:   dec nk3
1592:   dec Dif3
1593:   jnz @ueberlauf_3
1594:   mov PermDo_3,0
1595: @ueberlauf_3:
1596:   cmp bl,100
1597:   jna @Inst3_weiter
1598:   sub bl,100
1599:   inc Inst3vk
1600:   inc in3p
1601:   jmp @ueberlauf_3
1602: @Inst3_weiter:
1603:   mov Inst3nk,bl
1604: @Stimme_4:
1605:   mov bx,in4l
1606:   sub bx,in4p
1607:   cmp bx,10
1608:   ja @Nochn_label4
1609:   cmp 114,10
1610:   jae @Ton_St4_loop
1611:   mov Vkl,0
1612:   mov Nkl,0
1613: @Nochn_label4:
1614:   mov bx,i4.sgm
1615:   mov es,bx
1616:   mov bx,Inst4vk
1617:   mov al,es:[bx]
1618:   sub al,128
1619:   mul Notvol4
1620:   shr ax,6
1621:   add Output,ax
1622:   jmp @Weiter_Stimme_4
1623: @Ton_St4_loop:
1624:   mov bx,ls4
1625:   add bx,5
1626:   mov in4p,bx
1627:   mov Inst4vk,bx
1628:   jmp @Nochn_label4
1629: @Weiter_Stimme_4:
1630:   mov bx,Vkl
1631:   add Inst4vk,bx
1632:   add in4p,bx
1633:   mov bl,Inst4nk
1634:   mov dl,Nkl
1635:   add bl,dl
1636: @CheckPermUp_4:
1637:   cmp PermUp_4,0
1638:   je @CheckPermDo_4
1639:   inc nk4
1640:   dec Dif4
1641:   jnz @ueberlauf_4
1642:   mov PermUp_4,0
1643:   jmp @ueberlauf_4
1644: @CheckPermDo_4:
1645:   cmp PermDo_4,0

```

```

1646:   je @ueberlauf_4
1647:   dec nk4
1648:   dec Dif4
1649:   jnz @ueberlauf_4
1650:   mov PermDo_4,0
1651: @ueberlauf_4:
1652:   cmp bl,100
1653:   jna @Inst4_weiter
1654:   sub bl,100
1655:   inc Inst4vk
1656:   inc in4p
1657:   jmp @ueberlauf_4
1658: @Inst4_weiter:
1659:   mov Inst4nk,bl
1660: @Stimme_5:
1661:   mov bx,in5l
1662:   sub bx,in5p
1663:   cmp bx,10
1664:   ja @Nochn_label5
1665:   cmp 115,10
1666:   jae @Ton_St5_loop
1667:   mov Vkl,0
1668:   mov Nkl,0
1669: @Nochn_label5:
1670:   mov bx,i5.sgm
1671:   mov es,bx
1672:   mov bx,Inst5vk
1673:   mov al,es:[bx]
1674:   sub al,128
1675:   mul Notvol5
1676:   shr ax,6
1677:   add Output,ax
1678:   jmp @Weiter_Stimme_5
1679: @Ton_St5_loop:
1680:   mov bx,ls5
1681:   add bx,5
1682:   mov in5p,bx
1683:   mov Inst5vk,bx
1684:   jmp @Nochn_label5
1685: @Weiter_Stimme_5:
1686:   mov bx,Vkl
1687:   add Inst5vk,bx
1688:   add in5p,bx
1689:   mov bl,Inst5nk
1690:   mov dl,Nkl
1691:   add bl,dl
1692: @CheckPermUp_5:
1693:   cmp PermUp_5,0
1694:   je @CheckPermDo_5
1695:   inc nk5
1696:   dec Dif5
1697:   jnz @ueberlauf_5
1698:   mov PermUp_5,0
1699:   jmp @ueberlauf_5
1700: @CheckPermDo_5:
1701:   cmp PermDo_5,0
1702:   je @ueberlauf_5
1703:   dec nk5
1704:   dec Dif5
1705:   jnz @ueberlauf_5
1706:   mov PermDo_5,0
1707: @ueberlauf_5:
1708:   cmp bl,100
1709:   jna @Inst5_weiter
1710:   sub bl,100
1711:   inc Inst5vk
1712:   inc in5p
1713:   jmp @ueberlauf_5
1714: @Inst5_weiter:
1715:   mov Inst5nk,bl
1716: @Stimme_6:
1717:   mov bx,in6l
1718:   sub bx,in6p
1719:   cmp bx,10
1720:   ja @Nochn_label6
1721:   cmp 116,10
1722:   jae @Ton_St6_loop
1723:   mov Vkl,0
1724:   mov Nkl,0
1725: @Nochn_label6:
1726:   mov bx,i6.sgm
1727:   mov es,bx
1728:   mov bx,Inst6vk
1729:   mov al,es:[bx]
1730:   sub al,128
1731:   mul Notvol6
1732:   shr ax,6
1733:   add Output,ax
1734:   jmp @Weiter_Stimme_6
1735: @Ton_St6_loop:
1736:   mov bx,ls6
1737:   add bx,5
1738:   mov in6p,bx
1739:   mov Inst6vk,bx
1740:   jmp @Nochn_label6
1741: @Weiter_Stimme_6:
1742:   mov bx,Vkl
1743:   add Inst6vk,bx
1744:   add in6p,bx
1745:   mov bl,Inst6nk
1746:   mov dl,Nkl
1747:   add bl,dl
1748: @CheckPermUp_6:
1749:   cmp PermUp_6,0
1750:   je @CheckPermDo_6
1751:   inc nk6
1752:   dec Dif6
1753:   jnz @ueberlauf_6
1754:   mov PermUp_6,0
1755:   jmp @ueberlauf_6
1756: @CheckPermDo_6:
1757:   cmp PermDo_6,0

```

Listing 1. Die Unit »modplay.pas« gestattet die Ausgabe von MOD-Dateien während des Ablaufs von Applikationen.


```

1758: je @ueberlauf_6
1759: dec nk6
1760: dec Dif6
1761: jnz @ueberlauf_6
1762: mov PermDo_6,0
1763: @ueberlauf_6:
1764: cmp bl,100
1765: jna @Inst6_weiter
1766: sub bl,100
1767: inc Inst6vk
1768: inc in6p
1769: jmp @ueberlauf_6
1770: @Inst6_weiter:
1771: mov Inst6nk,bl
1772: @Stimme_7:
1773: mov bx,in7l
1774: sub bx,in7p
1775: cmp bx,10
1776: ja @Nochn_label7
1777: cmp il7,10
1778: jae @Ton_St7_loop
1779: mov Vx7,0
1780: mov Nk7,0
1781: @Nochn_label7:
1782: mov bx,i7.sgm
1783: mov es,bx
1784: mov bx,Inst7vk
1785: mov al,es:[bx]
1786: sub al,128
1787: mul Notvol7
1788: shr ax,6
1789: add Output,ax
1790: jmp @Weiter_Stimme_7
1791: @Ton_St7_loop:
1792: mov bx,ls7
1793: add bx,5
1794: mov in7p,bx
1795: mov Inst7vk,bx
1796: jmp @Nochn_label7
1797: @Weiter_Stimme_7:
1798: mov bx,Vx7
1799: add Inst7vk,bx
1800: add in7p,bx
1801: mov bl,Inst7nk
1802: mov dl,Nk7
1803: add bl,dl
1804: @CheckPermUp_7:
1805: cmp PermUp_7,0
1806: je @CheckPermDo_7
1807: inc nk7
1808: dec Dif7
1809: jnz @ueberlauf_7
1810: mov PermUp_7,0
1811: jmp @ueberlauf_7
1812: @CheckPermDo_7:
1813: cmp PermDo_7,0
1814: je @ueberlauf_7
1815: dec nk7
1816: dec Dif7
1817: jnz @ueberlauf_7
1818: mov PermDo_7,0
1819: @ueberlauf_7:
1820: cmp bl,100
1821: jna @Inst7_weiter
1822: sub bl,100
1823: inc Inst7vk
1824: inc in7p
1825: jmp @ueberlauf_7
1826: @Inst7_weiter:
1827: mov Inst7nk,bl
1828: @Stimme_8:
1829: mov bx,in8l
1830: sub bx,in8p
1831: cmp bx,10
1832: ja @Nochn_label8
1833: cmp il8,10
1834: jae @Ton_St8_loop
1835: mov Vx8,0
1836: mov Nk8,0
1837: @Nochn_label8:
1838: mov bx,i8.sgm
1839: mov es,bx
1840: mov bx,Inst8vk
1841: mov al,es:[bx]
1842: sub al,128
1843: mul Notvol8
1844: shr ax,6
1845: add Output,ax
1846: jmp @Weiter_Stimme_8
1847: @Ton_St8_loop:
1848: mov bx,ls8
1849: add bx,5
1850: mov in8p,bx
1851: mov Inst8vk,bx
1852: jmp @Nochn_label8
1853: @Weiter_Stimme_8:
1854: mov bx,Vx8
1855: add Inst8vk,bx
1856: add in8p,bx
1857: mov bl,Inst8nk
1858: mov dl,Nk8
1859: add bl,dl
1860: @CheckPermUp_8:
1861: cmp PermUp_8,0
1862: je @CheckPermDo_8
1863: inc nk8
1864: dec Dif8
1865: jnz @ueberlauf_8
1866: mov PermUp_8,0
1867: jmp @ueberlauf_8
1868: @CheckPermDo_8:
1869: cmp PermDo_8,0
1870: je @ueberlauf_8
1871: dec nk8
1872: dec Dif8
1873: jnz @ueberlauf_8
1874: mov PermDo_8,0
1875: @ueberlauf_8:
1876: cmp bl,100
1877: jna @Inst8_weiter

```

```

1878: sub bl,100
1879: inc Inst8vk
1880: inc in8p
1881: jmp @ueberlauf_8
1882: @Inst8_weiter:
1883: mov Inst8nk,bl
1884: @Weiter_Ziel:
1885: cmp playeffect,1
1886: { Effekt abspielen ? }
1887: jne @effekt_ende
1888: mov bx,Effect.sgm
1889: mov es,bx
1890: mov bx,Effectvk
1891: mov ah,0
1892: mov al,es:[bx]
1893: cmp converteff,1
1894: jne @PC_Format
1895: { Für PC/AMIGA Format }
1896: sub al,128
1897: @PC_Format:
1898: mul effectvolume
1899: shr ax,6
1900: { Lautstärke berechnen }
1901: add Output,ax
1902: add Output,ax
1903: add Output,ax
1904: add Output,ax
1905: add Output,ax
1906: add Output,ax
1907: add Output,ax
1908: add Output,ax
1909: shr Output,4
1910: mov bx,effektgroesse
1911: sub bx,effektposi
1912: cmp bx,5
1913: ja @Sound_moves
1914: jmp @Sound_raus
1915: @Sound_moves:
1916: mov bx,Effectvk
1917: add Effectvk,bx
1918: add Effectposi,bx
1919: mov bl,Effectnk
1920: mov dl,Effectnk
1921: add bl,dl
1922: @ueberlauf_E:
1923: cmp bl,100
1924: { Nachkomma - Behandlung }
1925: jna @Effekt_weiter
1926: sub bl,100
1927: inc Effectvk
1928: inc Effectposi
1929: jmp @ueberlauf_E
1930: @effekt_weiter:
1931: mov Effectnk,bl
1932: mov bx,effektgroesse
1933: sub bx,effektposi
1934: cmp bx,0
1935: ja @Sound_raus
1936: mov playeffect,0
1937: jmp @Sound_raus
1938: @effekt_ende:
1939: shr Output,4
1940: @Sound_raus:
1941: mov bx,ziel.sgm
1942: mov es,bx
1943: mov bx,ziel ofs
1944: mov ax,Output
1945: mov es:[bx],al
1946: {Byte ins Ziel schreiben}
1947: inc ziel ofs
1948: pop cx
1949: dec cx
1950: cmp cx,0
1951: ja @Die_Schleife
1952: end;
1953:
1954: { $F+ }
1955: procedure vermische_start_4;
1956: {
1957:   Initialisiert benötigte Variablen
1958: }
1959: var rdifff : real;
1960: dummy : byte;
1961: begin;
1962: asm
1963:   mov ax,il ofs
1964:   add ax,inlp
1965:   mov Instlvk,ax
1966:   mov Instlnk,0
1967:   mov ax,il2 ofs
1968:   add ax,in2p
1969:   mov Inst2vk,ax
1970:   mov Inst2nk,0
1971:   mov ax,il3 ofs
1972:   add ax,in3p
1973:   mov Inst3vk,ax
1974:   mov Inst3nk,0
1975:   mov ax,il4 ofs
1976:   add ax,in4p
1977:   mov Inst4vk,ax
1978:   mov Inst4nk,0
1979: end;
1980: if notel < 0 then begin;
1981:   tonhoehe := (Rm_Song[mli,1,1]
1982:     and $0F)
1983:   *256+Rm_Song[mli,1,2];
1984:   get_pctune(tonhoehe,vk1,nk1,
1985:     difb1);
1986: end;
1987: dif1 := difb1 * Rm_Song[mli,1,4];
1988: rdifff := dif1 / 4;
1989: dif1 := trunc(rdifff);
1990: Pnk1 := Pnk1 + round((rdifff-dif1)*
1991:   100);
1992: if Pnk1 > 100 then begin;
1993:   inc(dif1);
1994:   dec(Pnk1,100);
1995: end;
1996: if note2 < 0 then begin;
1997:   tonhoehe := (Rm_Song[mli,2,1]

```

```

1998:     and $0F)
1999:   *256+Rm_Song[mli,2,2];
2000:   get_pctune(tonhoehe,vk2,nk2,
2001:     difb2);
2002: end;
2003: dif2 := difb2 * Rm_Song[mli,2,4];
2004: rdifff := dif2 / 4;
2005: dif2 := trunc(rdifff);
2006: Pnk2 := Pnk2 + round((rdifff-dif2)*
2007:   100);
2008: if Pnk2 > 100 then begin;
2009:   inc(dif2);
2010:   dec(Pnk2,100);
2011: end;
2012: if note3 < 0 then begin;
2013:   tonhoehe := (Rm_Song[mli,3,1]
2014:     and $0F)
2015:   *256+Rm_Song[mli,3,2];
2016:   get_pctune(tonhoehe,vk3,nk3,
2017:     difb3);
2018: end;
2019: dif3 := difb3 * Rm_Song[mli,3,4];
2020: rdifff := dif3 / 4;
2021: dif3 := trunc(rdifff);
2022: Pnk3 := Pnk3 + round((rdifff-dif3)*
2023:   100);
2024: if Pnk3 > 100 then begin;
2025:   inc(dif3);
2026:   dec(Pnk3,100);
2027: end;
2028: if note4 < 0 then begin;
2029:   tonhoehe := (Rm_Song[mli,4,1]
2030:     and $0F)
2031:   *256+Rm_Song[mli,4,2];
2032:   get_pctune(tonhoehe,vk4,nk4,
2033:     difb4);
2034: end;
2035: dif4 := difb4 * Rm_Song[mli,4,4];
2036: rdifff := dif4 / 4;
2037: dif4 := trunc(rdifff);
2038: Pnk4 := Pnk4 + round((rdifff-dif4)*
2039:   100);
2040: if Pnk4 > 100 then begin;
2041:   inc(dif4);
2042:   dec(Pnk4,100);
2043: end;
2044: ls1 := loop_s[In_st1];
2045: ll1 := loop_l[In_st1];
2046: ls2 := loop_s[In_st2];
2047: ll2 := loop_l[In_st2];
2048: ls3 := loop_s[In_st3];
2049: ll3 := loop_l[In_st3];
2050: ls4 := loop_s[In_st4];
2051: ll4 := loop_l[In_st4];
2052: if ll1 > 30 then inll := ll1+ls1;
2053: if ll2 > 30 then inl2 := ll2+ls2;
2054: if ll3 > 30 then inl3 := ll3+ls3;
2055: if ll4 > 30 then inl4 := ll4+ls4;
2056: end;
2057: { $F- }
2058:
2059: { $F+ }
2060: procedure vermische_start_8;
2061: {
2062:   Initialisiert benötigte Variablen
2063: }
2064: var rdifff : real;
2065: begin;
2066: asm
2067:   mov ax,il ofs
2068:   add ax,inlp
2069:   mov Instlvk,ax
2070:   mov Instlnk,0
2071:   mov ax,il2 ofs
2072:   add ax,in2p
2073:   mov Inst2vk,ax
2074:   mov Inst2nk,0
2075:   mov ax,il3 ofs
2076:   add ax,in3p
2077:   mov Inst3vk,ax
2078:   mov Inst3nk,0
2079:   mov ax,il4 ofs
2080:   add ax,in4p
2081:   mov Inst4vk,ax
2082:   mov Inst4nk,0
2083:   mov ax,il5 ofs
2084:   add ax,in5p
2085:   mov Inst5vk,ax
2086:   mov Inst5nk,0
2087:   mov ax,il6 ofs
2088:   add ax,in6p
2089:   mov Inst6vk,ax
2090:   mov Inst6nk,0
2091:   mov ax,il7 ofs
2092:   add ax,in7p
2093:   mov Inst7vk,ax
2094:   mov Inst7nk,0
2095:   mov ax,il8 ofs
2096:   add ax,in8p
2097:   mov Inst8vk,ax
2098:   mov Inst8nk,0
2099: end;
2100: if notel < 0 then begin;
2101:   tonhoehe := (Rm_Song[mli,1,1]
2102:     and $0F)
2103:   *256+Rm_Song[mli,1,2];
2104:   get_pctune(tonhoehe,vk1,nk1,
2105:     difb1);
2106: end;
2107: dif1 := difb1 * Rm_Song[mli,1,4];
2108: rdifff := dif1 / 4;
2109: dif1 := trunc(rdifff);

```

Listing 1. Die Unit »modplay.pas« gestattet die Ausgabe von MOD-Dateien während des Ablaufs von Applikationen.


```

2110: Pnk1 := Pnk1 + round((rdiff-dif1)*
2111:         100);
2112: if Pnk1 > 100 then begin;
2113:   inc(dif1);
2114:   dec(Pnk1,100);
2115: end;
2116: if note2 <> 0 then begin;
2117:   tonhoehe := (Rm_Song[mli,2,1]
2118:     and $0F)
2119:   *256+Rm_Song[mli,2,2];
2120:   get_pctune(tonhoehe,vk2,nk2,
2121:     difb2);
2122: end;
2123: dif2 := difb2 * Rm_Song[mli,2,4];
2124: rdiff := dif2 / 4;
2125: dif2 := trunc(rdiff);
2126: Pnk2 := Pnk2 + round((rdiff-dif2)*
2127:   100);
2128: if Pnk2 > 100 then begin;
2129:   inc(dif2);
2130:   dec(Pnk2,100);
2131: end;
2132: if note3 <> 0 then begin;
2133:   tonhoehe := (Rm_Song[mli,3,1]
2134:     and $0F)
2135:   *256+Rm_Song[mli,3,2];
2136:   get_pctune(tonhoehe,vk3,nk3,
2137:     difb3);
2138: end;
2139: dif3 := difb3 * Rm_Song[mli,3,4];
2140: rdiff := dif3 / 4;
2141: dif3 := trunc(rdiff);
2142: Pnk3 := Pnk3 + round((rdiff-dif3)*
2143:   100);
2144: if Pnk3 > 100 then begin;
2145:   inc(dif3);
2146:   dec(Pnk3,100);
2147: end;
2148: if note4 <> 0 then begin;
2149:   tonhoehe := (Rm_Song[mli,4,1]
2150:     and $0F)
2151:   *256+Rm_Song[mli,4,2];
2152:   get_pctune(tonhoehe,vk4,nk4,
2153:     difb4);
2154: end;
2155: dif4 := difb4 * Rm_Song[mli,4,4];
2156: rdiff := dif4 / 4;
2157: dif4 := trunc(rdiff);
2158: Pnk4 := Pnk4 + round((rdiff-dif4)*
2159:   100);
2160: if Pnk4 > 100 then begin;
2161:   inc(dif4);
2162:   dec(Pnk4,100);
2163: end;
2164: if note5 <> 0 then begin;
2165:   tonhoehe := (Rm_Song[mli,5,1]
2166:     and $0F)
2167:   *256+Rm_Song[mli,5,2];
2168:   get_pctune(tonhoehe,vk5,nk5,
2169:     difb5);
2170: end;
2171: dif5 := difb5 * Rm_Song[mli,5,4];
2172: rdiff := dif5 / 4;
2173: dif5 := trunc(rdiff);
2174: Pnk5 := Pnk5 + round((rdiff-dif5)*
2175:   100);
2176: if Pnk5 > 100 then begin;
2177:   inc(dif5);
2178:   dec(Pnk5,100);
2179: end;
2180: if note6 <> 0 then begin;
2181:   tonhoehe := (Rm_Song[mli,6,1]
2182:     and $0F)
2183:   *256+Rm_Song[mli,6,2];
2184:   get_pctune(tonhoehe,vk6,nk6,
2185:     difb6);
2186: end;
2187: dif6 := difb6 * Rm_Song[mli,6,4];
2188: rdiff := dif6 / 4;
2189: dif6 := trunc(rdiff);
2190: Pnk6 := Pnk6 + round((rdiff-dif6)*
2191:   100);
2192: if Pnk6 > 100 then begin;
2193:   inc(dif6);
2194:   dec(Pnk6,100);
2195: end;
2196: if note7 <> 0 then begin;
2197:   tonhoehe := (Rm_Song[mli,7,1]
2198:     and $0F)
2199:   *256+Rm_Song[mli,7,2];
2200:   get_pctune(tonhoehe,vk7,nk7,
2201:     difb7);
2202: end;
2203: dif7 := difb7 * Rm_Song[mli,7,4];
2204: rdiff := dif7 / 4;
2205: dif7 := trunc(rdiff);
2206: Pnk7 := Pnk7 + round((rdiff-dif7)*
2207:   100);
2208: if Pnk7 > 100 then begin;
2209:   inc(dif7);
2210:   dec(Pnk7,100);
2211: end;
2212: if note8 <> 0 then begin;
2213:   tonhoehe := (Rm_Song[mli,8,1]
2214:     and $0F)
2215:   *256+Rm_Song[mli,8,2];
2216:   get_pctune(tonhoehe,vk8,nk8,
2217:     difb8);
2218: end;
2219: dif8 := difb8 * Rm_Song[mli,8,4];
2220: rdiff := dif8 / 4;
2221: dif8 := trunc(rdiff);
2222: Pnk8 := Pnk8 + round((rdiff-dif8)*
2223:   100);
2224: if Pnk8 > 100 then begin;
2225:   inc(dif8);
2226:   dec(Pnk8,100);
2227: end;
2228: lsl := loop_s[In_stl];
2229: lll := loop_l[In_stl];

```

```

2230: ls2 := loop_s[In_st2];
2231: ll2 := loop_l[In_st2];
2232: ls3 := loop_s[In_st3];
2233: ll3 := loop_l[In_st3];
2234: ls4 := loop_s[In_st4];
2235: ll4 := loop_l[In_st4];
2236: ls5 := loop_s[In_st5];
2237: ll5 := loop_l[In_st5];
2238: ls6 := loop_s[In_st6];
2239: ll6 := loop_l[In_st6];
2240: ls7 := loop_s[In_st7];
2241: ll7 := loop_l[In_st7];
2242: ls8 := loop_s[In_st8];
2243: ll8 := loop_l[In_st8];
2244: if lll > 30 then inll := lll+ls1;
2245: if ll2 > 30 then inll := ll2+ls2;
2246: if ll3 > 30 then inll := ll3+ls3;
2247: if ll4 > 30 then inll := ll4+ls4;
2248: if ll5 > 30 then inll := ll5+ls5;
2249: if ll6 > 30 then inll := ll6+ls6;
2250: if ll7 > 30 then inll := ll7+ls7;
2251: if ll8 > 30 then inll := ll8+ls8;
2252: end;
2253: ($F)
2254: function Notenvolumen(Stm : byte) :
2255:   byte;
2256: begin;
2257:   Notenvolumen := Rm_Song[mli,Stm,4];
2258: end;
2259: procedure nmw_all_4;
2260: var idx : byte;
2261: begin;
2262:   inc(mli);
2263:   if mli > 64 then mli := 1;
2264:   if mli = 1 then begin;
2265:     inc(mlj);
2266:     if mlj > Liedlaenge then begin;
2267:       mlj := 1;
2268:       Modp := Addr(Rm_Song);
2269:       memmove(pt(rm[ljed[mlj]]),
2270:         pt(Modp),1024);
2271:       end else begin;
2272:         asm
2273:           call [periodisch_anhalten]
2274:         end;
2275:         music_aus := true;
2276:       end;
2277:     end else begin;
2278:       Modp := Addr(Rm_Song);
2279:       memmove(pt(rm[ljed[mlj]]),
2280:         pt(Modp),1024);
2281:     end;
2282:   end;
2283:   notel := (Rm_Song[mli,1,1] AND $F0)+
2284:     ((Rm_Song[mli,1,3] AND $F0) shr 4);
2285:   note2 := (Rm_Song[mli,2,1] AND $F0)+
2286:     ((Rm_Song[mli,2,3] AND $F0) shr 4);
2287:   note3 := (Rm_Song[mli,3,1] AND $F0)+
2288:     ((Rm_Song[mli,3,3] AND $F0) shr 4);
2289:   note4 := (Rm_Song[mli,4,1] AND $F0)+
2290:     ((Rm_Song[mli,4,3] AND $F0) shr 4);
2291:   if notel <> 0 then
2292:   begin;
2293:     Noten_Anschlag[1] := 500;
2294:     In_St1 := notel;
2295:     inst1 := Ptr(pt(Samp[In_St1]).sgm,
2296:       pt(Samp[In_St1]).ofs);
2297:     i1 := pt(inst1);
2298:     inll := Sam_l[In_St1];
2299:     inlp := 0;
2300:     notvoll := Sam_vol[In_St1];
2301:     Pnk1 := 0;
2302:   end;
2303:   if note2 <> 0 then
2304:   begin;
2305:     Noten_Anschlag[2] := 500;
2306:     In_St2 := note2;
2307:     inst2 := Ptr(pt(Samp[In_St2]).sgm,
2308:       pt(Samp[In_St2]).ofs);
2309:     i2 := pt(inst2);
2310:     in21 := Sam_l[In_St2];
2311:     in2p := 0;
2312:     notvol2 := Sam_vol[In_St2];
2313:     Pnk2 := 0;
2314:   end;
2315:   if note3 <> 0 then
2316:   begin;
2317:     Noten_Anschlag[3] := 500;
2318:     In_St3 := note3;
2319:     inst3 := Ptr(pt(Samp[In_St3]).sgm,
2320:       pt(Samp[In_St3]).ofs);
2321:     i3 := pt(inst3);
2322:     in31 := Sam_l[In_St3];
2323:     in3p := 0;
2324:     notvol3 := Sam_vol[In_St3];
2325:     Pnk3 := 0;
2326:   end;
2327:   if note4 <> 0 then
2328:   begin;
2329:     Noten_Anschlag[4] := 500;
2330:     In_St4 := note4;
2331:     inst4 := Ptr(pt(Samp[In_St4]).sgm,
2332:       pt(Samp[In_St4]).ofs);
2333:     i4 := pt(inst4);
2334:     in41 := Sam_l[In_St4];
2335:     in4p := 0;
2336:     notvol4 := Sam_vol[In_St4];
2337:     Pnk4 := 0;
2338:   end;
2339:   if Rm_Song[mli,1,3] and $0F <= 15
2340:   then begin;
2341:     case (Rm_Song[mli,1,3] and $0F) of
2342:       01 : begin;
2343:         PermUp_1 := 1;
2344:       end;
2345:       02 : begin;
2346:         PermDo_1 := 1;
2347:       end;
2348:       13 : begin;

```

```

2350:         mli := 64;
2351:         end;
2352:       11 : begin;
2353:         mli := 64;
2354:         mlj := Rm_Song[mli,1,4];
2355:         end;
2356:       12 : begin;
2357:         notvoll := Notenvolumen(1);
2358:         end;
2359:       15 : begin;
2360:         idx := Rm_Song[mli,1,4];
2361:         if idx < 32 then begin;
2362:           Aktspeed := idx;
2363:           modmultiply :=
2364:             ModPara[idx].mult;
2365:           Speed :=
2366:             ModPara[idx].Speed;
2367:           blockgroesse :=
2368:             ModPara[idx].bgr;
2369:           Dma_abbruch :=
2370:             ModPara[idx].Ab;
2371:         end;
2372:         end;
2373:     end;
2374:   end;
2375:   if Rm_Song[mli,2,3] and $0F <= 15
2376:   then begin;
2377:     case (Rm_Song[mli,2,3] and $0F) of
2378:       01 : begin;
2379:         PermUp_2 := 1;
2380:       end;
2381:       02 : begin;
2382:         PermDo_2 := 1;
2383:       end;
2384:       13 : begin;
2385:         mli := 64;
2386:         end;
2387:       11 : begin;
2388:         mli := 64;
2389:         mlj := Rm_Song[mli,2,4];
2390:         end;
2391:       12 : begin;
2392:         notvol2 := Notenvolumen(2);
2393:         end;
2394:       15 : begin;
2395:         idx := Rm_Song[mli,2,4];
2396:         if idx < 32 then begin;
2397:           Aktspeed := idx;
2398:           modmultiply :=
2399:             ModPara[idx].mult;
2400:           Speed :=
2401:             ModPara[idx].Speed;
2402:           blockgroesse :=
2403:             ModPara[idx].bgr;
2404:           Dma_abbruch :=
2405:             ModPara[idx].Ab;
2406:         end;
2407:         end;
2408:     end;
2409:   end;
2410:   if Rm_Song[mli,3,3] and $0F <= 15
2411:   then begin;
2412:     case (Rm_Song[mli,3,3] and $0F) of
2413:       01 : begin;
2414:         PermUp_3 := 1;
2415:       end;
2416:       02 : begin;
2417:         PermDo_3 := 1;
2418:       end;
2419:       13 : begin;
2420:         mli := 64;
2421:         end;
2422:       11 : begin;
2423:         mli := 64;
2424:         mlj := Rm_Song[mli,3,4];
2425:         end;
2426:       12 : begin;
2427:         notvol3 := Notenvolumen(3);
2428:         end;
2429:       15 : begin;
2430:         idx := Rm_Song[mli,3,4];
2431:         if idx < 32 then begin;
2432:           Aktspeed := idx;
2433:           modmultiply :=
2434:             ModPara[idx].mult;
2435:           Speed :=
2436:             ModPara[idx].Speed;
2437:           blockgroesse :=
2438:             ModPara[idx].bgr;
2439:           Dma_abbruch :=
2440:             ModPara[idx].Ab;
2441:         end;
2442:         end;
2443:     end;
2444:   end;
2445:   if Rm_Song[mli,4,3] and $0F <= 15
2446:   then begin;
2447:     case (Rm_Song[mli,4,3] and $0F) of
2448:       01 : begin;
2449:         PermUp_4 := 1;
2450:       end;
2451:       02 : begin;
2452:         PermDo_4 := 1;
2453:       end;
2454:       13 : begin;
2455:         mli := 64;
2456:         end;
2457:       11 : begin;
2458:         mli := 64;
2459:         mlj := Rm_Song[mli,4,4];
2460:         end;
2461:       12 : begin;

```

Listing 1. Die Unit »modplay.pas« gestattet die Ausgabe von MOD-Dateien während des Ablaufs von Applikationen.


```

2462:      notvol4 := Notenvolumen(4);
2463:    end;
2464:    15 : begin;
2465:      idx := Rm_Song[mli,4,4];
2466:      if idx < 32 then begin;
2467:        Aktspeed := idx;
2468:        modmultiply :=
2469:          ModPara[idx].mult;
2470:        Speed :=
2471:          ModPara[idx].Speed;
2472:        blockgroesse :=
2473:          ModPara[idx].bgr;
2474:        Dmaabbruch :=
2475:          ModPara[idx].Ab;
2476:      end;
2477:    end;
2478:  end;
2479: end;
2480: end;
2481: procedure nmw_all_8;
2482: var idx : byte;
2483: begin;
2484:   inc(mli);
2485:   if mli > 64 then mli := 1;
2486:   if mli = 1 then begin;
2487:     inc(mlj);
2488:     if mlj > Liedlaenge then begin;
2489:       if mloop then begin;
2490:         mlj := 1;
2491:         Modp := Addr(Rm_Song);
2492:         memmove(pt(rm[lied[mlj]]),
2493:           pt(Modp),1024);
2494:       end else begin;
2495:         asm
2496:           call [periodisch_anhalten]
2497:         end;
2498:         music_aus := true;
2499:       end;
2500:     end else begin;
2501:       Modp := Addr(Rm_Song);
2502:       memmove(pt(rm[lied[mlj]]),
2503:         pt(Modp),1024);
2504:     end;
2505:   end;
2506:   notel := (Rm_Song[mli,1,1] AND $F0) +
2507:     ((Rm_Song[mli,1,3] AND $F0) shr 4);
2508:   note2 := (Rm_Song[mli,2,1] AND $F0) +
2509:     ((Rm_Song[mli,2,3] AND $F0) shr 4);
2510:   note3 := (Rm_Song[mli,3,1] AND $F0) +
2511:     ((Rm_Song[mli,3,3] AND $F0) shr 4);
2512:   note4 := (Rm_Song[mli,4,1] AND $F0) +
2513:     ((Rm_Song[mli,4,3] AND $F0) shr 4);
2514:   note5 := (Rm_Song[mli,5,1] AND $F0) +
2515:     ((Rm_Song[mli,5,3] AND $F0) shr 4);
2516:   note6 := (Rm_Song[mli,6,1] AND $F0) +
2517:     ((Rm_Song[mli,6,3] AND $F0) shr 4);
2518:   note7 := (Rm_Song[mli,7,1] AND $F0) +
2519:     ((Rm_Song[mli,7,3] AND $F0) shr 4);
2520:   note8 := (Rm_Song[mli,8,1] AND $F0) +
2521:     ((Rm_Song[mli,8,3] AND $F0) shr 4);
2522:   if notel < 0 then
2523:   begin;
2524:     Noten_Anschlag[1] := 500;
2525:     In_St1 := note1;
2526:     inst1 := Ptr(pt(Samp[In_St1]).sgm,
2527:       pt(Samp[In_St1]).ofs);
2528:     i1 := pt(inst1);
2529:     in11 := Sam_l[In_St1];
2530:     in1p := 0;
2531:     notvol1 := Sam_vol[In_St1];
2532:     Pnk1 := 0;
2533:   end;
2534:   if note2 < 0 then
2535:   begin;
2536:     Noten_Anschlag[2] := 500;
2537:     In_St2 := note2;
2538:     inst2 := Ptr(pt(Samp[In_St2]).sgm,
2539:       pt(Samp[In_St2]).ofs);
2540:     i2 := pt(inst2);
2541:     in21 := Sam_l[In_St2];
2542:     in2p := 0;
2543:     notvol2 := Sam_vol[In_St2];
2544:     Pnk2 := 0;
2545:   end;
2546:   if note3 < 0 then
2547:   begin;
2548:     Noten_Anschlag[3] := 500;
2549:     In_St3 := note3;
2550:     inst3 := Ptr(pt(Samp[In_St3]).sgm,
2551:       pt(Samp[In_St3]).ofs);
2552:     i3 := pt(inst3);
2553:     in31 := Sam_l[In_St3]-500;
2554:     in3p := 0;
2555:     notvol3 := Sam_vol[In_St3];
2556:     Pnk3 := 0;
2557:   end;
2558:   if note4 < 0 then
2559:   begin;
2560:     Noten_Anschlag[4] := 500;
2561:     In_St4 := note4;
2562:     inst4 := Ptr(pt(Samp[In_St4]).sgm,
2563:       pt(Samp[In_St4]).ofs);
2564:     i4 := pt(inst4);
2565:     in41 := Sam_l[In_St4];
2566:     in4p := 0;
2567:     notvol4 := Sam_vol[In_St4];
2568:     Pnk4 := 0;
2569:   end;
2570:   if note5 < 0 then
2571:   begin;
2572:     Noten_Anschlag[5] := 500;
2573:     In_St5 := note5;
2574:     inst5 := Ptr(pt(Samp[In_St5]).sgm,
2575:       pt(Samp[In_St5]).ofs);
2576:     i5 := pt(inst5);
2577:     in51 := Sam_l[In_St5];
2578:     in5p := 0;
2579:     notvol5 := Sam_vol[In_St5];
2580:     Pnk5 := 0;
2581:   end;

```

```

2582:   if note6 < 0 then
2583:   begin;
2584:     Noten_Anschlag[6] := 500;
2585:     In_St6 := note6;
2586:     inst6 := Ptr(pt(Samp[In_St6]).sgm,
2587:       pt(Samp[In_St6]).ofs);
2588:     i6 := pt(inst6);
2589:     in61 := Sam_l[In_St6];
2590:     in6p := 0;
2591:     notvol6 := Sam_vol[In_St6];
2592:     Pnk6 := 0;
2593:   end;
2594:   if note7 < 0 then
2595:   begin;
2596:     Noten_Anschlag[7] := 500;
2597:     In_St7 := note7;
2598:     inst7 := Ptr(pt(Samp[In_St7]).sgm,
2599:       pt(Samp[In_St7]).ofs);
2600:     i7 := pt(inst7);
2601:     in71 := Sam_l[In_St7];
2602:     in7p := 0;
2603:     notvol7 := Sam_vol[In_St7];
2604:     Pnk7 := 0;
2605:   end;
2606:   if note8 < 0 then
2607:   begin;
2608:     Noten_Anschlag[8] := 500;
2609:     In_St8 := note8;
2610:     inst8 := Ptr(pt(Samp[In_St8]).sgm,
2611:       pt(Samp[In_St8]).ofs);
2612:     i8 := pt(inst8);
2613:     in81 := Sam_l[In_St8];
2614:     in8p := 0;
2615:     notvol8 := Sam_vol[In_St8];
2616:     Pnk8 := 0;
2617:   end;
2618:   if Rm_Song[mli,1,3] and $0F <= 15
2619:   then begin;
2620:     case (Rm_Song[mli,1,3] and $0F) of
2621:       01 : begin;
2622:         PermUp_1 := 1;
2623:       end;
2624:       02 : begin;
2625:         PermDo_1 := 1;
2626:       end;
2627:       13 : begin;
2628:         mli := 64;
2629:       end;
2630:     end;
2631:   end;
2632:   if Rm_Song[mli,2,3] and $0F <= 15
2633:   then begin;
2634:     case (Rm_Song[mli,2,3] and $0F) of
2635:       01 : begin;
2636:         PermUp_2 := 1;
2637:       end;
2638:       02 : begin;
2639:         PermDo_2 := 1;
2640:       end;
2641:       13 : begin;
2642:         mli := 64;
2643:       end;
2644:     end;
2645:   end;
2646:   if Rm_Song[mli,3,3] and $0F <= 15
2647:   then begin;
2648:     case (Rm_Song[mli,3,3] and $0F) of
2649:       01 : begin;
2650:         PermUp_3 := 1;
2651:       end;
2652:       02 : begin;
2653:         PermDo_3 := 1;
2654:       end;
2655:       13 : begin;
2656:         mli := 64;
2657:       end;
2658:     end;
2659:   end;
2660:   if Rm_Song[mli,4,3] and $0F <= 15
2661:   then begin;
2662:     case (Rm_Song[mli,4,3] and $0F) of
2663:       01 : begin;
2664:         PermUp_4 := 1;
2665:       end;
2666:       02 : begin;
2667:         PermDo_4 := 1;
2668:       end;
2669:       13 : begin;
2670:         mli := 64;
2671:       end;
2672:     end;
2673:   end;
2674:   if Rm_Song[mli,5,3] and $0F <= 15
2675:   then begin;
2676:     case (Rm_Song[mli,5,3] and $0F) of
2677:       01 : begin;
2678:         PermUp_5 := 1;
2679:       end;
2680:       02 : begin;
2681:         PermDo_5 := 1;
2682:       end;
2683:       13 : begin;
2684:         mli := 64;
2685:       end;
2686:     end;
2687:   end;
2688:   if Rm_Song[mli,6,3] and $0F <= 15
2689:   then begin;
2690:     case (Rm_Song[mli,6,3] and $0F) of
2691:       01 : begin;
2692:         PermUp_6 := 1;
2693:       end;
2694:       02 : begin;
2695:         PermDo_6 := 1;
2696:       end;
2697:       13 : begin;
2698:         mli := 64;
2699:       end;
2700:     end;
2701:   end;

```

```

2702:   mlj := Rm_Song[mli,3,4];
2703:   end;
2704:   12 : begin;
2705:     notvol3 := Notenvolumen(3);
2706:   end;
2707:   15 : begin;
2708:     idx := Rm_Song[mli,3,4];
2709:     if idx < 32 then begin;
2710:       Aktspeed := idx;
2711:       modmultiply :=
2712:         ModPara[idx].mult;
2713:       Speed :=
2714:         ModPara[idx].Speed;
2715:       blockgroesse :=
2716:         ModPara[idx].bgr;
2717:       Dmaabbruch :=
2718:         ModPara[idx].Ab;
2719:     end;
2720:   end;
2721: end;
2722: end;
2723: if Rm_Song[mli,4,3] and $0F <= 15
2724: then begin;
2725:   case (Rm_Song[mli,4,3] and $0F) of
2726:     01 : begin;
2727:       PermUp_4 := 1;
2728:     end;
2729:     02 : begin;
2730:       PermDo_4 := 1;
2731:     end;
2732:     13 : begin;
2733:       mli := 64;
2734:     end;
2735:   end;
2736: end;
2737: if Rm_Song[mli,5,3] and $0F <= 15
2738: then begin;
2739:   case (Rm_Song[mli,5,3] and $0F) of
2740:     01 : begin;
2741:       PermUp_5 := 1;
2742:     end;
2743:     02 : begin;
2744:       PermDo_5 := 1;
2745:     end;
2746:     13 : begin;
2747:       mli := 64;
2748:     end;
2749:   end;
2750: end;
2751: if Rm_Song[mli,6,3] and $0F <= 15
2752: then begin;
2753:   case (Rm_Song[mli,6,3] and $0F) of
2754:     01 : begin;
2755:       PermUp_6 := 1;
2756:     end;
2757:     02 : begin;
2758:       PermDo_6 := 1;
2759:     end;
2760:     13 : begin;
2761:       mli := 64;
2762:     end;
2763:   end;
2764: end;
2765: if Rm_Song[mli,7,3] and $0F <= 15
2766: then begin;
2767:   case (Rm_Song[mli,7,3] and $0F) of
2768:     01 : begin;
2769:       PermUp_7 := 1;
2770:     end;
2771:     02 : begin;
2772:       PermDo_7 := 1;
2773:     end;
2774:     13 : begin;
2775:       mli := 64;
2776:     end;
2777:   end;
2778: end;
2779: if Rm_Song[mli,8,3] and $0F <= 15
2780: then begin;
2781:   case (Rm_Song[mli,8,3] and $0F) of
2782:     01 : begin;
2783:       PermUp_8 := 1;
2784:     end;
2785:     02 : begin;
2786:       PermDo_8 := 1;
2787:     end;
2788:     13 : begin;
2789:       mli := 64;
2790:     end;
2791:   end;
2792: end;
2793: if Rm_Song[mli,9,3] and $0F <= 15
2794: then begin;
2795:   case (Rm_Song[mli,9,3] and $0F) of
2796:     01 : begin;
2797:       PermUp_9 := 1;
2798:     end;
2799:     02 : begin;
2800:       PermDo_9 := 1;
2801:     end;
2802:     13 : begin;
2803:       mli := 64;
2804:     end;
2805:   end;
2806: end;
2807: if Rm_Song[mli,10,3] and $0F <= 15
2808: then begin;
2809:   case (Rm_Song[mli,10,3] and $0F) of
2810:     01 : begin;
2811:       PermUp_10 := 1;
2812:     end;
2813:     02 : begin;
2814:       PermDo_10 := 1;
2815:     end;
2816:     13 : begin;
2817:       mli := 64;
2818:     end;
2819:   end;
2820: end;
2821: if Rm_Song[mli,11,3] and $0F <= 15
2822: then begin;
2823:   case (Rm_Song[mli,11,3] and $0F) of
2824:     01 : begin;
2825:       PermUp_11 := 1;
2826:     end;
2827:     02 : begin;
2828:       PermDo_11 := 1;
2829:     end;
2830:     13 : begin;
2831:       mli := 64;
2832:     end;
2833:   end;
2834: end;
2835: if Rm_Song[mli,12,3] and $0F <= 15
2836: then begin;
2837:   case (Rm_Song[mli,12,3] and $0F) of
2838:     01 : begin;
2839:       PermUp_12 := 1;
2840:     end;
2841:     02 : begin;
2842:       PermDo_12 := 1;
2843:     end;
2844:     13 : begin;
2845:       mli := 64;
2846:     end;
2847:   end;
2848: end;
2849: if Rm_Song[mli,13,3] and $0F <= 15
2850: then begin;
2851:   case (Rm_Song[mli,13,3] and $0F) of
2852:     01 : begin;
2853:       PermUp_13 := 1;
2854:     end;
2855:     02 : begin;
2856:       PermDo_13 := 1;
2857:     end;
2858:     13 : begin;
2859:       mli := 64;
2860:     end;
2861:   end;
2862: end;
2863: if Rm_Song[mli,14,3] and $0F <= 15
2864: then begin;
2865:   case (Rm_Song[mli,14,3] and $0F) of
2866:     01 : begin;
2867:       PermUp_14 := 1;
2868:     end;
2869:     02 : begin;
2870:       PermDo_14 := 1;
2871:     end;
2872:     13 : begin;
2873:       mli := 64;
2874:     end;
2875:   end;
2876: end;
2877: if Rm_Song[mli,15,3] and $0F <= 15
2878: then begin;
2879:   case (Rm_Song[mli,15,3] and $0F) of
2880:     01 : begin;
2881:       PermUp_15 := 1;
2882:     end;
2883:     02 : begin;
2884:       PermDo_15 := 1;
2885:     end;
2886:     13 : begin;
2887:       mli := 64;
2888:     end;
2889:   end;
2890: end;
2891: if Rm_Song[mli,16,3] and $0F <= 15
2892: then begin;
2893:   case (Rm_Song[mli,16,3] and $0F) of
2894:     01 : begin;
2895:       PermUp_16 := 1;
2896:     end;
2897:     02 : begin;
2898:       PermDo_16 := 1;
2899:     end;
2900:     13 : begin;
2901:       mli := 64;
2902:     end;
2903:   end;
2904: end;
2905: if Rm_Song[mli,17,3] and $0F <= 15
2906: then begin;
2907:   case (Rm_Song[mli,17,3] and $0F) of
2908:     01 : begin;
2909:       PermUp_17 := 1;
2910:     end;
2911:     02 : begin;
2912:       PermDo_17 := 1;
2913:     end;
2914:     13 : begin;
2915:       mli := 64;
2916:     end;
2917:   end;
2918: end;
2919: if Rm_Song[mli,18,3] and $0F <= 15
2920: then begin;
2921:   case (Rm_Song[mli,18,3] and $0F) of
2922:     01 : begin;
2923:       PermUp_18 := 1;
2924:     end;
2925:     02 : begin;
2926:       PermDo_18 := 1;
2927:     end;
2928:     13 : begin;
2929:       mli := 64;
2930:     end;
2931:   end;
2932: end;
2933: if Rm_Song[mli,19,3] and $0F <= 15
2934: then begin;
2935:   case (Rm_Song[mli,19,3] and $0F) of
2936:     01 : begin;
2937:       PermUp_19 := 1;
2938:     end;
2939:     02 : begin;
2940:       PermDo_19 := 1;
2941:     end;
2942:     13 : begin;
2943:       mli := 64;
2944:     end;
2945:   end;
2946: end;
2947: if Rm_Song[mli,20,3] and $0F <= 15
2948: then begin;
2949:   case (Rm_Song[mli,20,3] and $0F) of
2950:     01 : begin;
2951:       PermUp_20 := 1;
2952:     end;
2953:     02 : begin;
2954:       PermDo_20 := 1;
2955:     end;
2956:     13 : begin;
2957:       mli := 64;
2958:     end;
2959:   end;
2960: end;
2961: if Rm_Song[mli,21,3] and $0F <= 15
2962: then begin;
2963:   case (Rm_Song[mli,21,3] and $0F) of
2964:     01 : begin;
2965:       PermUp_21 := 1;
2966:     end;
2967:     02 : begin;
2968:       PermDo_21 := 1;
2969:     end;
2970:     13 : begin;
2971:       mli := 64;
2972:     end;
2973:   end;
2974: end;
2975: if Rm_Song[mli,22,3] and $0F <= 15
2976: then begin;
2977:   case (Rm_Song[mli,22,3] and $0F) of
2978:     01 : begin;
2979:       PermUp_22 := 1;
2980:     end;
2981:     02 : begin;
2982:       PermDo_22 := 1;
2983:     end;
2984:     13 : begin;
2985:       mli := 64;
2986:     end;
2987:   end;
2988: end;
2989: if Rm_Song[mli,23,3] and $0F <= 15
2990: then begin;
2991:   case (Rm_Song[mli,23,3] and $0F) of
2992:     01 : begin;
2993:       PermUp_23 := 1;
2994:     end;
2995:     02 : begin;
2996:       PermDo_23 := 1;
2997:     end;
2998:     13 : begin;
2999:       mli := 64;
3000:     end;
3001:   end;
3002: end;
3003: if Rm_Song[mli,24,3] and $0F <= 15
3004: then begin;
3005:   case (Rm_Song[mli,24,3] and $0F) of
3006:     01 : begin;
3007:       PermUp_24 := 1;
3008:     end;
3009:     02 : begin;
3010:       PermDo_24 := 1;
3011:     end;
3012:     13 : begin;
3013:       mli := 64;
3014:     end;
3015:   end;
3016: end;
3017: if Rm_Song[mli,25,3] and $0F <= 15
3018: then begin;
3019:   case (Rm_Song[mli,25,3] and $0F) of
3020:     01 : begin;
3021:       PermUp_25 := 1;
3022:     end;
3023:     02 : begin;
3024:       PermDo_25 := 1;
3025:     end;
3026:     13 : begin;
3027:       mli := 64;
3028:     end;
3029:   end;
3030: end;
3031: if Rm_Song[mli,26,3] and $0F <= 15
3032: then begin;
3033:   case (Rm_Song[mli,26,3] and $0F) of
3034:     01 : begin;
3035:       PermUp_26 := 1;
3036:     end;
3037:     02 : begin;
3038:       PermDo_26 := 1;
3039:     end;
3040:     13 : begin;
3041:       mli := 64;
3042:     end;
3043:   end;
3044: end;
3045: if Rm_Song[mli,27,3] and $0F <= 15
3046: then begin;
3047:   case (Rm_Song[mli,27,3] and $0F) of
3048:     01 : begin;
3049:       PermUp_27 := 1;
3050:     end;
3051:     02 : begin;
3052:       PermDo_27 := 1;
3053:     end;
3054:     13 : begin;
3055:       mli := 64;
3056:     end;
3057:   end;
3058: end;
3059: if Rm_Song[mli,28,3] and $0F <= 15
3060: then begin;
3061:   case (Rm_Song[mli,28,3] and $0F) of
3062:     01 : begin;
3063:       PermUp_28 := 1;
3064:     end;
3065:     02 : begin;
3066:       PermDo_28 := 1;
3067:     end;
3068:     13 : begin;
3069:       mli := 64;
3070:     end;
3071:   end;
3072: end;
3073: if Rm_Song[mli,29,3] and $0F <= 15
3074: then begin;
3075:   case (Rm_Song[mli,29,3] and $0F) of
3076:     01 : begin;
3077:       PermUp_29 := 1;
3078:     end;
3079:     02 : begin;
3080:       PermDo_29 := 1;
3081:     end;
3082:     13 : begin;
3083:       mli := 64;
3084:     end;
3085:   end;
3086: end;
3087: if Rm_Song[mli,30,3] and $0F <= 15
3088: then begin;
3089:   case (Rm_Song[mli,30,3] and $0F) of
3090:     01 : begin;
3091:       PermUp_30 := 1;
3092:     end;
3093:     02 : begin;
3094:       PermDo_30 := 1;
3095:     end;
3096:     13 : begin;
3097:       mli := 64;
3098:     end;
3099:   end;
3100: end;
3101: if Rm_Song[mli,31,3] and $0F <= 15
3102: then begin;
3103:   case (Rm_Song[mli,31,3] and $0F) of
3104:     01 : begin;
3105:       PermUp_31 := 1;
3106:     end;
3107:     02 : begin;
3108:       PermDo_31 := 1;
3109:     end;
3110:     13 : begin;
3111:       mli := 64;
3112:     end;
3113:   end;
3114: end;
3115: if Rm_Song[mli,32,3] and $0F <= 15
3116: then begin;
3117:   case (Rm_Song[mli,32,3] and $0F) of
3118:     01 : begin;
3119:       PermUp_32 := 1;
3120:     end;
3121:     02 : begin;
3122:       PermDo_32 := 1;
3123:     end;
3124:     13 : begin;
3125:       mli := 64;
3126:     end;
3127:   end;
3128: end;
3129: if Rm_Song[mli,33,3] and $0F <= 15
3130: then begin;
3131:   case (Rm_Song[mli,33,3] and $0F) of
3132:     01 : begin;
3133:       PermUp_33 := 1;
3134:     end;
3135:     02 : begin;
3136:       PermDo_33 := 1;
3137:     end;
3138:     13 : begin;
3139:       mli := 64;
3140:     end;
3141:   end;
3142: end;
3143: if Rm_Song[mli,34,3] and $0F <= 15
3144: then begin;
3145:   case (Rm_Song[mli,34,3] and $0F) of
3146:     01 : begin;
3147:       PermUp_34 := 1;
3148:     end;
3149:     02 : begin;
3150:       PermDo_34 := 1;
3151:     end;
3152:     13 : begin;
3153:       mli := 64;
3154:     end;
3155:   end;
3156: end;
3157: if Rm_Song[mli,35,3] and $0F <= 15
3158: then begin;
3159:   case (Rm_Song[mli,35,3] and $0F) of
3160:     01 : begin;
3161:       PermUp_35 := 1;
3162:     end;
3163:     02 : begin;
3164:       PermDo_35 := 1;
3165:     end;
3166:     13 : begin;
3167:       mli := 64;
3168:     end;
3169:   end;
3170: end;
3171: if Rm_Song[mli,36,3] and $0F <= 15
3172: then begin;
3173:   case (Rm_Song[mli,36,3] and $0F) of
3174:     01 : begin;
3175:       PermUp_36 := 1;
3176:     end;
3177:     02 : begin;
3178:       PermDo_36 := 1;
3179:     end;
3180:     13 : begin;
3181:       mli := 64;
3182:     end;
3183:   end;
3184: end;
3185: if Rm_Song[mli,37,3] and $0F <= 15
3186: then begin;
3187:   case (Rm_Song[mli,37,3] and $0F) of
3188:     01 : begin;
3189:       PermUp_37 := 1;
3190:     end;
3191:     02 : begin;
3192:       PermDo_37 := 1;
3193:     end;
3194:     13 : begin;
3195:       mli := 64;
3196:     end;
3197:   end;
3198: end;
3199: if Rm_Song[mli,38,3] and $0F <= 15
3200: then begin;
3201:   case (Rm_Song[mli,38,3] and $0F) of
3202:     01 : begin;
3203:       PermUp_38 := 1;
3204:     end;
3205:     02 : begin;
3206:       PermDo_38 := 1;
3207:     end;
3208:     13 : begin;
3209:       mli := 64;
3210:     end;
3211:   end;
3212: end;
3213: if Rm_Song[mli,39,3] and $0F <= 15
3214: then begin;
3215:   case (Rm_Song[mli,39,3] and $0F) of
3216:     01 : begin;
3217:       PermUp_39 := 1;
3218:     end;
3219:     02 : begin;
3220:       PermDo_39 := 1;
3221:     end;
3222:     13 : begin;
3223:       mli := 64;
3224:     end;
3225:   end;
3226: end;
3227: if Rm_Song[mli,40,3] and $0F <= 15
3228: then begin;
3229:   case (Rm_Song[mli,40,3] and $0F) of
3230:     01 : begin;
3231:       PermUp_40 := 1;
3232:     end;
3233:     02 : begin;
3234:       PermDo_40 := 1;
3235:     end;
3236:     13 : begin;
3237:       mli := 64;
3238:     end;
3239:   end;
3240: end;
3241: if Rm_Song[mli,41,3] and $0F <= 15
3242: then begin;
3243:   case (Rm_Song[mli,41,3] and $0F) of
3244:     01 : begin;
3245:       PermUp_41 := 1;
3246:     end;
3247:     02 : begin;
3248:       PermDo_41 := 1;
3249:     end;
3250:     13 : begin;
3251:       mli := 64;
3252:     end;
3253:   end;
3254: end;
3255: if Rm_Song[mli,42,3] and $0F <= 15
3256: then begin;
3257:   case (Rm_Song[mli,42,3] and $0F) of
3258:     01 : begin;
3259:       PermUp_42 := 1;
3260:     end;
3261:     02 : begin;
3262:       PermDo_42 := 1;
3263:     end;
3264:     13 : begin;
3265:       mli := 64;
3266:     end;
3267:   end;
3268: end;
3269: if Rm_Song[mli,43,3] and $0F <= 15
3270: then begin;
3271:   case (Rm_Song[mli,43,3] and $0F) of
3272:     01 : begin;
3273:       PermUp_43 := 1;
3274:     end;
3275:     02 : begin;
3276:       PermDo_43 := 1;
3277:     end;
3278:     13 : begin;
3279:       mli := 64;
3280:     end;
3281:   end;
3282: end;
3283: if Rm_Song[mli,44,3] and $0F <= 15
3284: then begin;
3285:   case (Rm_Song[mli,44,3] and $0F) of
3286:     01 : begin;
3287:       PermUp_44 := 1;
3288:     end;
3289:     02 : begin;
3290:       PermDo_44 := 1;
3291:     end;
3292:     13 : begin;
3293:       mli := 64;
3294:     end;
3295:   end;
3296: end;
3297: if Rm_Song[mli,45,3] and $0F <= 15
3298: then begin;
3299:   case (Rm_Song[mli,45,3] and $0F) of
3300:     01 : begin;
3301:       PermUp_45 := 1;
3302:     end;
3303:     02 : begin;
3304:       PermDo_45 := 1;
3305:     end;
3306:     13 : begin;
3307:       mli := 64;
3308:     end;
3309:   end;
3310: end;
3311: if Rm_Song[mli,46,3] and $0F <= 15
3312: then begin;
3313:   case (Rm_Song[mli,46,3] and $0F) of
3314:     01 : begin;
3315:       PermUp_46 := 1;
3316:     end;
3317:     02 : begin;
3318:       PermDo
```



```

2814:         if idx < 32 then begin;
2815:             Aktspeed := idx;
2816:             modmultiply :=
2817:                 ModPara[idx].mult;
2818:             Speed :=
2819:                 ModPara[idx].Speed;
2820:             blockgroesse :=
2821:                 ModPara[idx].bgr;
2822:             Dma_abbruch :=
2823:                 ModPara[idx].Ab;
2824:         end;
2825:     end;
2826: end;
2827: if Rm_Song[mli,7,3] and $0F <= 15
2828: then begin;
2829:     case (Rm_Song[mli,7,3] and $0F) of
2830:     01 : begin;
2831:         PermUp_7 := 1;
2832:     end;
2833:     02 : begin;
2834:         PermDo_7 := 1;
2835:     end;
2836:     13 : begin;
2837:         mli := 64;
2838:     end;
2839:     11 : begin;
2840:         mli := 64;
2841:         mlj := Rm_Song[mli,7,4];
2842:     end;
2843:     12 : begin;
2844:         notvol17 := Notenvolumen(7);
2845:     end;
2846:     15 : begin;
2847:         idx := Rm_Song[mli,7,4];
2848:         if idx < 32 then begin;
2849:             Aktspeed := idx;
2850:             modmultiply :=
2851:                 ModPara[idx].mult;
2852:             Speed :=
2853:                 ModPara[idx].Speed;
2854:             blockgroesse :=
2855:                 ModPara[idx].bgr;
2856:             Dma_abbruch :=
2857:                 ModPara[idx].Ab;
2858:         end;
2859:     end;
2860: end;
2861: end;
2862: if Rm_Song[mli,8,3] and $0F <= 15
2863: then begin;
2864:     case (Rm_Song[mli,8,3] and $0F) of
2865:     01 : begin;
2866:         PermUp_8 := 1;
2867:     end;
2868:     02 : begin;
2869:         PermDo_8 := 1;
2870:     end;
2871:     13 : begin;
2872:         mli := 64;
2873:     end;
2874:     11 : begin;
2875:         mli := 64;
2876:         mlj := Rm_Song[mli,8,4];
2877:     end;
2878:     12 : begin;
2879:         notvol18 := Notenvolumen(8);
2880:     end;
2881:     15 : begin;
2882:         idx := Rm_Song[mli,8,4];
2883:         if idx < 32 then begin;
2884:             modmultiply :=
2885:                 ModPara[idx].mult;
2886:             Speed :=
2887:                 ModPara[idx].Speed;
2888:             blockgroesse :=
2889:                 ModPara[idx].bgr;
2890:             Dma_abbruch :=
2891:                 ModPara[idx].Ab;
2892:         end;
2893:     end;
2894: end;
2895: end;
2896: end;
2897: end;
2898: procedure initialisiere_vermischen;
2899: begin;
2900:     ziel := pt(blk);
2901:     asm
2902:     call [vermische_proc]
2903:     end;
2904: end;
2905: FUNCTION ConvertString(Source :
2906:     Pointer; Size : BYTE):String;
2907: VAR
2908:     WorkStr : String;
2909: BEGIN
2910:     Move(Source^,WorkStr[1],Size);
2911:     WorkStr[0] := CHR(Size);
2912:     ConvertString := WorkStr;
2913: END;
2914: function init_Song : boolean;
2915: var rmod : file;
2916: sgr : word;
2917: { Größe eines Sampls }
2918: inststart : longint;
2919: { Position in Datei, wo Sampledaten
2920:     starten }
2921: datgr : longint;
2922: { Die Größe der MOD - Datei }
2923: Mkg : array[1..4] of char;
2924: { für Modtyp - Erkennung }
2925: hilfsp : ^byte;
2926: strptr : pointer;
2927: begin;
2928:     In_St1 := 0;
2929:     In_St2 := 0;
2930:     In_St3 := 0;
2931:     In_St4 := 0;
2932:     In_St5 := 0;
2933:     In_St6 := 0;

```

```

2934:     In_St7 := 0;
2935:     In_St8 := 0;
2936:     getmem(blk,8000);
2937:     { Speicher für Datenpuffer
2938:         reservieren }
2939:     for mlj := 0 to 128 do
2940:         Lied[mlj] := 0;
2941:     {SI-}
2942:     assign(rmod,Mod_Name);
2943:     reset(rmod,1);
2944:     {SI+}
2945:     if IOresult <> 0 then begin;
2946:         init_song := false;
2947:         exit;
2948:     end;
2949:     datgr := filesize(rmod);
2950:     inststart := datgr;
2951:     for mlj := 1 to 31 do begin;
2952:         seek(rmod,42+(mlj-1)*30);
2953:         blockread(rmod,sgr,2);
2954:         sgr := swap(sgr) * 2;
2955:         if sgr < 0 then inststart :=
2956:             inststart - sgr;
2957:         Sam_l[mlj] := sgr;
2958:         seek(rmod,45+(mlj-1)*30);
2959:         blockread(rmod,Sam_vol[mlj],1);
2960:         blockread(rmod,loop_s[mlj],2);
2961:         blockread(rmod,loop_l[mlj],2);
2962:         loop_s[mlj] := swap(loop_s[mlj])*2;
2963:         loop_l[mlj] := swap(loop_l[mlj])*2;
2964:     end;
2965:     seek(rmod,1080);
2966:     blockread(rmod,Mkg,4);
2967:     if ((Mkg[1] = '8')
2968:         AND (Mkg[2] = 'C'))
2969:         AND ((Mkg[3] = 'H')
2970:             AND (Mkg[4] = 'N'))
2971:     then begin
2972:         { 8 - Stimmige MOD-Datei }
2973:         Pattgroesse := 2048;
2974:         Stimmen := 8;
2975:         Vermische_Proc :=
2976:             @vermische_start_8;
2977:         nmw_Proc := @nmw_all_8;
2978:         inner_proc :=
2979:             @innen_schleife_8n;
2980:     end else begin;
2981:         { 4-Stimmige MOD-Datei }
2982:         Pattgroesse := 1024;
2983:         Stimmen := 4;
2984:         Vermische_Proc :=
2985:             @vermische_start_4;
2986:         nmw_Proc := @nmw_all_4;
2987:         inner_proc :=
2988:             @innen_schleife_4;
2989:     end;
2990:     seek(rmod,inststart);
2991:     for mlj := 1 to 31 do begin;
2992:         getmem(Samp[mlj],Sam_l[mlj]);
2993:         blockread(rmod,Samp[mlj]^,
2994:             sam_l[mlj]);
2995:     end;
2996:     datgr := inststart - 1083;
2997:     pat_anz := datgr div Pattgroesse;
2998:     for mlj := 0 to pat_anz-1 do begin;
2999:         getmem(rm[mlj],2048);
3000:         fillchar(rm[mlj]^,2048,0);
3001:         seek(rmod,1084+mlj*Pattgroesse);
3002:         hilfsp := ptr(seg(rm[mlj]^),
3003:             ofs(rm[mlj]^));
3004:         for mli := 0 to 63 do begin;
3005:             hilfsp := ptr(seg(rm[mlj]^),
3006:                 ofs(rm[mlj]^)+mli*32);
3007:             blockread(rmod,hilfsp^,
3008:                 Pattgroesse
3009:                 div 64);
3010:         end;
3011:     end;
3012:     seek(rmod,952);
3013:     blockread(rmod,Lied,128);
3014:     getmem(strptr,25);
3015:     for i := 0 to 30 do begin;
3016:         seek(rmod,20+i*30);
3017:         blockread(rmod,strptr^,22);
3018:         instnamen[i+1] :=
3019:             convertstring(strptr,22);
3020:     end;
3021:     seek(rmod,0);
3022:     blockread(rmod,strptr^,20);
3023:     songname := convertstring(strptr,
3024:         20);
3025:     seek(rmod,950);
3026:     blockread(rmod,Liedlaenge,1);
3027:     freemem(strptr,25);
3028:     mlj := 0;
3029:     mli := 0;
3030:     close(rmod);
3031:     init_song := true;
3032: end;
3033: procedure exit_song;
3034: begin;
3035:     Port[dsp_adr+$C] := $D3;
3036:     halt(0);
3037: end;
3038: procedure Free_Soundmem;
3039: begin;
3040:     if music_played then begin;
3041:         freemem(blk,8000);
3042:         for mlj := 0 to pat_anz-1 do
3043:             begin;
3044:                 freemem(rm[mlj],2048);
3045:             end;
3046:         end;
3047:     end;
3048: procedure init_sbperiod(p : pointer);
3049: begin;
3050:     periodisch_anhalten := p;
3051: end;
3052: procedure mod_SetLoopflag(loopen
3053: : boolean);

```

```

3054: begin;
3055:     mloop := loopen;
3056: end;
3057: procedure mod_SetMultiply(msm :
3058:     word);
3059: begin;
3060:     modmultiply := msm;
3061: end;
3062: procedure mod_SetLoop(msl : word);
3063: begin;
3064:     Sound_Schleifen := msl;
3065:     loop3 := msl;
3066: end;
3067: procedure mod_SetSpeed(msp : word);
3068: begin;
3069:     speed := msp;
3070:     Speed3 := msp;
3071: end;
3072: procedure mod_transpose(
3073:     transposerwert : integer);
3074: begin;
3075:     tpw := transposerwert;
3076: end;
3077: procedure init_data;
3078: begin;
3079:     m_played := false;
3080:     In_St1 := 0;
3081:     In_St2 := 0;
3082:     In_St3 := 0;
3083:     In_St4 := 0;
3084:     In_St5 := 0;
3085:     In_St6 := 0;
3086:     In_St7 := 0;
3087:     In_St8 := 0;
3088: end;
3089: procedure init_Paramtable;
3090: var ls : byte;
3091: h : real;
3092: begin;
3093:     for ls := 1 to 31 do begin;
3094:         ModPara[ls].mult := 125
3095:         (90+((ls)*2));
3096:         ModPara[ls].Speed := ls*ModPara[ls].
3097:             mult div 10;
3098:         ModPara[ls].bgr := ModPara[ls].Speed
3099:             *Sound_Schleifen;
3100:         h := Sampling_Frequenz /
3101:             ModPara[ls].
3102:             bgr;
3103:         ModPara[ls].Ab :=
3104:             round((timer_per_second/h) * 0.9);
3105:     end;
3106: end;
3107: procedure mod_Samplefreq(Rate :
3108:     integer);
3109: var h : real;
3110: begin;
3111:     Speed := 66;
3112:     case Rate of
3113:     08 : begin;
3114:         Sampling_Rate := 131;
3115:         set_timeconst_sbl6(131);
3116:         mod_transpose(1);
3117:         mod_SetLoop(15);
3118:         blockgroesse := Speed *
3119:             Sound_Schleifen;
3120:         Sampling_Frequenz := 8000;
3121:         h := Sampling_Frequenz /
3122:             blockgroesse;
3123:         Dma_abbruch := round((
3124:             timer_per_second/h) * 0.9);
3125:         init_Paramtable;
3126:     end;
3127:     10 : begin;
3128:         Sampling_Rate := 156;
3129:         set_timeconst_sbl6(156);
3130:         mod_transpose(5);
3131:         mod_SetLoop(19);
3132:         blockgroesse := Speed *
3133:             Sound_Schleifen;
3134:         Sampling_Frequenz := 10000;
3135:         h := Sampling_Frequenz /
3136:             blockgroesse;
3137:         Dma_abbruch := round((
3138:             timer_per_second/h) * 0.9);
3139:         init_Paramtable;
3140:     end;
3141:     13 : begin;
3142:         Sampling_Rate := 181;
3143:         set_timeconst_sbl6(181);
3144:         mod_transpose(10);
3145:         mod_SetLoop(25);
3146:         blockgroesse := Speed *
3147:             Sound_Schleifen;
3148:         Sampling_Frequenz := 13333;
3149:         h := Sampling_Frequenz /
3150:             blockgroesse;
3151:         Dma_abbruch := round((
3152:             timer_per_second/h) * 0.9);
3153:         init_Paramtable;
3154:     end;
3155:     16 : begin;
3156:         Sampling_Rate := 196;
3157:         set_timeconst_sbl6(196);
3158:         mod_transpose(14);
3159:         mod_SetLoop(32);
3160:         blockgroesse := Speed *
3161:             Sound_Schleifen;
3162:         Sampling_Frequenz := 16666;
3163:         h := Sampling_Frequenz /
3164:             blockgroesse;
3165:         Dma_abbruch := round((

```

Listing 1. Die Unit »modplay.pas« gestattet die Ausgabe von MOD-Dateien während des Ablaufs von Applikationen.


```

3166: timer_per_second/h) * 0.9);
3167: init_Paratable;
3168: end;
3169: 22 : begin;
3170:   Sampling_Rate := 211;
3171:   set_timeconst_sbl6(211);
3172:   mod_transpose(19);
3173:   mod_SetLoop(42);
3174:   blockgroesse := Speed *
3175:   Sound_Schleifen;
3176:   Sampling_Frequenz := 22222;
3177:   h := Sampling_Frequenz /
3178:   blockgroesse;
3179:   Dma_abbruch := round((
3180:   timer_per_second/h) * 0.9);
3181:   init_Paratable;
3182: end;
3183: end;
3184: end;
3185: procedure gsreg; assembler;
3186: asm
3187: mov gax,ax
3188: mov gbx,bx
3189: mov gcx,cx
3190: mov gdx,dx
3191: mov gsi,si
3192: mov gdi,di
3193: mov ax,es
3194: mov ges,ax
3195: mov ax,ds
3196: mov gds,ax
3197: end;
3198: procedure grreg; assembler;
3199: asm
3200: mov bx,gbx
3201: mov cx,gcx
3202: mov dx,gdx
3203: mov si,gsi
3204: mov di,gdi
3205: mov ax,ges
3206: mov es,ax
3207: mov ax,gds
3208: mov ds,ax
3209: mov ax,gax
3210: end;
3211:
3212: {$F+}
3213: procedure Sound_handler;
3214: begin;
3215: if mycli <> 0 then exit;
3216: mycli := 1;
3217: inc(Loop_pos);
3218: if (Loop_pos > Speed) then begin;
3219:   if dma_Zaehler >= dma_abbruch
3220:   then begin;
3221:     dma_Zaehler := 0;
3222:     if phase_1 then begin;
3223:       dsp_block_sbl6;
3224:       phase_1 := false;
3225:       phase_2 := true;
3226:       Loop_pos := 0;
3227:     end;
3228:   end;
3229:   if phase_2 then begin;
3230:     asm
3231:       call [nmw_proc]
3232:     end;
3233:     Initialisiere_Vermischen;
3234:     phase_2 := false;
3235:     phase_1 := true;
3236:   end;
3237: end else begin;
3238: if Noten_Anschlag[1] > 1 then
3239:   dec(Noten_Anschlag[1]);
3240: if Noten_Anschlag[2] > 1 then
3241:   dec(Noten_Anschlag[2]);
3242: if Noten_Anschlag[3] > 1 then
3243:   dec(Noten_Anschlag[3]);
3244: if Noten_Anschlag[4] > 1 then
3245:   dec(Noten_Anschlag[4]);
3246: if Noten_Anschlag[5] > 1 then
3247:   dec(Noten_Anschlag[5]);
3248: if Noten_Anschlag[6] > 1 then
3249:   dec(Noten_Anschlag[6]);
3250: if Noten_Anschlag[7] > 1 then
3251:   dec(Noten_Anschlag[7]);
3252: if Noten_Anschlag[8] > 1 then
3253:   dec(Noten_Anschlag[8]);
3254:   asm
3255:     call [innen_proc]
3256:   end;
3257: end;
3258: mycli := 0;
3259: end;
3260: {$F-}
3261:
3262: {$F+}
3263: procedure calculate_music; assembler;
3264: asm
3265: cmp mycli,0
3266: jne @ende_stop
3267: cmp music_aus,0
3268: jne @ende_stop
3269: call gsreg
3270: call Sound_handler
3271: call grreg
3272: @ende_stop:
3273: end;
3274: {$F-}
3275: procedure mod_waitretrace;
3276: var dl : integer;
3277: begin;
3278: dl := 1;
3279: while (dl <= 30)
3280: or (waitret_flag < frames) do begin;
3281:   calculate_music;
3282:   inc(dl);
3283: end;
3284: in_retrace := true;
3285: waitret_flag := 0;

```

```

3286: asm
3287:   push dx
3288: @ll:
3289:   mov dx,3dah
3290:   in al,dx
3291:   and al,8h
3292:   jnz @ll
3293: @l2:
3294:   mov dx,3dah
3295:   in al,dx
3296:   and al,8h
3297:   jz @l2
3298:   pop dx
3299: end;
3300: in_retrace := false;
3301: end;
3302: procedure StelleTimerEin(Proc :
3303:   pointer;
3304:   Freq : word);
3305: {
3306:   Der Timer wird so umprogrammiert,
3307:   daß er mit der übergebenen Frequenz
3308:   die in Proc übergebene Procedure
3309:   aufruft. Der alte Timerinterrupt
3310:   wird auf INT $71 verbogen
3311: }
3312: var izaehler : word;
3313: oldv : pointer;
3314: begin;
3315:   asm cli end;
3316:   izaehler := 1193180 DIV Freq;
3317:   Port[$43] := $36;
3318:   Port[$40] := Lo(izaehler);
3319:   Port[$40] := Hi(izaehler);
3320:   GetIntVec(8,OldV);
3321:   setIntVec(OldTimerInt,OldV);
3322:   SetIntVec(8,Proc);
3323:   old_tzaehler := 1;
3324:   dma_zzaehler := 0;
3325:   seczaehler := 0;
3326:   Timerzaehler := timer_per_second
3327:   div 18;
3328:   asm sti end;
3329: end;
3330: procedure StelleTimerAus;
3331: {
3332:   Stellt den Urzustand des Timers her
3333: }
3334: var oldv : pointer;
3335: begin;
3336:   asm cli end;
3337:   port[$43] := $36;
3338:   Port[$40] := 0;
3339:   Port[$40] := 0;
3340:   GetIntVec(OldTimerInt,OldV);
3341:   SetIntVec(8,OldV);
3342:   asm sti end;
3343: end;
3344: procedure NeuerTimer; interrupt;
3345: {
3346:   Unser neuer Timer-Interrupt
3347: }
3348: var dummyreg : registers;
3349: begin;
3350:   syreg;
3351:   inc(waitret_flag);
3352:   inc(Dma_Zaehler);
3353:   inc(Seczaehler);
3354:   dec(Timerzaehler);
3355:   if Seczaehler = timer_per_second
3356:   then begin;
3357:     Seczaehler := 0;
3358:     inc(Laufsec);
3359:     if Laufsec = 60 then begin;
3360:       inc(Laufmin);
3361:       Laufsec := 0;
3362:     end;
3363:   end;
3364:   if Timerzaehler = 0 then begin;
3365:     Timerzaehler := timer_per_second
3366:     div 18;
3367:     intr(OldTimerInt,dummyreg);
3368:   end;
3369:   if not in_retrace then
3370:     calculate_Music;
3371:   Port[$20] := $20;
3372:   ryreg;
3373: end;
3374: procedure init_sb;
3375: begin;
3376:   init_sbl6;
3377: end;
3378: function Lade_Soundeffekt(s : string;
3379:   var ef : effect_type) : integer;
3380: var efi : file;
3381: begin;
3382:   assign(efi,s);
3383:   reset(efi,1);
3384:   {$I+}
3385:   if IOResult <> 0 then begin;
3386:     {$I-}
3387:     Lade_Soundeffekt := -1;
3388:     exit;
3389:   end;
3390:   {$I-}
3391:   efi := filesize(efi);
3392:   getmem(ef.p,ef.1);
3393:   blockread(efi,ef.p^,ef.1);
3394:   close(efi);
3395:   Playeffect := false;
3396:   Lade_Soundeffekt := 0;
3397: end;
3398: procedure dispose_Soundeffekt(ef
3399:   : effect_type);
3400: begin;
3401:   freemem(ef.p,ef.1);
3402: end;
3403: procedure Starte_Soundeffekt(ef :
3404:   effect_type;frequenz,Vol,styp :
3405:   word);

```

```

3406: {
3407:   Startet den Soundeffekt
3408: }
3409: var realwert : real;
3410: begin;
3411:   effektgroesse := ef.1;
3412:   realwert := frequenz/
3413:   Sampling_frequenz;
3414:   Effvk := trunc(realwert);
3415:   Effnk := round((realwert-Effvk)*
3416:   100);
3417:   Effekt := pt(ef.p);
3418:   Effistvk := Effekt.ofs;
3419:   Effistnk := 0;
3420:   Effektposi := 0;
3421:   effectvolume := vol;
3422:   convertteff := styp;
3423:   Playeffect := true;
3424: end;
3425: function lade_moddatei(modname :
3426:   string;
3427:   ispeed,iloop : integer) : integer;
3428: {
3429:   Läd die Moddatei. Returnwerte:
3430:   -2 Wenn Datei nicht gefunden
3431:   -1 Wenn Initialisierung fehlerhaft
3432:   0 Wenn alles O.K.
3433: }
3434: var df : file;
3435: fgr : longint;
3436: begin;
3437:   wr_dsp_sbl6($D0);
3438:   Mod_Name := modname;
3439:   assign(df,Mod_name);
3440:   reset(df,1);
3441:   {$I+}
3442:   if IOResult <> 0 then begin;
3443:     {$I-}
3444:     close(df);
3445:     lade_moddatei := -2;
3446:     exit;
3447:   end;
3448:   {$I-}
3449:   fgr := filesize(df);
3450:   close(df);
3451:   music_played := true;
3452:   music_aus := false;
3453:   if ispeed <> AUTO then Speed3 :=
3454:   ispeed;
3455:   if loop <> AUTO then Loop3 :=
3456:   loop;
3457:   init_data;
3458:   if init_song then begin;
3459:     phase_1 := false;
3460:     phase_2 := true;
3461:     mycli := 0;
3462:     asm call [nmw_proc] end;
3463:     {
3464:       Speed := Speed3;
3465:       Sound_Schleifen := loop3;
3466:       set_timeconst_sbl6(Sampling_Rate);
3467:       Initialisiere_Vermischen;
3468:       Laufsec := 0;
3469:       Laufmin := 0;
3470:       wr_dsp_sbl6($D1);
3471:       Lade_Moddatei := 0;
3472:     end else begin;
3473:       Lade_Moddatei := -1;
3474:     end;
3475:   procedure ende_mod;
3476:   {
3477:     Beendet MOD-Ausgabe
3478:   }
3479:   var mlj : integer;
3480:   begin;
3481:     Free_Soundmem;
3482:     for mlj := 1 to 31 do begin;
3483:       freemem(Samp[mlj],Sam_1[mlj]);
3484:     end;
3485:     mod_terminated := true;
3486:   end;
3487:   Procedure periodisch_on;
3488:   {
3489:     Schaltet das Abspielen der MOD-Datei
3490:     ein.
3491:   }
3492:   Begin
3493:     init_sbperiod(@periodisch_off);
3494:     music_played := true;
3495:     dma_zzaehler := 0;
3496:     StelleTimerEin(@NeuerTimer,
3497:     timer_per_second);
3498:   End;
3499:   Procedure periodisch_off;
3500:   {
3501:     Schaltet das Abspielen der MOD-Datei
3502:     wieder aus
3503:   }
3504:   Begin
3505:     StelleTimerAus;
3506:   End;
3507:   {$F+}
3508:   procedure MODExitProc;
3509:   var mlj : byte;
3510:   begin
3511:     ExitProc := SaveExitProc;
3512:     if music_played then periodisch_off;
3513:     if not mod_terminated
3514:     and music_played
3515:     then ende_mod;
3516:     Exit_Sbl6;
3517:   end;
3518:   {$F-}
3519:   {$F+}
3520:   begin;
3521:     SaveExitProc := ExitProc;
3522:     ExitProc := @MODExitProc;
3523:     dsp_rdy_sbl6 := true;
3524:     dma_zzaehler := 0;
3525:     mod_terminated := false;

```



```

3526: music_played := false;
3527: mloop := true;
3528: songname := ' ';
3529: liedlaenge := 1;
3530: for mli := 1 to 31 do begin;
3531:   instnamen[mli] := ' ';
3532:   ' ';
3533:   Sam_l[mli] := 0;
3534:   Loop_s[mli] := 0;
3535:   Loop_l[mli] := 0;
3536: end;
3537: for mli := 1 to 31 do Sam_vol[mli] := 64;
3538: ' ';
3539: mli := 0;
3540: mlj := 0;
3541: tpw := 5;
3542: In_St1 := 0;
3543: In_St2 := 0;
3544: In_St3 := 0;
3545: In_St4 := 0;
3546: In_St5 := 0;
3547: In_St6 := 0;
3548: In_St7 := 0;
3549: In_St8 := 0;
3550: loop_pos := 0;
3551: modmultiply := 20;
3552: Sound_Schleifen := 10;
3553: Noten_Anschlag[1] := 0;
3554: Noten_Anschlag[2] := 0;
3555: Noten_Anschlag[3] := 0;
3556: Noten_Anschlag[4] := 0;
3557: Noten_Anschlag[5] := 0;
3558: Noten_Anschlag[6] := 0;
3559: Noten_Anschlag[7] := 0;
3560: Noten_Anschlag[8] := 0;
3561: end.

```

Listing 1. Die Unit »modplay.pas« gestattet die Ausgabe von MOD-Dateien während des Ablaufs von Applikationen.

Listing 2: mod_demo.pas

```

1: (
2:   Programm: MOD_DEMO
3:   Funktion: Demonstriert den Einsatz
4:   der Unit MODPLAY
5:   Sprache: Turbo Pascal ab 6.0
6:   Autor: Boris Bertelsens
7:   (c) 1993 DMV Widuch GmbH & Co. KG
8: )
9:
10: ($A+,B-,D+,E+,F-,G+,I-,L+,N-,O-,P-)
11: ($Q-,R-,S-,T-,V-,X+,Y+)
12: ($M 16384,0,655360)
13:
14: ( $DEFINE Miteffekten)
15:
16: uses crt,dos,modplay;
17:
18: type
19:   t = record
20:     c : char;
21:     a : byte;
22:   end;
23:
24: var gi : integer;
25:   my_modname : string;
26:   stapo,stinc : integer;
27:   ch : char;
28:   next_song : integer;
29:   mod1,mod2,mod3 : string;
30:   eposit : longint;
31:   effekt : Effect_type;
32:
33: procedure cursor_Off; assembler;
34: { Schaltet den sichtbaren Cursor aus
35: }
36:
37: asm
38:   xor ax,ax
39:   mov ah,01h
40:   mov cx,2020h
41:   int 10h
42: end;
43:
44: procedure cursor_on; assembler;
45: { Schaltet den Cursor wieder ein
46: }
47:
48: asm
49:   mov ah,01h
50:   mov cx,0607h
51:   int 10h
52: end;
53:
54: procedure writexy(x,y : integer;
55:   s : string);
56: begin;
57:   gotoxy(x,y);
58:   write(s);
59: end;
60:
61: procedure Scala_Kasten;
62: var li : integer;
63: begin;
64:   textcolor(3);
65:   textbackground(black);
66:   clrscr;

```

```

67:   write('TP MOD-Player Vers. 1.0 ',
68:     '(c) '93 by Boris Bertelsens & ',
69:     'DMV Widuch GmbH & Co. KG');
70:   textcolor(lightblue);
71:   for li := 1 to 10 do begin;
72:     gotoxy(2,li+4);
73:     write(li+2,' ',instnamen[li]);
74:     gotoxy(28,li+4);
75:     write(li+10+2,' ',instnamen[li+10]);
76:     gotoxy(54,li+4);
77:     write(li+20+2,' ',instnamen[li+20]);
78:   end;
79:   if Stimmen = 4 then begin;
80:     textcolor(black);
81:     textbackground(7);
82:     writexy(02,16,'=====');
83:     '=====';
84:     writexy(02,17,'||');
85:     '||';
86:     writexy(02,18,'||');
87:     '||';
88:     writexy(02,19,'||');
89:     '||';
90:     writexy(02,20,'||');
91:     '||';
92:     writexy(02,21,'=====');
93:     '=====';
94:     gotoxy(1,23);
95:     end else begin;
96:     textcolor(black);
97:     textbackground(7);
98:     writexy(02,16,'=====');
99:     '=====';
100:    writexy(02,17,'||');
101:    '||';
102:    writexy(02,18,'||');
103:    '||';
104:    writexy(02,19,'||');
105:    '||';
106:    writexy(02,20,'||');
107:    '||';
108:    writexy(02,21,'=====');
109:    '=====';
110:    writexy(02,22,'||');
111:    '||';
112:    writexy(02,23,'||');
113:    '||';
114:    writexy(02,24,'||');
115:    '||';
116:    writexy(02,25,'=====');
117:    '=====';
118:    gotoxy(1,23);
119:    end;
120:   end;
121:   procedure Scala;
122:   var li,lj : integer;
123:   screen := array[1..50,1..80] of t
124:   absolute $B800:$0000;
125:   secu : string[3];
126:   begin;
127:     textcolor(lightblue);
128:     textbackground(black);
129:     str(laufsec,secu);
130:     if laufsec < 10 then secu := '0'+secu;
131:     gotoxy(2,3);
132:     write('Songname: ');
133:     textcolor(lightcyan);
134:     write(Songname);
135:     gotoxy(34,3);
136:     textcolor(lightblue);
137:     write('Frequenz: ');
138:     textcolor(lightcyan);
139:     write(Sampling_Frequenz);
140:     textcolor(lightblue);
141:     write('KHz Laufzeit: ');
142:     textcolor(lightcyan);
143:     write(Laufmin:2,'/',31);
144:     gotoxy(2,4);
145:     textcolor(lightblue);
146:     write('Pattern No.: ');
147:     textcolor(lightcyan);
148:     write(Lied[mlj:3]);
149:     textcolor(lightblue);
150:     write('Lied[mlj:3]');
151:     textcolor(lightcyan);
152:     write('Pattern: ');
153:     textcolor(lightcyan);
154:     write(mlj:3,'/',Liedlaenge:3);
155:     textcolor(lightblue);
156:     write('Zeile: ');
157:     textcolor(lightcyan);
158:     write(mli:3);
159:     textcolor(lightblue);
160:     write('Geschwindigkeit: ');
161:     textcolor(lightcyan);
162:     write(Aktspeed:2,'/31');
163:     gotoxy(2,2);
164:     textcolor(lightblue);
165:     write('Memory Free: ');
166:     textcolor(lightcyan);
167:     write(Maxavail:6,' KB');
168:     textcolor(black);
169:     textbackground(7);
170:     if Stimmen = 4 then begin;
171:       if In_St1 < 0 then

```

```

187:       writexy(4,17,instnamen[In_St1]);
188:       if In_St2 < 0 then
189:         writexy(4,18,instnamen[In_St2]);
190:       if In_St3 < 0 then
191:         writexy(4,19,instnamen[In_St3]);
192:       if In_St4 < 0 then
193:         writexy(4,20,instnamen[In_St4]);
194:     end else begin;
195:       if In_St1 < 0 then
196:         writexy(4,17,instnamen[In_St1]);
197:       if In_St2 < 0 then
198:         writexy(4,18,instnamen[In_St2]);
199:       if In_St3 < 0 then
200:         writexy(4,19,instnamen[In_St3]);
201:       if In_St4 < 0 then
202:         writexy(4,20,instnamen[In_St4]);
203:       if In_St5 < 0 then
204:         writexy(4,21,instnamen[In_St5]);
205:       if In_St6 < 0 then
206:         writexy(4,22,instnamen[In_St6]);
207:       if In_St7 < 0 then
208:         writexy(4,23,instnamen[In_St7]);
209:       if In_St8 < 0 then
210:         writexy(4,24,instnamen[In_St8]);
211:     end;
212:   for lj := 1 to Stimmen do begin;
213:     for li := 1 to 16 do begin;
214:       if (Noten_Anschlag[lj] div 10) > li
215:       then begin;
216:         screen[16+lj,29+li].c := 'I';
217:         screen[16+lj,29+li].a := 114;
218:       end else begin;
219:         screen[16+lj,29+li].c := 'I';
220:         screen[16+lj,29+li].a := 112;
221:       end;
222:     end;
223:   for li := 16 to 32 do begin;
224:     if (Noten_Anschlag[lj] div 10) > li
225:     then begin;
226:       screen[16+lj,29+li].c := 'I';
227:       screen[16+lj,29+li].a := 126;
228:     end else begin;
229:       screen[16+lj,29+li].c := 'I';
230:       screen[16+lj,29+li].a := 112;
231:     end;
232:   end;
233:   for li := 33 to 48 do begin;
234:     if (Noten_Anschlag[lj] div 10) > li
235:     then begin;
236:       screen[16+lj,29+li].c := 'I';
237:       screen[16+lj,29+li].a := 116;
238:     end else begin;
239:       screen[16+lj,29+li].c := 'I';
240:       screen[16+lj,29+li].a := 112;
241:     end;
242:   end;
243: end;
244:
245: procedure init_The_Mod;
246: begin;
247:   mod_Setloopflag(ON);
248:   { Soll die Ausgabe wieder von vorne
249:   beginnen, wenn am Ende angelangt?}
250:   init_sb;
251:   { Speaker einschalten, automatische
252:   Erkennung der Soundkarte }
253:   clrcsr;
254:   mod_Samplefreq(22);
255: end;
256:
257: procedure Play_the_Mod(s : string);
258: var h : byte;
259: error : integer;
260: begin;
261:   mod_Samplefreq(22);
262:   error := lade_moddatei(s,AUTO,AUTO);
263:   if error < 0 then begin;
264:     clrcsr;
265:     writeln('Fehler beim Laden der ',
266:       'MOD-Datei !');
267:     if error = -1 then writeln(
268:       'Initialisierung fehlerhaft !');
269:     if error = -2 then writeln(
270:       'Datei nicht gefunden !');
271:     halt(0);
272:   end;
273:   end;
274:   mli := 0;
275:   mlj := 0;
276:   periodisch_on;
277: end;
278:
279: begin;
280:   clrcsr;
281:   cursor_off;
282:   if paramstr(1) = '' then begin;
283:     write('Name der MOD-Datei: ');
284:     readln(mod1);
285:   end else
286:     mod1 := paramstr(1);
287:   if pos('.',mod1) = 0 then mod1 :=
288:     mod1+'.mod';
289:   Init_The_Mod;
290:   Play_The_Mod(mod1);
291:   writeln('--- Tastendruck ---');
292:   repeat until keypressed; readkey;
293:   periodisch_off;
294:   ende_mod;
295:   if SbVersMaj < 4 then
296:     port(dsp_adr+$c) := 1;
297:   cursor_on;
298:   asm mov ax,0003; int 10h; end;
299: end.

```

Listing 2. »mod_demo.pas« veranschaulicht den Einsatz der Unit »modplay.pas«.

MOD-Dateien à la carte

Ein MOD-Editor wird dazu benutzt, MOD-Dateien anzulegen und zu bearbeiten. Derzeit sind zahlreiche, teilweise auch kommerzielle MOD-Editoren auf dem Markt verfügbar, wobei sich die verschiedenen Programme leider nicht nur durch eine Vielzahl von Funktionen, sondern ebenfalls durch diverse Dateiformate unterscheiden.

Zu den »Oldtimern« ist sicherlich MOD-Edit zu zählen. Dieser Editor erzeugt vierstimmige MOD-Dateien, die vollständig kompatibel zum vom Amiga stammenden Standard sind, und bietet darüber hinaus zahlreiche Fähigkeiten zur Soundausgabe. Leider finden sich bei seinen Funktionen nur einfaches Kopieren und manuelles Editieren.

Ein anderer bekannter Editor ist der 669-MOD-Editor von Renaissance aus den USA. Er bietet zwar acht Stimmen sowie gute Editiermerkmale, benutzt jedoch leider kein Standard-MOD-Format. Vielmehr ist es MOD-ähnlich aufgebaut, was die Nutzung durch die gegebenenfalls individuell erweiterbare Unit »modplay.pas« (in einem eigenen Beitrag in dieser Ausgabe behandelt) grundsätzlich erlaubt, hier jedoch aus Platzgründen entfallen muß. Das in diesem Beitrag enthaltene und vorgestellte Programm »editmod.pas« (Listing) wurde von Vision Factory, einer deutschen Demo-Crew, entwickelt.

Ein anderer Editor zur Erzeugung von sowohl vier- als auch achtstimmigen MODs ist Fastracker von Triton. Sein achtstimmiges Aufzeichnungsformat unterscheidet sich vom Standardformat lediglich in der Anzahl der Noten in einer Zeile (acht anstelle von vier). Dadurch wird ein Pattern 2048 und nicht 1024 Byte groß. Der im folgenden vorgestellte Vision-Factory-MOD-Editor benutzt ebenfalls dieses Format. Des weiteren besteht noch die Option, einzelne Patterns und Tracks zu speichern und zu laden; eine Fähigkeit, die die anderen Editoren leider vermissen lassen. »editmod.pas« ist auch in der Lage, die Noten eines Patterns nicht nur über die Tastatur, sondern auch über MIDI einzugeben, was vor allem Musikern zugute kommt.

Die programmtechnische Struktur des MOD-Editors im Listing ist relativ einfach. Nach dem Initialisierungsprozeß prüft eine Schleife auf Tastatur- und MIDI-Eingaben. Da es sich hierbei um grundlegende Pascal-Programmierung handelt, werden wir nicht weiter darauf eingehen. Von Interesse dürfte vor allem die Handhabung des Programms sein. Die zentrale Steuerung übernehmen die Funktionstasten

sowie die Tabulatortaste. Mit der Tabulatortaste springen Sie zwischen den einzelnen Bereichen hin und her ([Tab] für vorwärts und [Umschalt-Tab] für einen Sprung zurück). Bei Problemen gibt Ihnen das integrierte Hilfesystem durch Druck auf die Taste [F1] Auskunft.

Die Tastaturbelegung

● [F1] – Hilfesystem aufrufen

Die Taste [F1] ruft das Hilfesystem auf. Dieses gibt Ihnen kontextbezogenen Auskunft über die jeweils verwendbaren Bearbeitungsfunktionen sowie die unterstützten Tastenkombinationen.

● [F2] – Select

Je nachdem, an welcher Stelle sich das Programm gerade befindet, können Sie durch Drücken der Taste [F2] die gewünschte Note, einen Effekt oder ein Instrument auswählen. Die Pfeiltasten dienen zur Positionierung auf den beabsichtigten Eintrag. Des weiteren stehen Ihnen die Tasten [Pos1] und [Ende] zur schnelleren Bewegung in der Liste zur Verfügung.

● [F3] – MIDI On

Mit Betätigung der Taste [F3] schalten Sie den MIDI-Mode ein. Die vom MIDI-Instrument eintreffenden Signale werden als Noten interpretiert und in das Pattern eingetragen. Alle Tastaturfunktionen sind weiterhin aktiv, nur ein Abspielen des Songs/Patterns sollte man vermeiden, da der Soundblaster nur entweder MIDI-Informationen oder aber Sampling-Daten verarbeiten kann. Verwechseln Sie den MIDI-Mode allerdings nicht mit dem MIDI-Recording (siehe [F9]).

● [F4] – MIDI Off

Die Taste [F4] schaltet den über [F3] eingeschalteten MIDI-Modus wieder aus. Dem Ab-

spielen des Songs beziehungsweise Patterns steht nun wieder nichts im Wege.

● [F5] – Load Instrument

Im Instrumenten-Auswahlmenü stehen die Pfeiltasten zur Bewegung in der Liste zur Verfügung. [Pos1] und [Ende] springen zum Anfang oder Ende der dargestellten Instrumente. [Enter] bestätigt schließlich Ihre Auswahl, [Esc] verläßt die Liste ohne Änderung. Um ein Instrument von Festplatte zu laden, drücken Sie die Taste [Einf]. Sie können nun entweder direkt einen Dateinamen eingeben oder gelangen mit [Enter] zur Dateiauswahl. Auch beim Laden des gewünschten externen Instrumentes bestätigt [Enter] Ihre Eingabe. Um ein Instrument wieder zu entfernen, drücken Sie lediglich die Taste [Entf].

● [F6] – Save Instrument

Dieses Menü wird zum Speichern von Instrumenten eingesetzt. Dies ist vor allem dann sinnvoll, wenn man ein Instrument aus einer MOD-Datei in eine andere portieren möchte. Wiederum dienen Pfeiltasten zur Positionierung. Bestätigen Sie Ihre Auswahl mit [Enter] oder brechen Sie mit der Taste [Esc] ab.

● [F7] – Play Pattern

Diese Taste aktiviert das Abspielen eines Patterns. Dabei wird immer das gerade in Bearbeitung befindliche Pattern gespielt. Stellen Sie jedoch sicher, daß genügend freier Hauptspeicher vorhanden ist. Auf einem normalen 386- und 486-System dürfte es allerdings nur bei sehr großen MOD-Dateien unter Umständen zu Problemen kommen.

● [F8] – Play Song

Mit Betätigung der Taste [F8] aktivieren Sie das Abspielen der MOD-Datei ab der aktuellen Position. Entscheidend ist dabei die im Song-Feld eingestellte Position.

● [F9] – MIDI-Recording

Die Taste [F9] startet das MIDI-Recording. Diese Funktion dient aber nicht der Aufzeichnung von MIDI-Daten, vielmehr wird das aktuelle Pattern in einer wählbaren Geschwindigkeit aktiviert. Die gespielten MIDI-Noten zeichnet das Programm

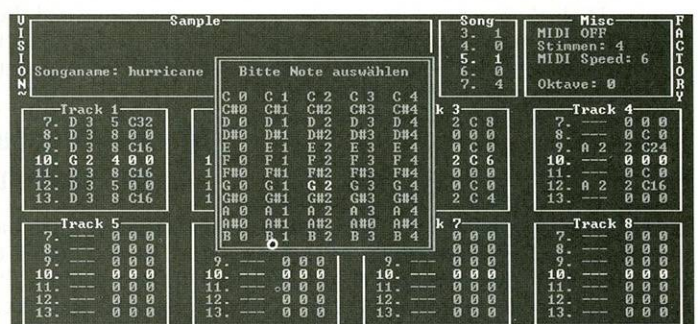


Bild 1. Der grundsätzliche Bildschirmaufbau von »editmod.pas«

als MOD-Noten im jeweils aktuellen Track auf. Die normale Tastaturbelegung bleibt während der Aufnahme gesperrt, die Aufnahme selbst läßt sich durch Tastendruck abbrechen.

● [Umschalt-F1] – Load Song

Im Load-Song-Menü bestimmen Sie eine beliebige MOD-Datei, die Sie laden möchten. Die Pfeiltasten übernehmen die Positionierung im Menü; des weiteren sind [Pos1] und [Ende] verfügbar. Ihre Auswahl bestätigen Sie mit der Enter-Taste oder Sie brechen mit [Esc] ohne Änderung der Einstellungen ab. Um eine nicht im aktuellen Verzeichnis befindliche Datei zu laden, drücken Sie die Taste [Einf]. Geben Sie in der erscheinenden Dialogbox den kompletten Pfad und Namen der zu ladenden Datei ein und bestätigen Sie mit [Enter].

● [Umschalt-F2] – Save Song

Im Save-Song-Menü läßt sich der Dateiname auswählen, unter dem Ihre aktuell im Speicher befindliche MOD-Datei aufgezeichnet werden soll. Zur Eingabe eines neuen Namens dient die Taste [Einf]. In der erscheinenden Dialogbox geben Sie dann den gewünschten Dateinamen gegebenenfalls mit Pfadangabe an.

● [Umschalt-F3] – Load Pattern

Das Load-Pattern-Menü bestimmt das zu ladende Pattern. Benutzen Sie zur Bewegung in der Liste die Pfeiltasten, bestätigen Sie mit [Enter] oder brechen Sie durch [Esc] ab. Das zu ladende Pattern wird allerdings nicht direkt in das aktuelle Pattern, sondern in den Pattern-Edit-Puffer übertragen. Die Tastenkombination [Umschalt-F8] fügt den Pufferinhalt im aktuellen Pattern ein. Von »editmod.pas« erzeugte Pattern-Dateien verwenden die Dateiendung »ptn«.

● [Umschalt-F4] – Save Pattern

Das Save-Pattern-Menü kümmert sich um die Festlegung des Dateinamens, unter dem Sie das momentan im Pattern-Edit-Puffer befindliche Pattern speichern möchten. Um einen neuen Namen einzugeben, drücken Sie die Taste [Einf]. In der folgenden Dialogbox geben Sie dann den gewünschten Dateinamen, wenn erforderlich mit Pfadangabe, an. Ein Pattern wird über die Tastenkombination [Umschalt-F7] in den Pattern-Edit-Puffer übertragen.

● [Umschalt-F5] – Load Track

In diesem Menü wählen Sie den Track aus, der in den Track-Edit-Puffer gelangen soll. Pfeiltasten und [Enter] bestätigen die Auswahl, [Esc] bricht ohne Korrektur des Pufferinhaltes ab. Die Track-Dateien von »editmod.pas« benutzen dabei die Endung »trk«. Einen Track fügen Sie mit der Ta-

stenkombination [Umschalt-F10] an der gewünschten Stelle ein.

● [Umschalt-F6] – Save Track

Im Save-Track-Menü legen Sie den Dateinamen des zu speichernden Tracks fest. Zur Bewegung in dieser Liste sowie Auswahl oder Abbruch verwenden Sie die bereits in anderen Auswahlmenüs vorgestellten Tasten. Dort nicht vorhandene Dateinamen gibt man nach Druck

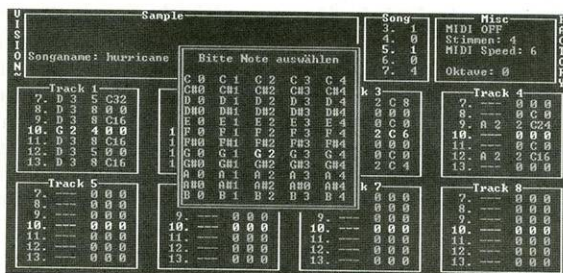


Bild 2. Noten können Sie per Tastatur, via MIDI oder anhand einer Auswahlbox eingeben.

auf die Taste [Einf] ein. Das Programm speichert immer den im Track-Edit-Puffer befindlichen Track. Um einen Track in den Track-Edit-Puffer zu kopieren, drücken Sie die Tastenkombination [Umschalt-F9].

● [Umschalt-F7] – Copy Pattern

Die Tastenkombination [Umschalt-F7] kopiert das momentan in Bearbeitung befindliche Pattern in den Pattern-Edit-Puffer. Von dort aus läßt es sich an anderer Stelle wieder einfügen oder auch speichern.

● [Umschalt-F8] – Paste Pattern

Hiermit fügen Sie das momentan im Pattern-Edit-Puffer befindliche Pattern in das aktuelle Pattern ein, das dabei überschrieben wird. Um nur Teile des Patterns zu überschreiben, verwendet man die Track-Funktionen.

● [Umschalt-F9] – Copy Track

Diese Funktion kopiert den momentan in Bearbeitung befindlichen Track in den Track-Edit-Puffer. Sie können ihn von dort an anderer Stelle einfügen oder speichern.

● [Umschalt-F10] – Paste Track

Diese Tastenkombination fügt den momentan im Track-Edit-Puffer befindlichen Track in den aktuellen Track ein. Der aktuelle Track wird dabei überschrieben.

Bildschirm Aufbau und Programmfenster

Der Bildschirm Aufbau geht aus Bild 1 hervor. In der linken oberen Ecke befindet sich der Sample-Bereich, daneben das Song- und ganz rechts das Misc-Fenster. Darunter liegen in zwei Reihen zu je vier Ausschnitten die Bildschirmbereiche für

die einzelnen Tracks. Zwischen den Fenstern schalten Sie mit den Tasten [Tab] und [Umschalt-Tab] um.

Das Sample-Fenster

Im Sample-Fenster können Sie die Parameter des jeweils aktiven Samples verändern. Drücken Sie dabei die Taste [N], um den Namen des Samples zu vergeben. Die maximale Namenslänge beträgt 22 Zeichen. Die Taste [S] kümmert sich darum, den Start der Sample-Loop einzugeben. Der Start darf logischerweise nicht größer als *Samplelänge* – *Looplänge* ausfallen und wird deshalb vom Programm gegebenenfalls automatisch korrigiert. Um die Loop-Länge einzugeben, drückt man die Taste [L]. Zu hohe Werte korrigiert das Programm wiederum automatisch. Nach Betätigen der Taste [G] läßt sich ein Song-Name vergeben. Die maximale Länge des Namens beträgt 20 Zeichen.

Das Song-Fenster

Im Song-Fenster bestimmen Sie das Arrangement des Songs. Die Zahlen stehen dabei für die Nummer des jeweils zu spielenden Patterns. Benutzen Sie die Pfeiltasten, um sich im Arrangement von oben nach unten zu bewegen. Die Taste [Einf] erzeugt einen neuen Track. Zur Eingabe der Nummer des Patterns dienen die Zahlentasten. Maximal sind 64 Patterns zulässig.

Das Misc-Fenster

Das Misc-Fenster besitzt verschiedene Aufgaben. In ihm finden Sie die Geschwindigkeit, mit der beim MIDI-Recording das Pattern abgespielt wird. Über die Taste [M] aktiviert man einen neuen Wert, der zwischen 1 und 12 liegen darf. [+] und [-] bewirken eine Veränderung der aktiven Oktave. Die Taste [S] bestimmt die Anzahl der Stimmen, die Ihre MOD-Datei verwenden soll (zulässige Werte sind 4 und 8). Bitte beachten Sie, daß bei achtstimmigen MOD-Dateien zwar mehr Stimmen zur Verfügung stehen, die Berechnung dann aber auch fast doppelt so lange dauert. Dies ist wichtig für die Programmierung zeitkritischer Routinen, zum Beispiel bei Sound-unterstützten Animationen, wo vier Stimmen, sofern realisierbar, vorzuziehen sind.

Die Track-Fenster

In den Track-Fenstern geben Sie die Noten des Pattern zusammen mit seinen Instrumenten ein. Zur Bewegung im Fenster dienen die Pfeiltasten, [Pos1], [Ende] sowie [Bild auf] und [Bild ab]. Das Track-

Fenster ist folgendermaßen aufgebaut: Nach der Zeilennummer folgt der Notenwert. Zur Eingabe der Notenwerte läßt sich neben dem MIDI-Recording die Tastatur benutzen. Dabei sind Q=C, 2=Cis, W=D und so weiter. Außerdem ist die gewünschte Note aus einem über [F2] aufrufbaren Auswahlménü (Bild 2) wählbar. Nach der Note folgt die Instrumentennummer, die zwischen 1 und 31 für aktive Instrumente liegen darf. Geben Sie die Nummer entweder direkt ein oder wählen Sie das gewünschte Instrument über [F2]. An die Instrumentennummer schließt sich die Effekt-Nummer an. Sie darf zwischen 0 für keinen Effekt und hexadezimal F liegen. Auch sie gibt man entweder direkt ein oder wählt sie über [F2] aus. An letzter Stelle stehen die Operanden des Effekts. Die Anzahl der Operanden variiert zwischen 0 und 2. Näheres hierzu finden Sie im Beitrag über MOD-Dateien in dieser Ausgabe.

Anhand der Beschreibung sind Sie nun in der Lage, problemlos selbst MOD-Dateien zu erzeugen. Zum Abspielen können Sie neben den bekannten MOD-Playern wie »Dmp« oder »Wowii« die Unit »modplay« benutzen.

(Boris Bertelons/hs/bj)

Listing: editmod.pas

```
1: {Programm: editmod.pas
2: Funktion: MOD-Editor
3: Sprache : Turbo Pascal ab 5.0
4: Autor : Boris Bertelons
5: (c) 1993 DMV Daten- u. Medienverlag)
6:
7: {$A+,B-,D+,E+,F-,G+,I-,L+,N-,O-,P-,}
8: {$Q-,R-,S-,T-,V-,X+,Y+}
9: {$M 16384,0,655360}
10: uses crt,dos,modplay,WinDos;
11: const editmodname = 'hurrica2.mod';
12: MIDI : boolean = false;
13: STEPMODE : boolean = true;
14: activ_inst : byte = 1;
15: steps : word = 0;
16: steplimit : word = 6;
17: highest_track : byte = 0;
18: Pattern_edited : boolean = false;
19: Screen_Akt : byte = 1;
20: Note : integer = 1;
21: Instrument : integer = 2;
22: Effekttyp : integer = 3;
23: Effektopp : integer = 4;
24: edoff : array[1..6] of byte =
25: (8,12,13,15,16,17);
26: trackanzahl : byte = 8;
27: edit_zeile : integer = -2;
28: edit_track : byte = 1;
29: Edit_value : byte = Note;
30: Edit_Oktave : byte = 0;
31: Edit_Arrang : integer = 1;
32: Aktual_Screen : byte = 1;
33: song_loaded : boolean = false;
34: notenvalues : array[0..4,1..12]
35: of word =
36: (
37: (1712,1616,1525,1440,1359,1283,1211,
38: 1143, 1078,1018, 961, 907),
39: (856, 808, 763, 720, 679, 641, 605,
40: 571, 539, 509, 480, 453),
41: (428, 404, 381, 360, 340, 321, 303,
42: 286, 270, 254, 240, 227),
43: (214, 202, 191, 180, 170, 160, 151,
44: 143, 135, 127, 120, 113),
45: (107, 101, 95, 90, 85, 80, 76,
46: 71, 67, 64, 60, 57)
47: );
48: Notennamen : array[1..12]
49: of string[2] =
50: ('C', 'C#', 'D', 'D#', 'E', 'F', 'F#',
51: 'G', 'G#', 'A', 'A#', 'B');
52: var actual_Pattern :
53: array[1..64,1..8,1..4] of byte;
54: Puffer_Pattern :
55: array[1..64,1..8,1..4] of byte;
56: Puffer_Track :
57: array[1..64,1..4] of byte;
58: procedure cursor_Off; assembler;
59: asm
60: xor ax,ax
61: mov ah,01h
62: mov cx,2020h
63: int 10h
```

```
64: end;
65: procedure cursor_on; assembler;
66: asm
67: mov ah,01h
68: mov cx,0607h
69: int 10h
70: end;
71: procedure writexy(x,y : integer;
72: s : string);
73: begin;
74: gotoxy(x,y);
75: write(s);
76: end;
77: procedure locate_cursor;
78: begin;
79: if edit_track < 5 then
80: gotoxy(edoff[edit_value]+
81: (edit_track-1)*20,12)
82: else
83: gotoxy(edoff[edit_value]+
84: (edit_track-5)*20,21);
85: end;
86: procedure rahmen(rt: byte;startx,
87: starty,dx,dy : integer);
88: const frames : array[1..2,1..6]
89: of char =
90: ((' ',' ',' ',' ',' ',' '),
91: ('F',' ',' ',' ',' ',' '));
92: var lx,ly : integer;
93: s : string;
94: begin;
95: { obere Zeile }
96: s := frames[rt,1];
97: for lx := 1 to dx-2 do s := s +
98: frames[rt,5];
99: s := s + frames[rt,2];
100: gotoxy(startx,starty);
101: write(s);
102: { mittleren Zeilen }
103: for ly := 1 to dy-2 do begin;
104: s := frames[rt,6];
105: for lx := 1 to dx-2 do s:=s+' ';
106: s := s + frames[rt,6];
107: gotoxy(startx,starty+ly);
108: write(s);
109: end;
110: { untere Zeile }
111: s := frames[rt,4];
112: for lx := 1 to dx-2 do s := s +
113: frames[rt,5];
114: s := s + frames[rt,3];
115: gotoxy(startx,starty+dy-1);
116: write(s);
117: end;
118: procedure write_oktave;
119: begin;
120: textcolor(LightGray);
121: gotoxy(63,6);
122: write('Oktave: ',edit_oktave);
123: locate_cursor;
124: end;
125: procedure write_Arrangement(start :
126: integer);
127: var zeilen : byte;
128: begin;
129: if start > liedlaenge then
130: liedlaenge := start;
131: textcolor(LightGray);
132: dec(start,2);
133: zeilen := 0;
134: while (start <=130) and (zeilen <5)
135: do begin;
136: if (start > 0) and (start <= 128)
137: then begin;
138: gotoxy(52,2+zeilen);
139: write(start,3,' ');
140: write(lied[start],2);
141: end else
142: writexy(52,2+zeilen,' ');
143: inc(start);
144: inc(zeilen);
145: end;
146: textcolor(14);
147: dec(start,3);
148: gotoxy(52,4);
149: write(start,3,' ');
150: write(lied[start],2);
151: end;
152: procedure downwrite(x,y : integer;
153: s : string);
154: var li : integer;
155: begin;
156: for li := 1 to length(s) do begin;
157: writexy(x,y+li-1,s[li]);
158: end;
159: end;
160: procedure redraw_screen;
161: begin;
162: textbackground(1);
163: textcolor(white);
164: clrscr;
165: rahmen(1,2,8,18,9);
166: rahmen(1,22,8,18,9);
167: rahmen(1,42,8,18,9);
168: rahmen(1,62,8,18,9);
169: rahmen(1,2,17,18,9);
170: rahmen(1,22,17,18,9);
171: rahmen(1,42,17,18,9);
172: rahmen(1,62,17,18,9);
173: rahmen(1,3,1,48,7);
174: rahmen(1,51,1,10,7);
175: rahmen(1,61,1,18,7);
176: downwrite(2,1,'VISION-');
177: downwrite(79,1,'FACTORY');
178: writexy(54,1,'Song');
179: writexy(7,8,'Track 1');
180: writexy(27,8,'Track 2');
181: writexy(47,8,'Track 3');
182: writexy(67,8,'Track 4');
183: writexy(7,17,'Track 5');
```

```
184: writexy(27,17,'Track 6');
185: writexy(47,17,'Track 7');
186: writexy(67,17,'Track 8');
187: writexy(4,5,' '+
188: ' ');
189: gotoxy(4,5);
190: textcolor(LightGray);
191: write('Songname: ',Songname);
192: writexy(63,2,'MIDI OFF');
193: gotoxy(63,3);
194: write('Stimmen: ',stimmen);
195: gotoxy(63,4);
196: write('MIDI Speed: ',steplimit);
197: write_Oktave;
198: locate_cursor;
199: end;
200: procedure clear_actual_Pattern;
201: begin;
202: fillchar(actual_Pattern,2048,0);
203: end;
204: procedure clear_arrangement;
205: begin;
206: fillchar(Lied,128,0);
207: end;
208: procedure Tonhoehe_in_Pattern(zeile,
209: track : byte;Wert : word);
210: var hibits,lobits : byte;
211: begin;
212: if Wert <> 9999 then begin;
213: Hibits := hi(Wert);
214: Lobits := lo(Wert);
215: Hibits := Hibits AND $0F;
216: actual_Pattern[zeile,track,1] :=
217: actual_Pattern[zeile,track,1]
218: AND $F0;
219: actual_Pattern[zeile,track,1] :=
220: actual_Pattern[zeile,track,1] +
221: Hibits;
222: actual_Pattern[zeile,track,2] :=
223: lobits;
224: end;
225: end;
226: procedure Instrument_in_Pattern(
227: zeile,track : byte;Wert : byte);
228: var hbyte,lobyte : byte;
229: begin;
230: if Wert <> 255 then begin;
231: hbyte := Wert AND $F0;
232: lobyte := Wert AND $0F;
233: lobyte := lobyte shl 4;
234: actual_Pattern[zeile,track,1] :=
235: actual_Pattern[zeile,track,1]
236: AND $0F;
237: actual_Pattern[zeile,track,1] :=
238: actual_Pattern[zeile,track,1] +
239: hbyte;
240: actual_Pattern[zeile,track,3] :=
241: actual_Pattern[zeile,track,3]
242: AND $0F;
243: actual_Pattern[zeile,track,3] :=
244: actual_Pattern[zeile,track,3] +
245: lobyte;
246: end;
247: end;
248: procedure Effekt_in_Pattern(zeile,
249: track : byte;Wert : byte);
250: begin;
251: Wert := Wert AND $0F;
252: actual_Pattern[zeile,track,3] :=
253: actual_Pattern[zeile,track,3]
254: AND $F0;
255: actual_Pattern[zeile,track,3] :=
256: actual_Pattern[zeile,track,3] +
257: Wert;
258: end;
259: procedure Oppoperanden_in_Pattern(
260: zeile,track,Wert : byte);
261: begin;
262: actual_Pattern[zeile,track,
263: 4] := Wert;
264: end;
265: procedure Inst_Zehner(zeile,track,
266: Wert : byte);
267: var altinst,help : byte;
268: begin;
269: altinst := actual_Pattern[zeile,
270: track,1]
271: AND $F0;
272: altinst := altinst+(actual_Pattern
273: [zeile,track,3] AND $F0) SHR 4);
274: altinst := altinst MOD 10;
275: altinst := Wert*10+altinst;
276: help := actual_Pattern[zeile,track,
277: 1] AND $0F;
278: help := help + (altinst AND $F0);
279: actual_Pattern[zeile,track,1] :=
280: help;
281: help := actual_Pattern[zeile,track,
282: 3] AND $0F;
283: help := help + (altinst shl 4);
284: actual_Pattern[zeile,track,3] :=
285: help;
286: end;
287: procedure Inst_Einer(zeile,track,
288: Wert : byte);
289: var altinst,help,hbyte,lbyte : byte;
290: begin;
291: { Alten Wert auslesen }
292: altinst := actual_Pattern[zeile,
293: track,1] AND $F0;
294: altinst := altinst+(actual_Pattern
295: [zeile,track,3] AND $F0) SHR 4);
296: { Neuen Wert einmaskieren }
297: help := altinst MOD 10;
298: altinst := altinst - help;
```

»editmod.pas« kann unter anderem einzelne Patterns sowie Tracks speichern und laden.


```

299: altinst := altinst + Wert;
300: { Wert zurück schreiben }
301: help := actual_Pattern[zeile,track,
302: 1] and $0F;
303: help := help + (altinst and $F0);
304: actual_Pattern[zeile,track,1] :=
305: help;
306: help := actual_Pattern[zeile,track,
307: 3] and $0F;
308: help := help + (altinst shl 4);
309: actual_Pattern[zeile,track,3] :=
310: help;
311: end;
312: procedure Opp_Zehner(zeile,track,
313: Wert : byte);
314: var altinst,help : byte;
315: begin;
316: altinst := actual_Pattern[zeile,
317: track,4];
318: altinst := altinst MOD 10;
319: altinst := Wert*10+altinst;
320: actual_Pattern[zeile,track,4] :=
321: altinst;
322: end;
323: procedure Opp_Einer(zeile,track,
324: Wert : byte);
325: var altinst,help,hbyte,lbyte : byte;
326: begin;
327: { Alten Wert auslesen }
328: altinst :=
329: actual_Pattern[zeile,track,4];
330: { Neuen Wert einmaskieren }
331: help := altinst MOD 10;
332: altinst := altinst - help;
333: altinst := altinst + Wert;
334: { Wert zurück schreiben }
335: actual_Pattern[zeile,track,4] :=
336: altinst;
337: end;
338: function Notennamen(val:word) :string;
339: var Oktave,Note : byte;
340: gefunden : boolean;
341: hs : string;
342: begin;
343: hs := '---';
344: gefunden := false;
345: Oktave := 0;
346: Note := 1;
347: while (Oktave < 5) and not gefunden
348: do begin;
349: if val > Notenvalues[Oktave,Note]
350: then begin;
351: str(Oktave,hs);
352: if Note = 1 then Note := 13;
353: hs := Notennamen[Note-1]+hs+' ';
354: gefunden := true;
355: end;
356: inc(Note);
357: if Note > 12 then begin;
358: inc(Oktave);
359: Note := 1;
360: end;
361: end;
362: Notennamen := hs;
363: end;
364: procedure write_Pattern(start :
365: integer);
366: const hexcar : array[0..15] of char =
367: ('0','1','2','3','4','5','6','7',
368: '8','9','A','B','C','D','E','F');
369: var zeilen,tracknr : byte;
370: n,s,z,ef,op : string;
371: tonhoehe : word;
372: instrum : byte;
373: effekt : byte;
374: operanden : word;
375: li : integer;
376: begin;
377: zeilen := 0;
378: textcolor(white);
379: writexy(20,1,'Sample');
380: writexy(67,1,'Misc');
381: writexy(54,1,'Song');
382: writexy(7,8,'Track 1');
383: writexy(27,8,'Track 2');
384: writexy(47,8,'Track 3');
385: writexy(67,8,'Track 4');
386: writexy(7,17,'Track 5');
387: writexy(27,17,'Track 6');
388: writexy(47,17,'Track 7');
389: writexy(67,17,'Track 8');
390: textcolor(10);
391: case edit_track of
392: 1: begin;
393: writexy(7,8,'Track 1');
394: end;
395: 2: begin;
396: writexy(27,8,'Track 2');
397: end;
398: 3: begin;
399: writexy(47,8,'Track 3');
400: end;
401: 4: begin;
402: writexy(67,8,'Track 4');
403: end;
404: 5: begin;
405: writexy(7,17,'Track 5');
406: end;
407: 6: begin;
408: writexy(27,17,'Track 6');
409: end;
410: 7: begin;
411: writexy(47,17,'Track 7');
412: end;
413: 8: begin;
414: writexy(67,17,'Track 8');
415: end;
416: end;
417: while (start<=67) and (zeilen<7) do
418: begin;

```

```

419: if zeilen = 3 then
420: textcolor(14) else
421: textcolor(7);
422: for tracknr := 1 to 4 do begin;
423: if (start > 0) and (start <= 64)
424: then begin;
425: tonhoehe := (actual_Pattern[
426: start,tracknr,1] and $0F)*256 +
427: actual_Pattern[start,tracknr,2];
428: instrum := (actual_Pattern[start,
429: tracknr,1] and $F0) +
430: ((actual_Pattern[start,tracknr,3]
431: and $F0) shr 4);
432: effekt := actual_Pattern[start,
433: tracknr,3] and $0F;
434: operanden := actual_Pattern[
435: start,tracknr,4];
436: if tonhoehe = 0 then n := '---'
437: else n := Notennamen(tonhoehe);
438: str(instrum,2,s);
439: str(start-1,2,z);
440: ef := hexcar[effekt];
441: str(operanden,2,op);
442: writexy(4+(tracknr-1)*20,
443: 9+zeilen,
444: z+'.'+n+s+'.'+ef+op);
445: end else begin;
446: writexy(4+(tracknr-1)*20,
447: 9+zeilen,
448: '');
449: end;
450: end;
451: for tracknr := 5 to Trackanzahl do
452: begin;
453: if (start>0) and (start<=64) then
454: begin;
455: tonhoehe := (actual_Pattern[
456: start,tracknr,1] and $0F) shl 8+
457: actual_Pattern[start,tracknr,2];
458: instrum := (actual_Pattern[
459: start,tracknr,1] and $F0) +
460: ((actual_Pattern[start,tracknr,3]
461: and $F0) shr 4);
462: effekt := actual_Pattern[start,
463: tracknr,3] and $0F;
464: operanden := actual_Pattern[
465: start,tracknr,4];
466: if tonhoehe = 0 then n := '---'
467: else n := Notennamen(tonhoehe);
468: str(instrum,2,s);
469: str(start-1,2,z);
470: ef := hexcar[effekt];
471: str(operanden,2,op);
472: writexy(4+(tracknr-5)*20,
473: 18+zeilen,
474: z+'.'+n+s+'.'+ef+op);
475: end else begin;
476: writexy(4+(tracknr-5)*20,
477: 18+zeilen,
478: '');
479: end;
480: end;
481: inc(start);
482: inc(zeilen);
483: end;
484: locate_cursor;
485: end;
486: procedure delete_in_pattern(start :
487: integer);
488: var li : integer;
489: help : array[1..4] of byte;
490: begin;
491: for li := start to 63 do begin;
492: help[1] := actual_Pattern[li+1,
493: edit_track,1];
494: help[2] := actual_Pattern[li+1,
495: edit_track,2];
496: help[3] := actual_Pattern[li+1,
497: edit_track,3];
498: help[4] := actual_Pattern[li+1,
499: edit_track,4];
500: actual_Pattern[li,edit_track,1] :=
501: help[1];
502: actual_Pattern[li,edit_track,2] :=
503: help[2];
504: actual_Pattern[li,edit_track,3] :=
505: help[3];
506: actual_Pattern[li,edit_track,4] :=
507: help[4];
508: end;
509: actual_Pattern[64,edit_track,1]:= 0;
510: actual_Pattern[64,edit_track,2]:= 0;
511: actual_Pattern[64,edit_track,3]:= 0;
512: actual_Pattern[64,edit_track,4]:= 0;
513: end;
514: procedure insert_in_pattern(start :
515: integer);
516: var li : integer;
517: help : array[1..4] of byte;
518: begin;
519: for li := 63 downto start do begin;
520: help[1] := actual_Pattern[li,
521: edit_track,1];
522: help[2] := actual_Pattern[li,
523: edit_track,2];
524: help[3] := actual_Pattern[li,
525: edit_track,3];
526: help[4] := actual_Pattern[li,
527: edit_track,4];
528: actual_Pattern[li+1,edit_track,1]:=
529: help[1];
530: actual_Pattern[li+1,edit_track,2]:=
531: help[2];
532: actual_Pattern[li+1,edit_track,3]:=
533: help[3];
534: actual_Pattern[li+1,edit_track,4]:=
535: help[4];
536: end;
537: actual_Pattern[start,edit_track,1]:=
538: 0;

```

```

539: actual_Pattern[start,edit_track,2]:=
540: 0;
541: actual_Pattern[start,edit_track,3]:=
542: 0;
543: actual_Pattern[start,edit_track,4]:=
544: 0;
545: end;
546: procedure save_screen;
547: var Screen : array[1..4,1..4000]
548: of byte;
549: absolute $b800:0000;
550: begin;
551: if Screen_Akt <= 4 then begin;
552: inc(Screen_Akt);
553: move(screen[1],screen[
554: Screen_Akt],4000);
555: end;
556: end;
557: procedure restore_screen;
558: var Screen : array[1..4,1..4000]
559: of byte;
560: absolute $b800:0000;
561: begin;
562: if Screen_Akt >= 2 then begin;
563: move(screen[Screen_Akt],
564: screen[1],4000);
565: dec(Screen_Akt);
566: end;
567: end;
568: function select_datei(mask : pchar;
569: mtext: string) : string;
570: const zeile : byte = 1;
571: spalte : byte = 0;
572: var
573: DirInfo: TSearchRec;
574: filenames : array[1..60]
575: of string[12];
576: count : integer;
577: Nullpos : byte;
578: var li,lj : integer;
579: inp : char;
580: retval : string;
581: begin
582: {$I+}
583: for li:=1 to 60 do filenames[li] :=
584: '---';
585: count := 1;
586: FindFirst(mask, faArchive,DirInfo);
587: while DosError = 0 do
588: begin
589: filenames[count] := (DirInfo.Name);
590: Nullpos:= pos(#0,filenames[count]);
591: if Nullpos < 0 then
592: filenames[count]:= copy(filenames
593: [count],0,Nullpos-1);
594: inc(count);
595: FindNext(DirInfo);
596: end;
597: {$I-}
598: save_screen;
599: rahmen(2,5,4,72,16);
600: writexy(21,5,
601: 'Bitte Datei auswählen!');
602: repeat
603: textcolor(LightGray);
604: for lj := 0 to 4 do begin;
605: for li := 1 to 12 do begin;
606: writexy(7+lj*14,5+li,
607: filenames[lj*12+li]);
608: end;
609: end;
610: textcolor(14);
611: writexy(7+Spalte*14,5+Zeile,
612: filenames[Spalte*12+Zeile]);
613: inp := readkey;
614: if ord(inp) = 0 then inp :=
615: readkey;
616: if ord(inp) = 0 then inp :=
617: readkey;
618: case ord(inp) of
619: 13: begin;
620: retval :=
621: filenames[Spalte*
622: 12+Zeile];
623: end;
624: 27: begin;
625: retval := 'xxxx';
626: end;
627: 71: begin;
628: { Pos 1 }
629: Zeile := 1;
630: Spalte := 0;
631: end;
632: 72: begin;
633: { Pfeil up }
634: if not ((Zeile = 1) and
635: (Spalte = 0)) then
636: dec(Zeile);
637: if Zeile = 0 then begin;
638: dec(Spalte);
639: Zeile := 12;
640: end;
641: end;
642: 75: begin;
643: { Pfeil links }
644: if Spalte > 0 then
645: dec(Spalte);
646: end;
647: 77: begin;
648: { Pfeil rechts }
649: if (Spalte < 3) and (Zeile+
650: (Spalte+1)*12 < count) then
651: inc(Spalte);
652: end;
653: 79: begin;

```

»editmod.pas« kann unter anderem einzelne Patterns sowie Tracks speichern und laden.


```

655: { End ,
        Spalte := count div 12;
656: Zeile := count-Spalte*12-1;
657: end;
658:
659: 80: begin;
660: if not ((Zeile = 12) and
661: (Spalte = 4)) and
662: (Zeile+Spalte*12 <count-1)
663: then
664: inc(Zeile);
665: if Zeile > 12 then begin;
666: inc(Spalte);
667: Zeile := 1;
668: end;
669:
670: 82 : begin;
671: save_screen;
672: rahmen(2,16,9,45,5);
673: writexy(20,10,'Dateinamen'+
674: ' eingeben ('+mtext;');');
675: writexy(20,12,'Name: ');
676: readln(retval);
677: if retval = '' then
678: retval := 'xxxxx';
679: restore_screen;
680: end;
681: end;
682: until (inp = #13) or (inp = #27)
683: or (inp = #82);
684: restore_screen;
685: select_datei := read;
686: end;
687: procedure General_Help;
688: const helptexte : array[1..16] of
689: string[66] =
690: (' ' Allgemeine Hilf'
691: ' '
692: ' V I S I O N F A C T O R Y ' +
693: ' M O D - E D I T O R ' ,
694: '
695: ' Mit dem VISION FACTORY MOD-Editor'+
696: ' können Sie MOD-Dateien
697: ' mit 4 und 8 Stimmen erstellen, di'+
698: ' e kompatibel zum Format
699: ' des TRITON Fast-Trackers sind.',
700: ' Folgende Funktionstasten sind defi'
701: '+niert ' ,
702: '
703: ' F1 = Help F8 = Play ' +
704: ' Song Sh-F6 = Save Track',
705: ' F2 = Select F9 = MIDI ' +
706: ' Record Sh-F7 = Copy Pattern',
707: ' F3 = MIDI ON Sh-F1 = Load ' +
708: ' Song Sh-F8 = Paste Pattern',
709: ' F4 = MIDI OFF Sh-F2 = Save ' +
710: ' Song Sh-F9 = Copy Track',
711: ' F5 = Lade Instr. Sh-F3 = Load ' +
712: ' Pattern Sh-F10 = Paste Track ' ,
713: ' F6 = Save Instr. Sh-F4 = Save ' +
714: ' Pattern',
715: ' F7 = Play Pattern Sh-F5 = Load ' +
716: ' Track',
717: var li : integer;
718: ch : char;
719: begin;
720: save_screen;
721: textcolor(14);
722: rahmen(2,07,4,67,19);
723: textcolor(LightGray);
724: for li := 1 to 16 do begin;
725: writexy(8,4+li,Helptexte[li]);
726: end;
727: writexy(58,21,'Enter = Ende');
728: repeat
729: ch := readkey;
730: until (ch = #27) or (ch=#13);
731: restore_screen;
732: locate_cursor;
733: end;
734: procedure display_Help(Kontext :
735: byte);
736: const helptexte : array[1..7,1..8] of
737: string[36] =
738: ((' ' N o t e n e i n g e b e n ' ,
739: '
740: ' Hier können Sie die zu spielenden'
741: ' Noten eingeben. Benutzen Sie dazu,
742: ' entweder die Tastatur über die
743: ' Tasten Q bis O (Auswahl über F2)
744: ' oder spielen Sie die Noten via MIDI
745: ' ein (F3/F4 = MIDI ON/OFF)
746: ' I n s t r u m e n t e n w a h l ' ,
747: '
748: ' Wählen Sie hier das zu spielende
749: ' Instrument aus. Erlaubte Werte
750: ' liegen im Bereich von 0 bis 31.
751: ' Sie können das Instrument auch
752: ' über F2 auswählen.
753: ' ),
754: (' ' E f f e k t e i n g e b e n '
755: '
756: ' Hier können Sie einen MOD-Effekt
757: ' eingeben. Der erlaubte Wertebereich
758: ' liegt zwischen 0 und F. Sie können
759: ' den Effekt auch über F2 auswählen.
760: ' Beachten Sie die Operanden !
761: ' ),
762: (' ' E f f e k t - O p p e r a n d e n e i n g e b e n '
763: '
764: ' Hier können Sie die Operanden eines
765: ' MOD-Effektes eingeben. Beachten Sie
766: ' dazu die Definition der Parameter
767: ' in der Anleitung !
768: ' ),
769: (' ' '
770: (' ' S a m p l e e d i t i e r e n '
771: '
772: ' Hier können Sie die Parameter des
773: ' aktiven Samples verändern.

```

```

774: N=Name S=Loop Start L=Loop Länge
775: V=Default-Volume G=Songname
776: F5=Instrument laden
777: F6=Instrument speichern
778: Song editieren
779:
780: Hier bestimmen Sie das Arrangement
781: des Songs.
782: Zahl = Pattern Nr <Zahl> wird an
783: dieser Stelle gespielt.
784: einf = Neues Pattern einfügen
785: entf = Pattern entfernen
786: Verschiedene Einstellungen
787:
788: Folgende Funktionen sind verfügbar:
789: S = Anzahl der Stimmen wählen (4/8)
790: M = MIDI Speed wählen (1-12)
791: Den Wert der Oktave können Sie über
792: die Tasten + und - erhöhen oder
793: verringern.
794: );
795: var li : integer;
796: ch,ew : char;
797: begin;
798: save_screen;
799: textcolor(14);
800: rahmen(2,20,4,40,12);
801: textcolor(LightGray);
802: for li := 1 to 8 do begin;
803: writexy(22,4+li,Helptexte[
804: Kontext,li]);
805: end;
806: writexy(22,14,'Fl = Help
807: Enter = Ende');
808: repeat
809: ch := readkey;
810: case ord(ch) of
811: 0 : begin;
812: ew := readkey;
813: case ord(ew) of
814: 59 : begin;
815: general_help;
816: end;
817: end;
818: end;
819: until (ch = #27) or (ch=#13);
820: restore_screen;
821: end;
822: locate_cursor;
823: end;
824: function eff_choice : byte;
825: const texte : array[1..11]
826: of string[30] =
827: (' 0 Arpeggio xx',
828: ' 1 Portamento Up xx',
829: ' 2 Portamento Down xx',
830: ' 3 Portamento to Note xx',
831: ' 4 Vibrato xy',
832: ' A Volume Slide x0/0x',
833: ' B Position Jump xx',
834: ' C Set Note Volume xx',
835: ' D Pattern Break -',
836: ' F Song Speed xx',
837: ' X Kein Effekt'
838: );
839: selectvalues : array[1..11]
840: of byte =
841: (0,1,2,3,4,10,11,12,13,15,0);
842: selected : byte = 1;
843: var li : integer;
844: inp : char;
845: retval : byte;
846: begin;
847: save_screen;
848: rahmen(2,20,4,32,16);
849: writexy(21,5,
850: ' Bitte Effekt auswählen');
851: writexy(21,7,' Effekt
852: Operanden');
853: repeat
854: textcolor(LightGray);
855: for li := 1 to 10 do begin;
856: writexy(21,7+li,texte[li]);
857: end;
858: textcolor(14);
859: writexy(21,7+selected,
860: texte[selected]);
861: inp := readkey;
862: if ord(inp) = 0 then inp :=
863: readkey;
864: case ord(inp) of
865: 13 : begin;
866: retval := selectvalues
867: [selected];
868: end;
869: 27 : begin;
870: retval := selectvalues[11];
871: end;
872: 71 : begin;
873: selected := 1;
874: end;
875: 72 : begin;
876: if selected > 1 then
877: dec(selected);
878: end;
879: 79 : begin;
880: selected := 11;
881: end;
882: 80 : begin;
883: if selected < 11 then
884: inc(selected);
885: end;
886: end;
887: until (inp = #13) or (inp = #27);
888: restore_screen;
889: eff_choice := retval;
890: end;
891: function Note_choice : word;
892: const texte : array[0..4,1..12]
893: of string[30] =

```

```

894: (('C 0','C#0','D 0','D#0','E 0',
895:      'F 0',
896:      'F#0','G 0','G#0','A 0','A#0','B 0'),
897:      ('C 1','C#1','D 1','D#1','E 1','F 1',
898:      'F#1','G 1','G#1','A 1','A#1','B 1'),
899:      ('C 2','C#2','D 2','D#2','E 2','F 2',
900:      'F#2','G 2','G#2','A 2','A#2','B 2'),
901:      ('C 3','C#3','D 3','D#3','E 3','F 3'),
902:      'F#3','G 3','G#3','A 3','A#0','B 3'),
903:      ('C 4','C#4','D 4','D#4','E 4','F 4',
904:      'F#4','G 4','G#4','A 4','A#4',
905:      'B 4')),
906:      Zeile : byte = 1;
907:      Spalte : byte = 2;
908:      var li,lj : integer;
909:      inp : char;
910:      retval : word;
911: begin;
912:   save_screen;
913:   rahmen(2,25,4,26,16);
914:   writexy(26,5,
915:   ' Bitte Note auswählen');
916:   repeat
917:     textcolor(LightGray);
918:     for lj := 0 to 4 do begin;
919:       for li := 1 to 12 do begin;
920:         writexy(26+lj*5,6+li,
921:         textc[lj,li]);
922:       end;
923:     end;
924:     textcolor(14);
925:     writexy(26+Spalte*5,6+Zeile,
926:     textc[Spalte,Zeile]);
927:     inp := readkey;
928:     if ord(inp) = 0 then inp :=
929:       readkey;
930:     case ord(inp) of
931:       13: begin;
932:         { Enter }
933:         retval :=
934:           Notenvalues[Spalte,
935:           Zeile];
936:         end;
937:       27: begin;
938:         { Abbruch }
939:         retval := 9999;
940:         end;
941:       71: begin;
942:         { Pos 1 }
943:         Zeile := 1;
944:         Spalte := 0;
945:         end;
946:       72: begin;
947:         { Pfeil up }
948:         if not ((Zeile = 1) and
949:         (Spalte = 0)) then
950:           dec(Zeile);
951:         if Zeile = 0 then begin;
952:           dec(Spalte);
953:           Zeile := 12;
954:         end;
955:         end;
956:       75: begin;
957:         { Pfeil links }
958:         if Spalte > 0 then
959:           dec(Spalte);
960:         end;
961:       77: begin;
962:         { Pfeil rechts }
963:         if Spalte < 4 then
964:           inc(Spalte);
965:         end;
966:       79: begin;
967:         { End }
968:         Zeile := 12;
969:         Spalte := 4;
970:         end;
971:       80: begin;
972:         { Pfeil down }
973:         if not ((Zeile = 12) and
974:         (Spalte = 4)) then
975:           inc(Zeile);
976:         if Zeile > 12 then begin;
977:           inc(Spalte);
978:           Zeile := 1;
979:         end;
980:         end;
981:     end;
982:     until (inp = #13) or (inp = #27);
983:     restore_screen;
984:     Note_choice := retval;
985:   end;
986:   procedure lade_Instrument(nr : byte;
987:   name : string);
988:   var sf : file;
989:   begin;
990:     if Sam_l[nr] <> 0 then freemem(
991:     Samp[nr],
992:     Sam_l[nr]);
993:     assign(sf,name);
994:     reset(sf,1);
995:     Sam_l[nr] := filesize(sf);
996:     getmem(Samp[nr],Sam_l[nr]);
997:     blockread(sf,Samp[nr]^,Sam_l[nr]);
998:     close(sf);
999:     Instramen[nr] := name;
1000:   end;
1001:   procedure entferne_Instrument(nr :
1002:   byte);
1003:   var sf : file;
1004:   begin;
1005:     freemem(Samp[nr],Sam_l[nr]);
1006:     Sam_l[nr] := 0;
1007:     Instramen[nr] := '';
1008:   end;

```

»editmod.pas« kann unter anderem einzelne Patterns sowie Tracks speichern und laden.


```

1009: function Inst_choice : word;
1010: const Zeile : byte = 1;
1011: Spalte : byte = 1;
1012: selectvalues : array[1..32]
1013: of byte =
1014: (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
1015: 16,17,18,19,20,21,22,23,24,25,26,27,
1016: 28,29,30,31,255
1017: );
1018: var di,li,lj : integer;
1019: inp : char;
1020: retval : word;
1021: dname : string;
1022: dummysp : pchar;
1023: mask : array[1..12] of char;
1024: begin;
1025: save_screen;
1026: rahmen(2,6,4,70,15);
1027: writexy(19,5,' Bitte Instrument'+
1028: ' auswählen');
1029: repeat
1030:   textcolor(LightGray);
1031:   for lj := 1 to 3 do begin;
1032:     for li := 1 to 10 do begin;
1033:       writexy(7+(lj-1)*22,6+li,
1034:         '');
1035:       gotoxy(7+(lj-1)*22,6+li);
1036:       write((lj-1)*10+li:2,'. ',
1037:         instnamen[(lj-1)*10+li]);
1038:     end;
1039:   end;
1040:   writexy(51,17,' '+
1041:     '');
1042:   gotoxy(51,17);
1043:   write('31.',instnamen[31]);
1044:   textcolor(14);
1045:   gotoxy(7+(Spalte-1)*22,6+Zeile);
1046:   write((Spalte-1)*10+Zeile:2,'. ',
1047:     instnamen[(Spalte-1)*10+Zeile]);
1048:   inp := readkey;
1049:   if ord(inp) = 0 then inp :=
1050:     readkey;
1051:   case ord(inp) of
1052:     13: begin;
1053:       { Enter }
1054:       retval := selectvalues[
1055:         (Spalte-1)*10+Zeile];
1056:       end;
1057:     27: begin;
1058:       { Abbruch }
1059:       retval := 255;
1060:       end;
1061:     71: begin;
1062:       { Pos 1 }
1063:       Zeile := 1;
1064:       Spalte := 1;
1065:       end;
1066:     72: begin;
1067:       { Pfeil up }
1068:       if not ((Zeile = 1) and
1069:         (Spalte = 1)) then
1070:         dec(Zeile);
1071:       if Zeile = 0 then begin;
1072:         dec(Spalte);
1073:         Zeile := 10;
1074:       end;
1075:       end;
1076:     75: begin;
1077:       { Pfeil links }
1078:       if (Spalte > 1) and (Zeile
1079:         <= 11) then dec(Spalte);
1080:       end;
1081:     77: begin;
1082:       { Pfeil rechts }
1083:       if Spalte < 3 then
1084:         inc(Spalte);
1085:       end;
1086:     79: begin;
1087:       { End }
1088:       Zeile := 10;
1089:       Spalte := 3;
1090:       end;
1091:     80: begin;
1092:       { Pfeil down }
1093:       if not ((Zeile = 11) and
1094:         (Spalte = 3)) then
1095:         inc(Zeile);
1096:       if (Zeile > 10) and
1097:         (spalte < 3) then begin;
1098:         inc(Spalte);
1099:         Zeile := 1;
1100:       end;
1101:       end;
1102:     82: begin;
1103:       { einf }
1104:       save_screen;
1105:       rahmen(1,17,8,43,5);
1106:       writexy(19,9,'Dateinamen ei'+
1107:         'ngeben (Enter for select)');
1108:       writexy(19,11,'Name: ');
1109:       readln(dname);
1110:       if (dname = '') or
1111:         (pos('*.','*.') <= 0) then
1112:         dname :=
1113:           select_datei('*.','*.');
1114:       restore_screen;
1115:       if dname <= 'xxxx' then
1116:         lade_instrument((Spalte-1)
1117:           *10+Zeile,dname);
1118:       end;
1119:     83: begin;
1120:       { entf }
1121:       Entferne_Instrument((Spalte-1)
1122:         *10+Zeile);
1123:       end;
1124:   end;
1125:   until (inp = #13) or (inp = #27);
1126:   restore_screen;
1127:   if retval <= 255 then
1128:     Activ_Inst := retval;

```

```

1129:   writexy(4,2,
1130:     '');
1131:   writexy(4,2,instnamen[Activ_Inst]);
1132:   gotoxy(4,3);
1133:   write('Groesse:',Sam_l[
1134:     Activ_Inst:5]);
1135:   write(' Start: ',
1136:     loop_s[Activ_Inst:5]);
1137:   write(' Länge: ',
1138:     loop_l[Activ_Inst:5]);
1139:   gotoxy(4,4);
1140:   write('Volume: ',
1141:     Sam_vol[Activ_Inst:2]);
1142:   Inst_choice := retval;
1143: end;
1144: procedure Speicher_Instrument(nr :
1145:   byte;
1146:   name : string);
1147: var sf : file;
1148: begin;
1149:   assign(sf,name);
1150:   rewrite(sf,1);
1151:   blockwrite(sf,Samp[nr]^,Sam_l[nr]);
1152:   close(sf);
1153: end;
1154: procedure save_Inst;
1155: const Zeile : byte = 1;
1156: Spalte : byte = 1;
1157: selectvalues : array[1..32]
1158: of byte =
1159: (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
1160: 16,17,18,19,20,21,22,23,24,25,26,27,
1161: 28,29,30,31,255
1162: );
1163: var di,li,lj : integer;
1164: inp : char;
1165: retval : word;
1166: dname : string;
1167: dummysp : pchar;
1168: mask : array[1..12] of char;
1169: begin;
1170:   save_screen;
1171:   rahmen(2,14,70,15);
1172:   writexy(12,5,'Zu speicherndes In'+
1173:     'strument auswählen');
1174:   repeat
1175:     textcolor(LightGray);
1176:     for lj := 1 to 3 do begin;
1177:       for li := 1 to 10 do begin;
1178:         gotoxy(2+(lj-1)*22,6+li);
1179:         write((lj-1)*10+li:2,'. ',
1180:           instnamen[(lj-1)*10+li]);
1181:       end;
1182:     end;
1183:     gotoxy(46,17);
1184:     write('31.',instnamen[31]);
1185:     textcolor(14);
1186:     gotoxy(2+(Spalte-1)*22,6+Zeile);
1187:     write((Spalte-1)*10+Zeile:2,'. ',
1188:       instnamen[(Spalte-1)*10+Zeile]);
1189:     inp := readkey;
1190:     if ord(inp) = 0 then inp :=
1191:       readkey;
1192:     case ord(inp) of
1193:       13: begin;
1194:         { Enter }
1195:         retval := selectvalues[
1196:           (Spalte-1)*10+Zeile];
1197:         end;
1198:       27: begin;
1199:         { Abbruch }
1200:         retval := 255;
1201:         end;
1202:       71: begin;
1203:         { Pos 1 }
1204:         Zeile := 1;
1205:         Spalte := 1;
1206:         end;
1207:       72: begin;
1208:         { Pfeil up }
1209:         if not ((Zeile = 1) and
1210:           (Spalte = 1)) then
1211:           dec(Zeile);
1212:         if Zeile = 0 then begin;
1213:           dec(Spalte);
1214:           Zeile := 10;
1215:         end;
1216:         end;
1217:       75: begin;
1218:         { Pfeil links }
1219:         if (Spalte > 1) and (Zeile <=
1220:           11) then dec(Spalte);
1221:         end;
1222:       77: begin;
1223:         { Pfeil rechts }
1224:         if Spalte < 3 then inc(Spalte);
1225:         end;
1226:       79: begin;
1227:         { End }
1228:         Zeile := 10;
1229:         Spalte := 3;
1230:         end;
1231:       80: begin;
1232:         { Pfeil down }
1233:         if not ((Zeile = 11) and
1234:           (Spalte = 3)) then
1235:           inc(Zeile);
1236:         if (Zeile > 10) and
1237:           (spalte < 3) then begin;
1238:           inc(Spalte);
1239:           Zeile := 1;
1240:         end;
1241:         end;
1242:     end;
1243:   until (inp = #13) or (inp = #27);
1244:   save_screen;
1245:   rahmen(1,12,6,43,5);
1246:   writexy(14,7,'Dateinamen eingeben '+
1247:     '(Enter for select)');
1248:   writexy(14,9,'Name: ');

```

```

1249:   readln(dname);
1250:   if (dname = '') or (pos('*.','*.') <= 0) then
1251:     dname := select_datei('*.','*.');
1252:   restore_screen;
1253:   if dname <= 'xxxx' then
1254:     Speicher_Instrument((Spalte-1)*10+
1255:       Zeile,dname);
1256:   end;
1257:   restore_screen;
1258: end;
1259: procedure init_The_Mod;
1260: begin;
1261:   mod_Setloopflag(ON);
1262:   init_sb;
1263:   mod_Samplefreq(22);
1264: end;
1265: procedure Play_The_Mod(s : string);
1266: var error : integer;
1267: begin;
1268:   mod_Samplefreq(22);
1269:   error := lade_moddatei(s,AUTO,AUTO);
1270:   if error <= 0 then begin;
1271:     textmode(3);
1272:     writeln('Fehler beim Laden der ',
1273:       'MOD-Datei !');
1274:     if error = -1 then writeln(
1275:       'Initialisierung fehlerhaft !');
1276:     if error = -2 then writeln(
1277:       'Datei nicht gefunden !');
1278:     halt(0);
1279:   end;
1280:   mli := 0;
1281:   mjl := 0;
1282:   { periodisch_on; }
1283: end;
1284: procedure init;
1285: begin;
1286:   init_the_mod;
1287:   redraw_screen;
1288:   clear_actual_pattern;
1289:   write_pattern(edit_Zeile);
1290:   write_arrangement(Edit_Arrang);
1291:   locate_cursor;
1292: end;
1293:
1294: procedure reset_Noten;
1295: begin;
1296:   Note1 := 0;
1297:   Note2 := 0;
1298:   Note3 := 0;
1299:   Note4 := 0;
1300:   Note5 := 0;
1301:   Note6 := 0;
1302:   Note7 := 0;
1303:   Note8 := 0;
1304: end;
1305: function MIDI_Note(note : byte) :
1306:   word;
1307: var Oktave,zeile : byte;
1308: begin;
1309:   dec(note,35);
1310:   Oktave := note DIV 12;
1311:   zeile := note - 12*Oktave;
1312:   MIDI_NOTE := notenvalues[Oktave,
1313:     zeile];
1314: end;
1315: procedure copy_pattern;
1316: begin;
1317:   move(Actual_Pattern,Puffer_Pattern,
1318:     2048);
1319: end;
1320: procedure Paste_Pattern;
1321: begin;
1322:   move(Puffer_Pattern,Actual_Pattern,
1323:     2048);
1324: end;
1325: procedure save_Pattern;
1326: var dname : string;
1327: tf : file;
1328: begin;
1329:   dname := select_datei('*.PTN',
1330:     '*.PTN');
1331:   if dname <= 'xxxx' then begin;
1332:     assign(tf,dname);
1333:     rewrite(tf,1);
1334:     blockwrite(tf,Puffer_Pattern,
1335:       sizeof(Puffer_Pattern));
1336:     close(tf);
1337:   end;
1338: end;
1339: procedure load_Pattern;
1340: var dname : string;
1341: tf : file;
1342: begin;
1343:   dname := select_datei('*.PTN',
1344:     '*.PTN');
1345:   if dname <= 'xxxx' then begin;
1346:     assign(tf,dname);
1347:     reset(tf,1);
1348:     blockread(tf,Puffer_Pattern,
1349:       sizeof(Puffer_Pattern));
1350:     close(tf);
1351:   end;
1352: end;
1353: procedure Copy_Track;
1354: var zeilen : byte;
1355: begin;
1356:   for zeilen := 1 to 64 do begin;
1357:     Puffer_Track[zeilen,1] :=
1358:       Actual_Pattern[zeilen,edit_track,1];
1359:     Puffer_Track[zeilen,2] :=
1360:       Actual_Pattern[zeilen,edit_track,2];
1361:     Puffer_Track[zeilen,3] :=
1362:       Actual_Pattern[zeilen,edit_track,3];
1363:     Puffer_Track[zeilen,4] :=

```

»editmod.pas« kann unter anderem einzelne Patterns sowie Tracks speichern und laden.


```

1364: Actual_Pattern[zeilen,edit_track,4];
1365: end;
1366: end;
1367: procedure Paste_Track;
1368: var zeilen : byte;
1369: begin;
1370:   for zeilen := 1 to 64 do begin;
1371:     Actual_Pattern[zeilen,edit_track,1]:=
1372:       Puffer_Track[zeilen,1];
1373:     Actual_Pattern[zeilen,edit_track,2]:=
1374:       Puffer_Track[zeilen,2];
1375:     Actual_Pattern[zeilen,edit_track,3]:=
1376:       Puffer_Track[zeilen,3];
1377:     Actual_Pattern[zeilen,edit_track,4]:=
1378:       Puffer_Track[zeilen,4];
1379:   end;
1380: end;
1381: procedure save_track;
1382: var dname : string;
1383:   tf : file;
1384: begin;
1385:   dname := select_datei('*.trk',
1386:     '*.trk');
1387:   if dname <> 'xxxx' then begin;
1388:     assign(tf,dname);
1389:     rewrite(tf,1);
1390:     blockwrite(tf,Puffer_Track,
1391:       sizeof(Puffer_Track));
1392:     close(tf);
1393:   end;
1394: end;
1395: procedure load_track;
1396: var dname : string;
1397:   tf : file;
1398: begin;
1399:   dname := select_datei('*.trk',
1400:     '*.trk');
1401:   if dname <> 'xxxx' then begin;
1402:     assign(tf,dname);
1403:     reset(tf,1);
1404:     blockread(tf,Puffer_Track,
1405:       sizeof(Puffer_Track));
1406:     close(tf);
1407:   end;
1408: end;
1409: procedure save_song;
1410: var dname : string;
1411:   stringarray : array[1..22]
1412:     of char;
1413:   chan,li,lj : integer;
1414:   mof : file;
1415:   helpw : word;
1416:   helpb : byte;
1417:   Pattern4 : array[1..64,1..4,1..4]
1418:     of byte;
1419:   Pattern8 : array[1..64,1..8,1..4]
1420:     of byte;
1421: begin;
1422:   dname := select_datei('*.mod',
1423:     '*.mod');
1424:   if dname <> 'xxxx' then begin;
1425:     assign(mof,dname);
1426:     rewrite(mof,1);
1427:     { Song-Namen schreiben }
1428:     for li := 1 to length(songname) do
1429:       stringarray[li] := songname[li];
1430:     stringarray[li+1] := #0;
1431:     blockwrite(mof,stringarray,20);
1432:     { Instrument - Infos }
1433:     for lj := 1 to 31 do begin;
1434:       for li := 1 to length(
1435:         instnamen[lj]) do
1436:         stringarray[li] :=
1437:           instnamen[lj,li];
1438:       stringarray[li+1] := #0;
1439:     { Instrumentname }
1440:     blockwrite(mof,stringarray,22);
1441:     helpw := Sam_l[lj];
1442:     helpw := helpw DIV 2;
1443:     helpw := swap(helpw);
1444:     { wgn. AMIGA }
1445:     { Sample - Länge }
1446:     blockwrite(mof,helpw,2);
1447:     helpb := 0;
1448:     { Fine Tune (NICHT UNIVERSITÄT ! ) }
1449:     blockwrite(mof,helpb,1);
1450:     { Noten - Volumen }
1451:     blockwrite(mof,Sam_vol[lj],1);
1452:     helpw := Loop_s[lj];
1453:     helpw := helpw DIV 2;
1454:     helpw := swap(helpw);
1455:     { wgn. AMIGA }
1456:     { Loop - Start }
1457:     blockwrite(mof,helpw,2);
1458:     helpw := Loop_l[lj];
1459:     helpw := helpw DIV 2;
1460:     helpw := swap(helpw);
1461:     { wgn. AMIGA }
1462:     { Loop - Länge }
1463:     blockwrite(mof,helpw,2);
1464:   end;
1465:   { Lied - Länge ( im Arrangement ) }
1466:   blockwrite(mof,liedlaenge,1);
1467:   { CIAA-Speed -> Müll schreiben da }
1468:   AMIGA-Spezifisch
1469:   blockwrite(mof,liedlaenge,1);
1470:   { Arrangement }
1471:   blockwrite(mof,Lied,128);
1472:   { Kennung }
1473:   if Stimmen = 8 then begin;
1474:     stringarray[1] := '8';
1475:     stringarray[2] := 'C';
1476:     stringarray[3] := 'H';
1477:     stringarray[4] := 'N';
1478:   end else begin;
1479:     stringarray[1] := 'M';
1480:     stringarray[2] := ' ';
1481:     stringarray[3] := 'K';
1482:     stringarray[4] := ' ';
1483:   end;

```

```

1484:   blockwrite(mof,stringarray,4);
1485:   for lj:=0 to Highest_track do begin;
1486:     if Stimmen = 4 then begin;
1487:       move(rm[lj]^,Pattern8,2048);
1488:       for li := 1 to 64 do begin;
1489:         for chan := 1 to 4 do begin;
1490:           Pattern4[li,chan,1] :=
1491:             Pattern8[li,chan,1];
1492:           Pattern4[li,chan,2] :=
1493:             Pattern8[li,chan,2];
1494:           Pattern4[li,chan,3] :=
1495:             Pattern8[li,chan,3];
1496:           Pattern4[li,chan,4] :=
1497:             Pattern8[li,chan,4];
1498:         end;
1499:       end;
1500:       blockwrite(mof,Pattern4,1024);
1501:     end else begin;
1502:       blockwrite(mof,rm[lj]^,2048);
1503:     end;
1504:   end;
1505:   for lj := 1 to 31 do begin;
1506:     blockwrite(mof,Samp[lj]^,Sam_l[lj]);
1507:   end;
1508:   close(mof);
1509: end;
1510: end;
1511: procedure init_PlayDaten;
1512: var li : byte;
1513: begin;
1514:   m_played := false;
1515:   In_St1 := 0;
1516:   In_St2 := 0;
1517:   In_St3 := 0;
1518:   In_St4 := 0;
1519:   In_St5 := 0;
1520:   In_St6 := 0;
1521:   In_St7 := 0;
1522:   In_St8 := 0;
1523:   getmem(blk,8000);
1524:   fillchar(blk^,8000,0);
1525:   for mlj:=0 to 128 do Lied[mlj] := 0;
1526:   pat_anz := 1;
1527:   mlj := 0;
1528:   mli := 0;
1529:   phase_1 := false;
1530:   phase_2 := true;
1531:   myc1 := 0;
1532:   Pattgroesse := 2048;
1533:   Stimmen := 8;
1534:   Vermische_Proc := vermische_start_8;
1535:   nmw_Proc := @nmw_all_8;
1536:   innen_proc := @innen_schleife_8n;
1537:   asm call [nmw_proc] end;
1538:   set_timeconst_sb16(Sampling_Rate);
1539:   Initialisiere_Vermischen;
1540:   Laufsec := 0;
1541:   Laufmin := 0;
1542:   wr_dsp_sb16(SD1);
1543:   for li := 0 to 63 do begin;
1544:     getmem(rm[li],2048);
1545:     fillchar(rm[li]^,2048,0);
1546:   end;
1547:   fillchar(actual_Pattern,2048,0);
1548: end;
1549: function choose_pattern : byte;
1550: var retval : byte;
1551: begin;
1552:   save_screen;
1553:   rahmen(2,30,12,20,5);
1554:   writexy(32,13,'Pattern eingeben');
1555:   writexy(32,15,'Pattern: ');
1556:   readln(retval);
1557:   restore_screen;
1558:   if retval > highest_track then
1559:     highest_track := retval;
1560:   choose_pattern := retval;
1561: end;
1562: procedure insert_pattern(start :
1563:   byte);
1564: var li : integer;
1565:   patt : byte;
1566: begin;
1567:   for li := 127 downto start do
1568:     Lied[li+1] := Lied[li];
1569:   Lied[start] := choose_pattern;
1570: end;
1571: procedure delete_pattern(start :
1572:   byte);
1573: var li : integer;
1574:   patt : byte;
1575: begin;
1576:   for li := start to 127 do
1577:     Lied[li] := Lied[li+1];
1578:   Lied[li] := 0;
1579:   highest_track := 0;
1580:   for li := 1 to 128 do
1581:     if Lied[li] > highest_track then
1582:       highest_track := Lied[li];
1583:   end;
1584: const timing_clock_message = 248;
1585:   active_sensing_message = 254;
1586:   start_note_on = 144;
1587:   end_note_on = 159;
1588: procedure kb;
1589: const exitit : boolean = false;
1590: var ch,sdz : char;
1591:   mididata : boolean;
1592:   Die_Note,dynamic : byte;
1593:   i : byte;
1594:   lsongname : string;
1595:   li : integer;
1596: begin;
1597:   repeat
1598:     if midi then begin;
1599:       mididata := false;
1600:       repeat
1601:         if ((port[dsp_adr+$E] and 128)
1602:           <> 0) then mididata := true;
1603:         inc(steps);

```

```

1604:   until (keypressed or mididata);
1605:   if keypressed then ch := readkey;
1606:   if mididata then begin;
1607:     ch := #255;
1608:     i:=port[dsp_adr+$A];
1609:     if(i<>$FE) and (i <>
1610:       Timing_clock_message) then begin
1611:       if ((i >= Start_Note_on) and
1612:         (i <= end_Note_on))
1613:         or (i < 128) then begin;
1614:         if i >= 128 then begin;
1615:           while not ((port[dsp_adr+$E]
1616:             and 128)<>0) do;
1617:             Die_Note := port[dsp_adr+$A];
1618:             end else Die_Note := i;
1619:           while ((port[dsp_adr+$E] and
1620:             128) = 0) do;
1621:             Dynamic := port[dsp_adr+$A];
1622:             if Dynamic <> 0 then begin;
1623:               Tonhoehe_in_pattern(
1624:                 edit_zeile+3,edit_track,
1625:                 MIDI_NOTE(Die_Note));
1626:               if edit_zeile < 61 then
1627:                 inc(edit_zeile);
1628:               write_Pattern(edit_zeile);
1629:             end;
1630:           end;
1631:         end;
1632:       end;
1633:     end else
1634:       ch := readkey;
1635:     case ord(ch) of
1636:       13: begin;
1637:         { Enter }
1638:         if edit_value = 8 then begin;
1639:           move(rm[lied[edit_arrang]]^
1640:             ,actual_Pattern,2048);
1641:           textcolor(white);
1642:           writexy(54,1,'Song');
1643:           edit_value := 1;
1644:           edit_track := 1;
1645:           write_Pattern(edit_zeile);
1646:         end;
1647:       end;
1648:       27: begin;
1649:         save_screen;
1650:         textcolor(4);
1651:         rahmen(2,23,9,34,3);
1652:         textcolor(14);
1653:         writexy(25,10,'Programm '+
1654:           'verlassen (J/N) ? ');
1655:         readln(sdz);
1656:         if (upcase(sdz) = 'J') then
1657:           exitit := true;
1658:         restore_screen;
1659:         locate_cursor;
1660:       end;
1661:       43: begin;
1662:         { + }
1663:         if edit_Oktave < 4 then
1664:           inc(edit_Oktave);
1665:         write_oktave;
1666:       end;
1667:       45: begin;
1668:         { - }
1669:         if edit_Oktave > 0 then
1670:           dec(edit_Oktave);
1671:         write_oktave;
1672:       end;
1673:       32: begin;
1674:         { leer }
1675:         if edit_value <= 6 then
1676:           Pattern_edited := true;
1677:         if (edit_value = 5) or
1678:           (edit_value = 6) then begin;
1679:           Opp_einer(edit_zeile+3,
1680:             edit_track,0);
1681:           Opp_zehner(edit_zeile+3,
1682:             edit_track,0);
1683:           write_Pattern(edit_zeile);
1684:         end;
1685:         if edit_value = 1 then begin;
1686:           Tonhoehe_in_pattern(
1687:             edit_zeile+3,edit_track,0);
1688:           write_Pattern(edit_zeile);
1689:         end;
1690:         if edit_value = 2 then begin;
1691:           Instrument_in_Pattern(
1692:             edit_zeile+3,edit_track,0);
1693:           write_Pattern(edit_zeile);
1694:         end;
1695:         if edit_value = 3 then begin;
1696:           Instrument_in_Pattern(
1697:             edit_zeile+3,edit_track,0);
1698:           write_Pattern(edit_zeile);
1699:         end;
1700:       end;
1701:       49: begin;
1702:         { 1 }
1703:         if edit_value <= 6 then
1704:           Pattern_edited := true;
1705:         if edit_value = 8 then begin;
1706:           Lied[edit_arrang] :=
1707:             choose_pattern;
1708:           write_arrangement(
1709:             edit_arrang);
1710:         end;
1711:         if edit_value = 6 then begin;
1712:           Opp_einer(edit_zeile+3,
1713:             edit_track,1);
1714:           write_Pattern(edit_zeile);
1715:         end;
1716:         if edit_value = 5 then begin;
1717:           Opp_zehner(edit_zeile+3,
1718:             edit_track,1);

```

»editmod.pas« kann unter anderem einzelne Patterns sowie Tracks speichern und laden.


```

1719:   inc(edit_value);
1720:   write_Pattern(edit_zeile);
1721: end;
1722: if edit_value = 4 then begin;
1723:   Effekt_in_Pattern(
1724:     edit_zeile+3,edit_track,1);
1725:   inc(edit_value);
1726:   write_Pattern(edit_zeile);
1727: end;
1728: if edit_value = 3 then begin;
1729:   Inst_einer(edit_zeile+3,
1730:     edit_track,1);
1731:   write_Pattern(edit_zeile);
1732: end;
1733: if edit_value = 2 then begin;
1734:   Inst_zehner(edit_zeile+3,
1735:     edit_track,1);
1736:   inc(edit_value);
1737:   write_Pattern(edit_zeile);
1738: end; end;
1739:
1740: 81,
1741: 113: begin;
1742: { q }
1743:   if edit_value <= 6 then
1744:     Pattern_edited := true;
1745:   Tonhoehe_in_Pattern(
1746:     edit_zeile+3,edit_track,
1747:     notenvalues[edit_Oktave,1]);
1748:   Instrument_in_Pattern(
1749:     edit_zeile+3,edit_track,
1750:     activ_inst);
1751:   if edit_zeile < 61 then
1752:     inc(edit_zeile);
1753:   write_Pattern(edit_zeile);
1754:   end;
1755: 50: begin;
1756: { 2 }
1757:   if edit_value <= 6 then
1758:     Pattern_edited := true;
1759:   if edit_value = 8 then begin;
1760:     Lied[edit_arrang] :=
1761:       choose_pattern;
1762:     write_arrangement(
1763:       edit_arrang);
1764:   end;
1765:   if edit_value = 6 then begin;
1766:     Opp_einer(edit_zeile+3,
1767:       edit_track,2);
1768:     write_Pattern(edit_zeile);
1769:   end;
1770:   if edit_value = 5 then begin;
1771:     Opp_zehner(edit_zeile+3,
1772:       edit_track,2);
1773:     inc(edit_value);
1774:     write_Pattern(edit_zeile);
1775:   end;
1776:   if edit_value = 4 then begin;
1777:     Effekt_in_Pattern(
1778:       edit_zeile+3,edit_track,3);
1779:     inc(edit_value);
1780:     write_Pattern(edit_zeile);
1781:   end;
1782:   if edit_value = 1 then begin;
1783:     Tonhoehe_in_Pattern(
1784:       edit_zeile+3,edit_track,
1785:       notenvalues[edit_Oktave,2]);
1786:     Instrument_in_Pattern(
1787:       edit_zeile+3,edit_track,
1788:       activ_inst);
1789:     if edit_zeile < 61 then
1790:       inc(edit_zeile);
1791:     write_Pattern(edit_zeile);
1792:   end;
1793:   if edit_value = 3 then begin;
1794:     Inst_einer(edit_zeile+3,
1795:       edit_track,2);
1796:     write_Pattern(edit_zeile);
1797:   end;
1798:   if edit_value = 2 then begin;
1799:     Inst_zehner(edit_zeile+3,
1800:       edit_track,2);
1801:     inc(edit_value);
1802:     write_Pattern(edit_zeile);
1803:   end;
1804:   end;
1805: 87,
1806: 119: begin;
1807: { w }
1808:   if edit_value <= 6 then
1809:     Pattern_edited := true;
1810:   Tonhoehe_in_Pattern(
1811:     edit_zeile+3,edit_track,
1812:     notenvalues[edit_Oktave,3]);
1813:   Instrument_in_Pattern(
1814:     edit_zeile+3,edit_track,
1815:     activ_inst);
1816:   if edit_zeile < 61 then
1817:     inc(edit_zeile);
1818:   write_Pattern(edit_zeile);
1819:   end;
1820: 51: begin;
1821: { 3 }
1822:   if edit_value <= 6 then
1823:     Pattern_edited := true;
1824:   if edit_value = 8 then begin;
1825:     Lied[edit_arrang] :=
1826:       choose_pattern;
1827:     write_arrangement(
1828:       edit_arrang);
1829:   end;
1830:   if edit_value = 6 then begin;
1831:     Opp_einer(edit_zeile+3,
1832:       edit_track,3);
1833:     write_Pattern(edit_zeile);
1834:   end;
1835:   if edit_value = 5 then begin;
1836:     Opp_zehner(edit_zeile+3,
1837:       edit_track,3);
1838:     inc(edit_value);

```

```

1839:   write_Pattern(edit_zeile);
1840: end;
1841: if edit_value = 4 then begin;
1842:   Effekt_in_Pattern(
1843:     edit_zeile+3,edit_track,3);
1844:   inc(edit_value);
1845:   write_Pattern(edit_zeile);
1846: end;
1847: if edit_value = 1 then begin;
1848:   Tonhoehe_in_Pattern(
1849:     edit_zeile+3,edit_track,
1850:     notenvalues[edit_Oktave,4]);
1851:   Instrument_in_Pattern(
1852:     edit_zeile+3,edit_track,
1853:     activ_inst);
1854:   if edit_zeile < 61 then
1855:     inc(edit_zeile);
1856:   write_Pattern(edit_zeile);
1857: end;
1858: if edit_value = 3 then begin;
1859:   Inst_einer(edit_zeile+3,
1860:     edit_track,3);
1861:   write_Pattern(edit_zeile);
1862: end;
1863: if edit_value = 2 then begin;
1864:   Inst_zehner(edit_zeile+3,
1865:     edit_track,3);
1866:   inc(edit_value);
1867:   write_Pattern(edit_zeile);
1868: end;
1869: end;
1870: 69,
1871: 101: begin;
1872: { e }
1873:   if edit_value <= 6 then
1874:     Pattern_edited := true;
1875:   if edit_value = 4 then begin;
1876:     Effekt_in_Pattern(
1877:       edit_zeile+3,edit_track,14);
1878:     inc(edit_value);
1879:     write_Pattern(edit_zeile);
1880:   end;
1881:   if edit_value = 1 then begin;
1882:     Tonhoehe_in_Pattern(
1883:       edit_zeile+3,edit_track,
1884:       notenvalues[edit_Oktave,5]);
1885:     Instrument_in_Pattern(
1886:       edit_zeile+3,edit_track,
1887:       activ_inst);
1888:     if edit_zeile < 61 then
1889:       inc(edit_zeile);
1890:     write_Pattern(edit_zeile);
1891:   end;
1892:   end;
1893: 82,
1894: 114: begin;
1895: { r }
1896:   if edit_value <= 6 then
1897:     Pattern_edited := true;
1898:   Tonhoehe_in_Pattern(
1899:     edit_zeile+3,edit_track,
1900:     notenvalues[edit_Oktave,6]);
1901:   if edit_zeile < 61 then
1902:     inc(edit_zeile);
1903:   write_Pattern(edit_zeile);
1904:   end;
1905: 52: begin;
1906: { 4 }
1907:   if edit_value <= 6 then
1908:     Pattern_edited := true;
1909:   if edit_value = 8 then begin;
1910:     Lied[edit_arrang] :=
1911:       choose_pattern;
1912:     write_arrangement(
1913:       edit_arrang);
1914:   end;
1915:   if edit_value = 6 then begin;
1916:     Opp_einer(edit_zeile+3,
1917:       edit_track,4);
1918:     write_Pattern(edit_zeile);
1919:   end;
1920:   if edit_value = 5 then begin;
1921:     Opp_zehner(edit_zeile+3,
1922:       edit_track,4);
1923:     inc(edit_value);
1924:     write_Pattern(edit_zeile);
1925:   end;
1926:   if edit_value = 4 then begin;
1927:     Effekt_in_Pattern(
1928:       edit_zeile+3,edit_track,4);
1929:     inc(edit_value);
1930:     write_Pattern(edit_zeile);
1931:   end;
1932:   if edit_value = 3 then begin;
1933:     Inst_einer(edit_zeile+3,
1934:       edit_track,4);
1935:     write_Pattern(edit_zeile);
1936:   end;
1937:   end;
1938: 53: begin;
1939: { 5 }
1940:   if edit_value <= 6 then
1941:     Pattern_edited := true;
1942:   if edit_value = 8 then begin;
1943:     Lied[edit_arrang] :=
1944:       choose_pattern;
1945:     write_arrangement(
1946:       edit_arrang);
1947:   end;
1948:   if edit_value = 6 then begin;
1949:     Opp_einer(edit_zeile+3,
1950:       edit_track,5);
1951:     write_Pattern(edit_zeile);
1952:   end;
1953:   if edit_value = 5 then begin;
1954:     Opp_zehner(edit_zeile+3,
1955:       edit_track,5);
1956:     inc(edit_value);
1957:     write_Pattern(edit_zeile);
1958:   end;

```

```

1959:   if edit_value = 4 then begin;
1960:     Effekt_in_Pattern(
1961:       edit_zeile+3,edit_track,5);
1962:     inc(edit_value);
1963:     write_Pattern(edit_zeile);
1964:   end;
1965:   if edit_value = 1 then begin;
1966:     Tonhoehe_in_Pattern(
1967:       edit_zeile+3,edit_track,
1968:       notenvalues[edit_Oktave,7]);
1969:     Instrument_in_Pattern(
1970:       edit_zeile+3,edit_track,
1971:       activ_inst);
1972:     if edit_zeile < 61 then
1973:       inc(edit_zeile);
1974:     write_Pattern(edit_zeile);
1975:   end;
1976:   if edit_value = 3 then begin;
1977:     Inst_einer(edit_zeile+3,
1978:       edit_track,5);
1979:     write_Pattern(edit_zeile);
1980:   end;
1981:   end;
1982: 84,
1983: 116: begin;
1984: { t }
1985:   if edit_value <= 6 then
1986:     Pattern_edited := true;
1987:   Tonhoehe_in_Pattern(
1988:     edit_zeile+3,edit_track,
1989:     notenvalues[edit_Oktave,8]);
1990:   if edit_zeile < 61 then
1991:     inc(edit_zeile);
1992:     write_Pattern(edit_zeile);
1993:   end;
1994: 54: begin;
1995: { 6 }
1996:   if edit_value <= 6 then
1997:     Pattern_edited := true;
1998:   if edit_value = 8 then begin;
1999:     Lied[edit_arrang] :=
2000:       choose_pattern;
2001:     write_arrangement(
2002:       edit_arrang);
2003:   end;
2004:   if edit_value = 6 then begin;
2005:     Opp_einer(edit_zeile+3,
2006:       edit_track,6);
2007:     write_Pattern(edit_zeile);
2008:   end;
2009:   if edit_value = 5 then begin;
2010:     Opp_zehner(edit_zeile+3,
2011:       edit_track,6);
2012:     inc(edit_value);
2013:     write_Pattern(edit_zeile);
2014:   end;
2015:   if edit_value = 4 then begin;
2016:     Effekt_in_Pattern(
2017:       edit_zeile+3,edit_track,6);
2018:     inc(edit_value);
2019:     write_Pattern(edit_zeile);
2020:   end;
2021:   if edit_value = 1 then begin;
2022:     Tonhoehe_in_Pattern(
2023:       edit_zeile+3,edit_track,
2024:       notenvalues[edit_Oktave,9]);
2025:     Instrument_in_Pattern(
2026:       edit_zeile+3,edit_track,
2027:       activ_inst);
2028:     if edit_zeile < 61 then
2029:       inc(edit_zeile);
2030:     write_Pattern(edit_zeile);
2031:   end;
2032:   if edit_value = 3 then begin;
2033:     Inst_einer(edit_zeile+3,
2034:       edit_track,6);
2035:     write_Pattern(edit_zeile);
2036:   end;
2037:   end;
2038: 90,
2039: 122: begin;
2040: { z }
2041:   if edit_value <= 6 then
2042:     Pattern_edited := true;
2043:   Tonhoehe_in_Pattern(
2044:     edit_zeile+3,edit_track,
2045:     notenvalues[edit_Oktave,10]);
2046:   if edit_zeile < 61 then
2047:     inc(edit_zeile);
2048:     write_Pattern(edit_zeile);
2049:   end;
2050: 55: begin;
2051: { 7 }
2052:   if edit_value <= 6 then
2053:     Pattern_edited := true;
2054:   if edit_value = 8 then begin;
2055:     Lied[edit_arrang] :=
2056:       choose_pattern;
2057:     write_arrangement(
2058:       edit_arrang);
2059:   end;
2060:   if edit_value = 6 then begin;
2061:     Opp_einer(edit_zeile+3,
2062:       edit_track,7);
2063:     write_Pattern(edit_zeile);
2064:   end;
2065:   if edit_value = 5 then begin;
2066:     Opp_zehner(edit_zeile+3,
2067:       edit_track,7);
2068:     inc(edit_value);
2069:     write_Pattern(edit_zeile);
2070:   end;
2071:   if edit_value = 4 then begin;
2072:     Effekt_in_Pattern(
2073:       edit_zeile+3,edit_track,7);

```

»editmod.pas« kann unter anderem einzelne Patterns sowie Tracks speichern und laden.


```

2074:     inc(edit_value);
2075:     write_Pattern(edit_zeile);
2076: end;
2077: if edit_value = 1 then begin;
2078:   Tonhoehe_in_Pattern(
2079:     edit_zeile+3,edit_track,
2080:     notenvalues[edit_Oktave,
2081:     11]);
2082:   Instrument_in_Pattern(
2083:     edit_zeile+3,edit_track,
2084:     activ_inst);
2085:   if edit_zeile < 61 then
2086:     inc(edit_zeile);
2087:   write_Pattern(edit_zeile);
2088: end;
2089: if edit_value = 3 then begin;
2090:   Inst_einer(edit_zeile+3,
2091:   edit_track,7);
2092:   write_Pattern(edit_zeile);
2093: end;
2094: end;
2095: 56: begin;
2096: { 8 }
2097:   if edit_value <= 6 then
2098:     Pattern_edited := true;
2099:   if edit_value = 8 then begin;
2100:     Lied(edit_arrang) :=
2101:       choose_pattern;
2102:     write_arrangement(
2103:       edit_arrang);
2104:   end;
2105:   if edit_value = 6 then begin;
2106:     Opp_einer(edit_zeile+3,
2107:     edit_track,8);
2108:     write_Pattern(edit_zeile);
2109:   end;
2110:   if edit_value = 5 then begin;
2111:     Opp_zehner(edit_zeile+3,
2112:     edit_track,8);
2113:     inc(edit_value);
2114:     write_Pattern(edit_zeile);
2115:   end;
2116:   if edit_value = 4 then begin;
2117:     Effekt_in_Pattern(
2118:       edit_zeile+3,edit_track,8);
2119:     inc(edit_value);
2120:     write_Pattern(edit_zeile);
2121:   end;
2122:   if edit_value = 3 then begin;
2123:     Inst_einer(edit_zeile+3,
2124:     edit_track,8);
2125:     write_Pattern(edit_zeile);
2126:   end;
2127: end;
2128: 85,
2129: 117: begin;
2130: { u }
2131:   if edit_value <= 6 then
2132:     Pattern_edited := true;
2133:   Tonhoehe_in_Pattern(
2134:     edit_zeile+3,edit_track,
2135:     notenvalues[edit_Oktave,12]);
2136:   if edit_zeile < 61 then
2137:     inc(edit_zeile);
2138:     write_Pattern(edit_zeile);
2139:   end;
2140: 73,
2141: 105: begin;
2142: { i }
2143:   if edit_value <= 6 then
2144:     Pattern_edited := true;
2145:   if edit_Oktave < 4 then
2146:     Tonhoehe_in_Pattern(
2147:       edit_zeile+3,edit_track,
2148:       notenvalues[edit_Oktave+1,11]);
2149:   else Tonhoehe_in_Pattern(
2150:     edit_zeile+3,edit_track,
2151:     notenvalues[edit_Oktave,11]);
2152:   if edit_zeile < 61 then
2153:     inc(edit_zeile);
2154:     write_Pattern(edit_zeile);
2155:   end;
2156: 57: begin;
2157: { 9 }
2158:   if edit_value <= 6 then
2159:     Pattern_edited := true;
2160:   if edit_value = 8 then begin;
2161:     Lied(edit_arrang) :=
2162:       choose_pattern;
2163:     write_arrangement(
2164:       edit_arrang);
2165:   end;
2166:   if edit_value = 6 then begin;
2167:     Opp_einer(edit_zeile+3,
2168:     edit_track,9);
2169:     write_Pattern(edit_zeile);
2170:   end;
2171:   if edit_value = 5 then begin;
2172:     Opp_zehner(edit_zeile+3,
2173:     edit_track,9);
2174:     inc(edit_value);
2175:     write_Pattern(edit_zeile);
2176:   end;
2177:   if edit_value = 4 then begin;
2178:     Effekt_in_Pattern(
2179:       edit_zeile+3,edit_track,9);
2180:     inc(edit_value);
2181:     write_Pattern(edit_zeile);
2182:   end;
2183:   if edit_value = 1 then begin;
2184:     if edit_Oktave < 4 then
2185:       Tonhoehe_in_Pattern(
2186:         edit_zeile+3,edit_track,
2187:         notenvalues[edit_Oktave+1,
2188:         2]) else Tonhoehe_in_Pattern(
2189:         edit_zeile+3,edit_track,
2190:         notenvalues[edit_Oktave,2]);
2191:     Instrument_in_Pattern(
2192:       edit_zeile+3,edit_track,
2193:       activ_inst);

```

```

2194:   if edit_zeile < 61 then
2195:     inc(edit_zeile);
2196:     write_Pattern(edit_zeile);
2197:   end;
2198:   if edit_value = 3 then begin;
2199:     Inst_einer(edit_zeile+3,
2200:     edit_track,9);
2201:     write_Pattern(edit_zeile);
2202:   end;
2203: end;
2204: 79,
2205: 111: begin;
2206: { o }
2207:   if edit_value <= 6 then
2208:     Pattern_edited := true;
2209:   if edit_Oktave < 4 then
2210:     Tonhoehe_in_Pattern(
2211:       edit_zeile+3,edit_track,
2212:       notenvalues[edit_Oktave+1,3]);
2213:   else Tonhoehe_in_Pattern(
2214:     edit_zeile+3,edit_track,
2215:     notenvalues[edit_Oktave,3]);
2216:   Instrument_in_Pattern(
2217:     edit_zeile+3,edit_track,
2218:     activ_inst);
2219:   if edit_zeile < 61 then
2220:     inc(edit_zeile);
2221:     write_Pattern(edit_zeile);
2222:   end;
2223: 77,
2224: 109: begin;
2225:   if edit_value = 9 then begin;
2226:     save_screen;
2227:     textcolor(14);
2228:     rahmen(2,28,10,29,5);
2229:     writexy(30,11,'MIDI-Speed '+
2230:     'eingegeben (1-12)');
2231:     readln(steplimit);
2232:     restore_screen;
2233:     if Steplimit < 1 then
2234:       Steplimit := 1;
2235:     if Steplimit > 12 then
2236:       Steplimit := 12;
2237:     textcolor(lightGray);
2238:     gotoxy(63,4);
2239:     write('MIDI Speed: ',
2240:     steplimit);
2241:   end;
2242: 48: begin;
2243: 48: begin;
2244: { 0 }
2245:   if edit_value <= 6 then
2246:     Pattern_edited := true;
2247:   if edit_value = 8 then begin;
2248:     Lied(edit_arrang) :=
2249:       choose_pattern;
2250:     write_arrangement(
2251:       edit_arrang);
2252:   end;
2253:   if edit_value = 6 then begin;
2254:     Opp_einer(edit_zeile+3,
2255:     edit_track,0);
2256:     write_Pattern(edit_zeile);
2257:   end;
2258:   if edit_value = 5 then begin;
2259:     Opp_zehner(edit_zeile+3,
2260:     edit_track,0);
2261:     inc(edit_value);
2262:     write_Pattern(edit_zeile);
2263:   end;
2264:   if edit_value = 4 then begin;
2265:     Effekt_in_Pattern(
2266:       edit_zeile+3,edit_track,0);
2267:     inc(edit_value);
2268:     write_Pattern(edit_zeile);
2269:   end;
2270:   if edit_value = 3 then begin;
2271:     Inst_einer(edit_zeile+3,
2272:     edit_track,0);
2273:     write_Pattern(edit_zeile);
2274:   end;
2275:   if edit_value = 2 then begin;
2276:     Inst_zehner(edit_zeile+3,
2277:     edit_track,0);
2278:     inc(edit_value);
2279:     write_Pattern(edit_zeile);
2280:   end;
2281:   if edit_value = 1 then begin;
2282:     if edit_Oktave < 4 then
2283:       Tonhoehe_in_Pattern(
2284:         edit_zeile+3,edit_track,
2285:         notenvalues[edit_Oktave+1,
2286:         4]) else Tonhoehe_in_Pattern(
2287:         edit_zeile+3,edit_track,
2288:         notenvalues[edit_Oktave,4]);
2289:     Instrument_in_Pattern(
2290:       edit_zeile+3,edit_track,
2291:       activ_inst);
2292:     if edit_zeile < 61 then
2293:       inc(edit_zeile);
2294:       write_Pattern(edit_zeile);
2295:     end;
2296:   end;
2297: 80,
2298: 112: begin;
2299: { p }
2300:   if edit_value <= 6 then
2301:     Pattern_edited := true;
2302:   if edit_Oktave < 4 then
2303:     Tonhoehe_in_Pattern(
2304:       edit_zeile+3,edit_track,
2305:       notenvalues[edit_Oktave+1,5]);
2306:   else Tonhoehe_in_Pattern(
2307:     edit_zeile+3,edit_track,
2308:     notenvalues[edit_Oktave,5]);
2309:   Instrument_in_Pattern(
2310:     edit_zeile+3,edit_track,
2311:     activ_inst);

```

```

2314:   if edit_zeile < 61 then
2315:     inc(edit_zeile);
2316:     write_Pattern(edit_zeile);
2317:   end;
2318: 154,
2319: 129: begin;
2320: { ü }
2321:   if edit_value <= 6 then
2322:     Pattern_edited := true;
2323:   if edit_Oktave < 4 then
2324:     Tonhoehe_in_Pattern(
2325:       edit_zeile+3,edit_track,
2326:       notenvalues[edit_Oktave+1,6]);
2327:   else Tonhoehe_in_Pattern(
2328:     edit_zeile+3,edit_track,
2329:     notenvalues[edit_Oktave,6]);
2330:   Instrument_in_Pattern(
2331:     edit_zeile+3,edit_track,
2332:     activ_inst);
2333:   if edit_zeile < 61 then
2334:     inc(edit_zeile);
2335:     write_Pattern(edit_zeile);
2336:   end;
2337: 65,
2338: 97: begin;
2339: { a }
2340:   if edit_value <= 6 then
2341:     Pattern_edited := true;
2342:   if edit_value = 4 then begin;
2343:     Effekt_in_Pattern(
2344:       edit_zeile+3,edit_track,10);
2345:     inc(edit_value);
2346:     write_Pattern(edit_zeile);
2347:   end;
2348: 66,
2349: 98: begin;
2350: { b }
2351:   if edit_value <= 6 then
2352:     Pattern_edited := true;
2353:   if edit_value = 4 then begin;
2354:     Effekt_in_Pattern(
2355:       edit_zeile+3,edit_track,11);
2356:     inc(edit_value);
2357:     write_Pattern(edit_zeile);
2358:   end;
2359: 67,
2360: 99: begin;
2361: { c }
2362:   if edit_value <= 6 then
2363:     Pattern_edited := true;
2364:   if edit_value = 4 then begin;
2365:     Effekt_in_Pattern(
2366:       edit_zeile+3,edit_track,12);
2367:     inc(edit_value);
2368:     write_Pattern(edit_zeile);
2369:   end;
2370: 68,
2371: 100: begin;
2372: { d }
2373:   if edit_value <= 6 then
2374:     Pattern_edited := true;
2375:   if edit_value = 4 then begin;
2376:     Effekt_in_Pattern(
2377:       edit_zeile+3,edit_track,13);
2378:     inc(edit_value);
2379:     write_Pattern(edit_zeile);
2380:   end;
2381: 70,
2382: 102: begin;
2383: { f }
2384:   if edit_value <= 6 then
2385:     Pattern_edited := true;
2386:   if edit_value = 4 then begin;
2387:     Effekt_in_Pattern(
2388:       edit_zeile+3,edit_track,15);
2389:     inc(edit_value);
2390:     write_Pattern(edit_zeile);
2391:   end;
2392: 78,
2393: 110: begin;
2394: { n }
2395:   if edit_value = 7 then begin;
2396:     save_screen;
2397:     textcolor(14);
2398:     rahmen(2,25,10,30,5);
2399:     writexy(27,11,'Bitte '+
2400:     'neuen Namen eingeben ');
2401:     writexy(27,13,'Name: ');
2402:     readln(Instnamen[
2403:     activ_inst]);
2404:     restore_screen;
2405:     writexy(4,2,' '+
2406:     ' ');
2407:     writexy(4,2,Instnamen[
2408:     Activ_Inst]);
2409:   end;
2410: 83,
2411: 115: begin;
2412: { s }
2413:   if edit_value = 7 then begin;
2414:     save_screen;
2415:     textcolor(14);
2416:     rahmen(2,28,10,29,5);
2417:     writexy(30,11,'Neuen Loop'+
2418:     '-Start eingeben ');
2419:     writexy(30,13,'Wert: ');
2420:     readln(Loop_s[activ_inst]);
2421:     restore_screen;
2422:     if Loop_s[activ_inst] > (

```

»editmod.pas« kann unter anderem einzelne Patterns sowie Tracks speichern und laden.


```

2429: Sam_l[activ_inst] - Loop_1
2430: [activ_inst] then Loop_s[
2431: activ_inst] := (Sam_l[
2432: activ_inst] - Loop_1[
2433: activ_inst]);
2434: writexy(4,3,' ');
2435:
2436: gotoxy(4,3);
2437: write('Groesse:');
2438: Sam_l[Activ_Inst]:5;
2439: write('Start: ');
2440: loop_s[Activ_Inst]:5;
2441: write('Länge: ');
2442: loop_l[Activ_Inst]:5;
2443: end;
2444: if edit_value = 9 then begin;
2445: save_screen;
2446: textcolor(14);
2447: rahmen(2,28,10,29,5);
2448: writexy(30,11,'Stimmen-'+
2449: 'Anzahl eingeben ');
2450: writexy(30,13,
2451: 'Anzahl (4/8): ');
2452: readln(Stimmen);
2453: restore_screen;
2454: if Stimmen <= 4 then
2455: Stimmen := 8;
2456: textcolor(lightgray);
2457: gotoxy(63,3);
2458: write('Stimmen: ',stimmen);
2459: end;
2460: end;
2461: 76,
2462: 108: begin;
2463: { l }
2464: if edit_value = 7 then begin;
2465: save_screen;
2466: textcolor(14);
2467: rahmen(2,28,10,29,5);
2468: writexy(30,11,'Neue Loop-'+
2469: 'Länge eingeben ');
2470: writexy(30,13,'Wert: ');
2471: readln(Loop_l[activ_inst]);
2472: restore_screen;
2473: if Loop_l[activ_inst] > (
2474: Sam_l[activ_inst] - Loop_s[
2475: activ_inst]) then Loop_l[
2476: activ_inst] := (Sam_l[
2477: activ_inst] - Loop_s[
2478: activ_inst]);
2479: writexy(4,3,' ');
2480:
2481: gotoxy(4,3);
2482: write('Groesse:');
2483: Sam_l[Activ_Inst]:5;
2484: write('Start: ');
2485: loop_s[Activ_Inst]:5;
2486: write('Länge: ');
2487: loop_l[Activ_Inst]:5;
2488: end;
2489: end;
2490: 71,
2491: 103: begin;
2492: { g }
2493: if edit_value = 7 then begin;
2494: save_screen;
2495: textcolor(14);
2496: rahmen(2,28,10,29,5);
2497: writexy(30,11,'Neue Song'+
2498: '-Namen eingeben ');
2499: writexy(30,13,'Name: ');
2500: readln(lsongname);
2501: if lsongname <= '' then
2502: lsongname := lsongname;
2503: restore_screen;
2504: writexy(4,5,' ');
2505:
2506: gotoxy(4,5);
2507: textcolor(lightgray);
2508: write('Songname: ',
2509: lsongname);
2510: end;
2511: end;
2512: 86,
2513: 118: begin;
2514: if edit_value = 7 then begin;
2515: save_screen;
2516: textcolor(14);
2517: rahmen(2,28,10,29,5);
2518: writexy(30,11,'Neue '+
2519: 'Volumen-Wert eingeben');
2520: writexy(30,13,'Wert: ');
2521: readln(Sam_vol[activ_inst]);
2522: restore_screen;
2523: if Sam_vol[activ_inst] > 64
2524: then Sam_vol[activ_inst]
2525: := 64;
2526: gotoxy(4,4);
2527: write('Volume: ',Sam_vol[
2528: Activ_Inst]:2);
2529: end;
2530: end;
2531: 0: begin;
2532: { Sonderzeichen }
2533: sdz := readkey;
2534: case ord(sdz) of
2535: 15: begin;
2536: { Shift - Tab }
2537: if Pattern_edited
2538: then begin;
2539: move(actual_Pattern,
2540: rm[Lied[edit_arrang]]^,
2541: 2048);
2542: Pattern_edited :=
2543: false;
2544: end;
2545: if edit_track > 1 then
2546: dec(edit_track) else
2547: edit_track := 1;
2548: if (edit_track > 8)

```

```

2549: then begin;
2550: edit_value :=
2551: edit_track-2;
2552: end else begin;
2553: write_Pattern(
2554: edit_zeile);
2555: edit_value := 1;
2556: end;
2557: if edit_value = 7 then
2558: begin;
2559: textcolor(white);
2560: writexy(54,1,'Song');
2561: textcolor(10);
2562: writexy(20,1,'Sample');
2563: end;
2564: if edit_value = 8 then
2565: begin;
2566: textcolor(white);
2567: writexy(67,1,' Misc');
2568: textcolor(10);
2569: writexy(54,1,'Song');
2570: write_Arrangement(
2571: Edit_Arrang);
2572: end;
2573: if edit_value = 9 then
2574: begin;
2575: textcolor(white);
2576: writexy(7,8,'Track 1');
2577: textcolor(10);
2578: writexy(67,1,' Misc');
2579: end;
2580: locate_cursor;
2581: end;
2582: 59: begin;
2583: { F1 }
2584:
2585: case edit_value of
2586: 1: display_Help(1);
2587: 2: display_Help(2);
2588: 3: display_Help(2);
2589: 4: display_Help(3);
2590: 5: display_Help(4);
2591: 6: display_Help(4);
2592: 7: display_Help(5);
2593: 8: display_Help(6);
2594: end;
2595: 60: begin;
2596: { F2 }
2597: if edit_value <= 6 then
2598: Pattern_edited := true;
2599: if edit_value = 4 then
2600: begin;
2601: Effekt_in_Pattern(3+
2602: edit_zeile,edit_track,
2603: eff_choice);
2604: inc(edit_value);
2605: write_Pattern(
2606: edit_zeile);
2607: end;
2608: if (edit_value = 2) or
2609: (edit_value = 3) then
2610: begin;
2611: Instrument_in_Pattern(
2612: edit_zeile+3,
2613: edit_track,inst_choice
2614: );
2615: write_Pattern(
2616: edit_zeile);
2617: end;
2618: if edit_value = 1 then
2619: begin;
2620: Tonhoehe_in_Pattern(
2621: edit_zeile+3,
2622: edit_track,
2623: Note_choice);
2624: Instrument_in_Pattern(
2625: edit_zeile+3,
2626: edit_track,
2627: activ_inst);
2628: if edit_zeile < 61
2629: then inc(edit_zeile);
2630: write_Pattern(
2631: edit_zeile);
2632: end;
2633: end;
2634: 61: begin;
2635: { F3 }
2636: wr_dsp_sb16($30);
2637: MIDI := true;
2638: textcolor(lightgray);
2639: writexy(63,2,'MIDI'+
2640: ' ON ');
2641: end;
2642: 62: begin;
2643: { F4 }
2644: wr_dsp_sb16($30);
2645: MIDI := false;
2646: textcolor(lightgray);
2647: writexy(63,2,'MIDI'+
2648: ' OFF ');
2649: end;
2650: 63: begin;
2651: { F5 }
2652: inst_choice;
2653: locate_cursor;
2654: end;
2655: 64: begin;
2656: { F6 }
2657: save_inst;
2658: locate_cursor;
2659: end;
2660: 65: begin;
2661: { F7 }
2662: gotoxy(1,1);
2663: write('edit_arrang: ',
2664: edit_arrang,
2665: 'rm[lied]: ',
2666: Lied[edit_arrang]);
2667: if edit_track <= 8 then
2668: begin;

```

```

2669: move(actual_Pattern,
2670: rm[Lied[edit_arrang]]^,
2671: 2048);
2672: end;
2673: if edit_value = 8 then
2674: begin;
2675: move(rm[Lied[
2676: edit_arrang]]^,
2677: actual_Pattern,2048);
2678: end;
2679: Reset_Noten;
2680: mli := 0;
2681: mli := edit_arrang-1;
2682: mididata := false;
2683: periodisch_on;
2684: repeat
2685: write_Pattern(mli-2);
2686: until (mli = 64) or
2687: keypressed;
2688: periodisch_off;
2689: while keypressed do
2690: readkey;
2691: locate_cursor;
2692: end;
2693: 66: begin;
2694: { F8 }
2695: move(actual_Pattern,
2696: rm[Lied[edit_arrang]]^,
2697: 2048);
2698: Reset_Noten;
2699: mli := 0;
2700: mli := edit_Arrang-1;
2701: cursor_off;
2702: periodisch_on;
2703: repeat
2704: if mli = 1 then
2705: begin;
2706: move(rm[Lied[mli]]^,
2707: actual_Pattern,
2708: 2048);
2709: edit_Arrang := mli;
2710: write_Arrangement(
2711: edit_Arrang);
2712: end;
2713: write_Pattern(mli-2);
2714: until keypressed;
2715: while keypressed do
2716: readkey;
2717: periodisch_off;
2718: cursor_on;
2719: locate_cursor;
2720: end;
2721: 67: begin;
2722: { F9 }
2723: if not midi then begin;
2724: wr_dsp_sb16($30);
2725: textcolor(lightgray);
2726: writexy(63,2,'MIDI'+
2727: ' ON ');
2728: end;
2729: repeat
2730: if steps > steplimit*
2731: 700 then steps := 0;
2732: mididata := false;
2733: if ((port[dsp_adr+$E])
2734: and 128) <> 0 then
2735: mididata := true;
2736: if mididata then begin;
2737: i:=port[dsp_adr+$A];
2738: if (i<>$FE) and (i <>
2739: Timing_clock_message)
2740: then begin
2741: if (i >=
2742: Start_Note_on) and
2743: (i <= end_Note_on)
2744: or (i < 128) then
2745: begin;
2746: if i >= 128 then
2747: begin;
2748: while not ((port[
2749: dsp_adr+$E]) and
2750: 128) <> 0 do;
2751: Die_Note := port
2752: [dsp_adr+$A];
2753: end else Die_Note
2754: := i;
2755: while ((port[
2756: dsp_adr+$E]) and
2757: 128) = 0 do;
2758: Dynamic := port[
2759: dsp_adr+$A];
2760: if Dynamic <= 0
2761: then begin;
2762: Tonhoehe_in_Pattern(
2763: edit_zeile+3,
2764: edit_track,
2765: MIDI_NOTE(Die_Note));
2766: Instrument_in_Pattern(
2767: edit_zeile+3,
2768: edit_track,
2769: activ_inst);
2770: end;
2771: end;
2772: end;
2773: inc(steps);
2774: if steps > steplimit*
2775: 700 then begin;
2776: if edit_zeile < 61
2777: then inc(edit_zeile);
2778: write_Pattern(
2779: edit_zeile);
2780: end;
2781: end;
2782: until keypressed or
2783: (edit_zeile = 61);

```

»editmod.pas« kann unter anderem einzelne Patterns sowie Tracks speichern und laden.


```

2784:         if not midi then begin;
2785:             wr_dsp_sbl6($30);
2786:             textcolor(LightGray);
2787:             writexy(63,2,'MIDI'+
2788:                 ' OFF ');
2789:         end;
2790:     end;
2791:     71: begin;
2792: { Pos 1 }
2793:     if edit_value <= 6
2794:     then begin;
2795:         edit_zeile := -2;
2796:         write_Pattern(
2797:             edit_zeile);
2798:     end;
2799:     if edit_value = 8 then
2800:     begin;
2801:         edit_Arrang := 1;
2802:         write_arrangement(
2803:             edit_Arrang);
2804:     end;
2805:     end;
2806:     72: begin;
2807: { Pfeil up }
2808:     case edit_value of
2809:     1,2,3,4,5,6:
2810:     begin;
2811:         if edit_zeile > -2
2812:         then dec(edit_zeile);
2813:         write_Pattern(
2814:             edit_zeile);
2815:     end;
2816:     8: begin;
2817:         if edit_Arrang > 1
2818:         then dec(
2819:             edit_Arrang);
2820:         write_arrangement(
2821:             edit_Arrang);
2822:     end;
2823:     end;
2824:     end;
2825:     73: begin;
2826: { Page up }
2827:     case edit_value of
2828:     1,2,3,4,5,6:
2829:     begin;
2830:         if edit_zeile > 4
2831:         then dec(edit_zeile,6)
2832:         else edit_zeile := -2;
2833:         write_Pattern(
2834:             edit_zeile);
2835:     end;
2836:     8: begin;
2837:         if edit_Arrang > 4
2838:         then dec(edit_Arrang,
2839:             4)
2840:         else edit_Arrang := 1;
2841:         write_arrangement(
2842:             edit_Arrang);
2843:     end;
2844:     end;
2845:     end;
2846:     75: begin;
2847:         if edit_track <= 8 then
2848:         begin;
2849:             if (edit_value > 1) then
2850:             dec(edit_value)
2851:             else edit_value := 6;
2852:         end;
2853:         locate_cursor;
2854:     end;
2855:     77: begin;
2856: { Pfeil rechts }
2857:     if edit_track <= 8
2858:     then begin;
2859:         if (edit_value < 6)
2860:         then inc(edit_value)
2861:         else edit_value := 1;
2862:     end;
2863:     locate_cursor;
2864:     end;
2865:     79: begin;
2866: { ende }
2867:     if (edit_value <= 6)
2868:     then begin;
2869:         edit_zeile := 61;
2870:         write_Pattern(
2871:             edit_zeile);
2872:     end;
2873:     if edit_value = 8
2874:     then begin;
2875:         edit_Arrang := 128;
2876:         write_arrangement(
2877:             edit_Arrang);
2878:     end;
2879:     end;
2880:     80: begin;
2881: { Pfeil down }
2882:     case edit_value of
2883:     1,2,3,4,5,6:
2884:     begin;
2885:         if edit_zeile < 61
2886:         then inc(edit_zeile);
2887:         write_Pattern(
2888:             edit_zeile);
2889:     end;
2890:     8: begin;
2891:         if edit_Arrang < 128
2892:         then inc(edit_Arrang);
2893:         write_arrangement(
2894:             edit_Arrang);
2895:     end;
2896:     end;
2897:     end;
2898:     81: begin;
2899: { Page down }
2900:     case edit_value of
2901:     1,2,3,4,5,6:
2902:     begin;
2903:         if edit_zeile < 55

```

```

2904:         then inc(edit_zeile,
2905:             6)
2906:         else edit_zeile :=
2907:             61;
2908:         write_Pattern(
2909:             edit_zeile);
2910:     end;
2911:     8: begin;
2912:         if edit_Arrang < 124
2913:         then inc(edit_Arrang,
2914:             4)
2915:         else edit_Arrang :=
2916:             128;
2917:         write_arrangement(
2918:             edit_Arrang);
2919:     end;
2920:     end;
2921:     end;
2922:     82: begin;
2923: { einfgr }
2924:     if edit_value <= 6
2925:     then begin;
2926:         Pattern_edited := true;
2927:         Insert_in_Pattern(
2928:             edit_zeile+3);
2929:         write_Pattern(
2930:             edit_zeile);
2931:     end;
2932:     if edit_value = 8
2933:     then begin;
2934:         Insert_Pattern(
2935:             edit_Arrang);
2936:         write_arrangement(
2937:             edit_Arrang);
2938:         move(rm[Lied[
2939:             edit_Arrang]]^,
2940:             actual_Pattern,2048);
2941:         textcolor(white);
2942:         writexy(54,1,'Song');
2943:         edit_value := 1;
2944:         edit_track := 1;
2945:         write_Pattern(
2946:             edit_zeile);
2947:     end;
2948:     end;
2949:     83: begin;
2950: { entf }
2951:     if edit_value <= 6
2952:     then begin;
2953:         Pattern_edited :=
2954:             true;
2955:         Delete_in_Pattern(
2956:             edit_zeile+3);
2957:         write_Pattern(
2958:             edit_zeile);
2959:     end;
2960:     if edit_value = 8 then
2961:     begin;
2962:         delete_pattern(
2963:             edit_Arrang);
2964:         write_arrangement(
2965:             edit_Arrang);
2966:     end;
2967:     end;
2968:     84: begin;
2969: { Sh-F1 }
2970: { load song }
2971:     if song_loaded then
2972:     ende_mod;
2973:     song_loaded := false;
2974:     lsongname :=
2975:         select_datei('*.mod',
2976:             '*.mod');
2977:     if lsongname <> 'xxxx'
2978:     then begin;
2979:         play_the_mod(
2980:             lsongname);
2981:         song_loaded := true;
2982:         redraw_screen;
2983:         move(rm[Lied[
2984:             edit_Arrang]]^,
2985:             actual_Pattern,2048);
2986:         write_arrangement(
2987:             edit_Arrang);
2988:         highest_track := 0;
2989:         for li := 1 to 128 do
2990:             if lied[li] >
2991:                 highest_track then
2992:                 highest_track :=
2993:                     lied[li];
2994:     end;
2995:     write_Pattern(
2996:         edit_zeile);
2997:     locate_cursor;
2998:     end;
2999:     85: begin;
3000: { Sh-F2 }
3001:     move(actual_Pattern,
3002:         rm[Lied[edit_Arrang]]^,
3003:         2048);
3004:     Save_Song;
3005:     write_Pattern(
3006:         edit_zeile);
3007:     locate_cursor;
3008:     end;
3009:     86: begin;
3010: { Sh-F3 }
3011:     Load_Pattern;
3012:     write_Pattern(
3013:         edit_zeile);
3014:     locate_cursor;
3015:     end;
3016:     87: begin;
3017: { Sh-F4 }
3018:     Save_Pattern;
3019:     write_Pattern(
3020:         edit_zeile);
3021:     locate_cursor;
3022:     end;
3023:     88: begin;

```

```

3024: { Sh-F5 }
3025:     Load_Track;
3026:     write_Pattern(
3027:         edit_zeile);
3028:     locate_cursor;
3029:     end;
3030:     89: begin;
3031: { Sh-F6 }
3032:     Save_Track;
3033:     write_Pattern(
3034:         edit_zeile);
3035:     locate_cursor;
3036:     end;
3037:     90: begin;
3038: { Sh-F7 }
3039:     Copy_Pattern;
3040:     write_Pattern(
3041:         edit_zeile);
3042:     move(actual_Pattern,
3043:         rm[Lied[edit_Arrang]]^,
3044:         2048);
3045:     locate_cursor;
3046:     end;
3047:     91: begin;
3048: { Sh-F8 }
3049:     Paste_Pattern;
3050:     write_Pattern(
3051:         edit_zeile);
3052:     move(actual_Pattern,
3053:         rm[Lied[edit_Arrang]]^,
3054:         2048);
3055:     locate_cursor;
3056:     end;
3057:     92: begin;
3058: { Sh-F9 }
3059:     Copy_track;
3060:     write_Pattern(
3061:         edit_zeile);
3062:     move(actual_Pattern,
3063:         rm[Lied[edit_Arrang]]^,
3064:         2048);
3065:     locate_cursor;
3066:     end;
3067:     93: begin;
3068: { Sh-F10 }
3069:     Paste_track;
3070:     write_Pattern(
3071:         edit_zeile);
3072:     move(actual_Pattern,
3073:         rm[Lied[edit_Arrang]]^,
3074:         2048);
3075:     locate_cursor;
3076:     end;
3077:     else begin;
3078:         end;
3079:     end;
3080:     end;
3081:     9: begin;
3082: { tab }
3083:     if Pattern_edited then begin;
3084:         move(actual_Pattern,
3085:             rm[Lied[edit_Arrang]]^,2048);
3086:         Pattern_edited := false;
3087:     end;
3088:     if edit_track < 11 then
3089:     inc(edit_track)
3090:     else begin;
3091:         edit_track := 1;
3092:         edit_value := 1;
3093:     end;
3094:     if (edit_track > 8) then begin;
3095:         edit_value := edit_track-2;
3096:         textcolor(white);
3097:         writexy(20,1,'Sample');
3098:         writexy(67,1,' Misc');
3099:         writexy(54,1,'Song');
3100:     end else begin;
3101:         write_Pattern(edit_zeile);
3102:     end;
3103:     if edit_value = 7 then begin;
3104:         textcolor(white);
3105:         writexy(54,1,'Song');
3106:         writexy(67,17,'Track 8');
3107:         textcolor(10);
3108:         writexy(20,1,'Sample');
3109:     end;
3110:     if edit_value = 8 then begin;
3111:         textcolor(white);
3112:         writexy(20,1,'Sample');
3113:         textcolor(10);
3114:         writexy(54,1,'Song');
3115:         write_Arrangement(
3116:             Edit_Arrang);
3117:     end;
3118:     if edit_value = 9 then begin;
3119:         textcolor(white);
3120:         writexy(54,1,'Song');
3121:         textcolor(10);
3122:         writexy(67,1,' Misc');
3123:     end;
3124:     locate_cursor;
3125:     end;
3126:     else begin;
3127:         end;
3128:     end;
3129:     until exitit;
3130: { esc }
3131:     end;
3132:     begin;
3133:     init;
3134:     init_Playdaten;
3135:     kb;
3136:     end.

```

»editmod.pas« kann unter anderem einzelne Patterns sowie Tracks speichern und laden.

Unterbrechung nach Maß

Zur Entgegennahme manueller Eingaben sind 18,2 Aktivierungen des Timer-Interrupts 1Ch pro Sekunde durchaus häufig genug. Bei der Soundprogrammierung ist man jedoch gezwungen, die Auslösefrequenz drastisch zu erhöhen.

Die Umprogrammierung des Timer-Interrupts 1Ch ist elementarer Bestandteil zahlreicher Aufgabenstellungen, in denen unter anderem das Timing von Prozessen eine Rolle spielt. Standardmäßig wird dieser Interrupt 18,2mal pro Sekunde aufgerufen. Für den Bereich Demo- und Soundprogrammierung ist es allerdings notwendig, den Timer so umzuprogrammieren, daß er zirka 1000mal pro Sekunde auslöst. Dadurch läßt sich die Soundroutine komplett im Hintergrund abwickeln und läuft quasi als zweite Task, parallel zur Hauptanwendung. Aufgrund der zentralen Stellung des Interrupts 1Ch in Verbindung mit ohnehin im »Hintergrund« aktiven Anwendungen oder Treibern, darunter Multitasking- oder Speichermanagement-Software, kann es jedoch zu Schwierigkeiten mit Programmen wie Qemm oder Windows kommen.

Um nun das Timer-Verhalten zu ändern, muß man direkt auf den Timer-Baustein des PC zugreifen. Der im PC vorhandene Timer-Baustein trägt den Namen Intel 8254-PIT, wobei PIT für Peripheral Interval Timer steht. Dieser Baustein kümmert sich um verschiedene Aufgaben, so ist er für die Echtzeituhr, die Tonerzeugung mit dem PC-Lautsprecher und für den RAM-

Refresh notwendig. Der 8254 stellt dafür drei Kanäle zur Verfügung. Der erste wird benutzt, um die Systemuhr zu steuern, der zweite ist für den DMA-Controller zuständig (RAM-Refresh), der dritte steuert den PC-internen Lautsprecher.

Wie sein Name schon vermuten läßt, zählt der PIT intern lediglich Zähler herab, bis alle, in diesem Fall drei Register den Wert 0 aufweisen. Sodann liefert der Baustein das Ausgangssignal und setzt die Zähler auf ihren Standardwert. Das Vermindern der Zähler startet daraufhin von neuem.

Angesprochen wird der PIT 8254 über die Port-Adressen 40h bis 43h. Die Adressen 40h bis 42h dienen dabei für die Initialisierungswerte, während 43h für die Steuerung des 8254 zuständig ist. Die genaue Funktion der Bits des Steuerregisters entnehmen Sie der Tabelle.

Um den Timer auf eine beliebige Frequenz zu programmieren, gehen Sie folgendermaßen vor:

- Der Registerzähler berechnet sich nach der Formel

$\text{Reg_Zaehler} := 1193180 \text{ DIV Frequenz};$

- Schreiben Sie den Wert 36h an die Port-Adresse 43h.

- Das Lo-Byte und schließlich das Hi-Byte des Registerzählers übertragen Sie nun

Bit	Inhalt	Bedeutung
0	1	BCD-Zähler
	0	Binärzähler
1-3	000	Modus 0, Interrupt-Erzeugung
	001	Modus 1, programmierbarer Monoflop
	010	Modus 2, Taktgenerator
	011	Modus 3, Rechtecksignalgenerator
	100	Modus 4, Triggern des Ausgangs durch Software
	101	Modus 5, Triggern des Ausgangs durch Hardware
	110	reserviert
	111	reserviert
4-5	00	Zähler zum Lesen übernehmen
	01	Lo-Byte laden
	10	Hi-Byte laden
	11	zuerst Lo-Byte, dann Hi-Byte laden
6-7	00	Zähler 0 programmieren
	01	Zähler 1 programmieren
	10	Zähler 2 programmieren
	11	reserviert

Das Steuerregister des Intel 8254-PIT

an den Port 40h.

Um den Timer wieder auszuschalten, verwenden Sie folgendes Schema:

- Schreiben Sie den Wert 36h an den Port 43h.

- Die Port-Adresse 40h erhält zweimal den Wert 0 (Hi-Byte/Lo-Byte).

Wenn Sie den Timer-Interrupt umlenken, sollten Sie allerdings nicht einfach den alten Interrupt 1Ch ersatzlos überschreiben, sondern vielmehr den Timer-Interrupt auf Ihre eigene Routine umleiten, von der aus Sie den alten Timer-Interrupt 18,2mal pro Sekunde aufrufen. Diese Technik ist deshalb so wichtig, weil andere Programme auch auf diesen Interrupt zugreifen und dann unter Umständen gar nicht mehr oder nur mit erheblichen Fehlern arbeiten. Zur einfachen Handhabung des Timers bietet sich die Unit »timer.pas« (Listing) geradezu an.

(Boris Bertelsohn/bj)

Listing: timer.pas

```

1: {
2: Programm: Unit timer.pas
3: Funktion: Unit zum Umprogrammieren
4: des Timer-Interrupts
5: Sprache: Turbo Pascal ab 6.0
6: Autor: Boris Bertelsohn
7: (c) 1993 DMV Widuch GmbH & Co. KG
8: }
9:
10: UNIT Timer;
11:
12: interface
13: uses dos;
14:
15: CONST Timer_per_Second = 1000;
16:       OldTimerInt      = $71;
17:
18: Var Timerzaehler : word;
19:
20: Procedure timer_on;
21: Procedure timer_off;
22:
23: implementation
24:
25: procedure StelleTimerEin(Proc :
26: pointer; Freq : word);
27: {
28: Der Timer wird so umprogrammiert, daß
29: er mit der übergebenen Frequenz die
30: in Proc übergebene Prozedur aufruft.
31: Der alte Timerinterrupt wird auf INT
32: $71 verbogen }
33: var izaehler : word;
34:     oldv : pointer;

```

```

35: begin;
36: asm cli end;
37: izaehler := 1193180 DIV Freq;
38: Port[$43] := $36;
39: Port[$40] := Lo(izaehler);
40: Port[$40] := Hi(izaehler);
41: GetIntVec(8,oldv);
42: setIntVec(OldTimerInt,oldv);
43: SetIntVec(8,Proc);
44: Timerzaehler :=
45:   round(timer_per_second / 18.2);
46: asm sti end;
47: end;
48:
49: procedure StelleTimerAus;
50: {
51: Stellt den Urzustand des Timers her
52: }
53: var oldv : pointer;
54: begin;
55:   asm cli end;
56:   port[$43] := $36;
57:   Port[$40] := 0;
58:   Port[$40] := 0;
59:   GetIntVec(OldTimerInt,oldv);
60:   SetIntVec(8,oldv);
61:   asm sti end;
62: end;
63:
64: procedure NeuerTimer; interrupt;
65: {
66: Unser neuer Timer-Interrupt
67: }
68: var dummyreg : registers;
69: begin;
70:   dec(Timerzaehler);

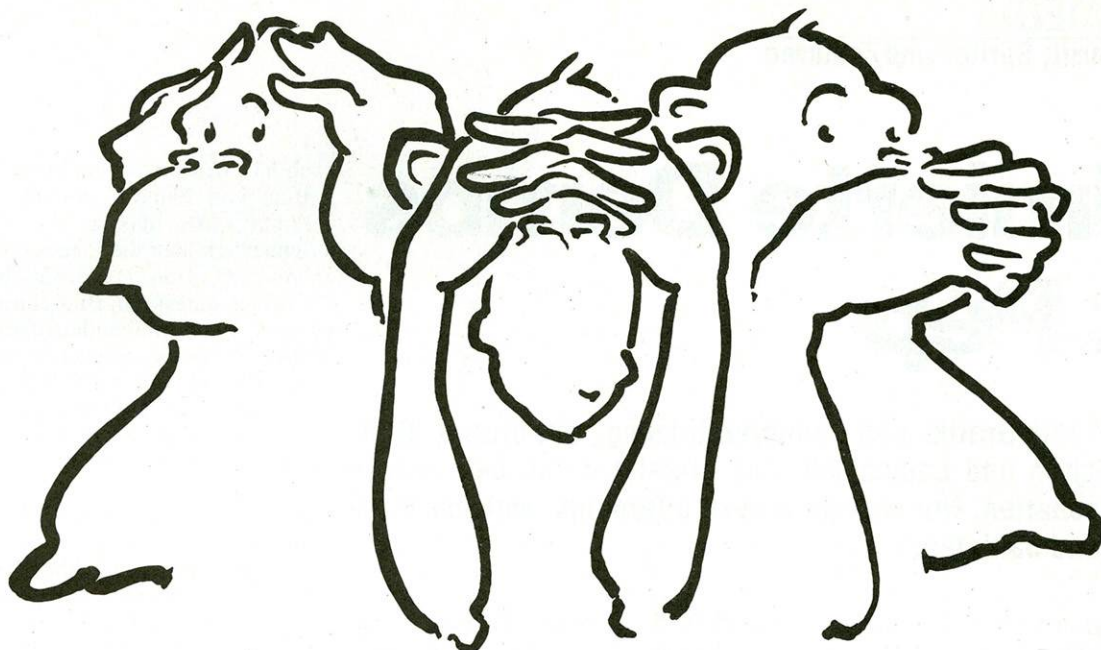
```

```

71: if Timerzaehler = 0 then begin;
72:   Timerzaehler :=
73:     round(timer_per_second / 18.2);
74:   intr(OldTimerInt,dummyreg);
75: end;
76: {
77: Hier können Ihre Prozeduren stehen
78: }
79: Port[$20] := $20;
80: end;
81:
82: Procedure timer_on;
83: {
84: Schaltet den Timer ein.
85: }
86: Begin
87:   StelleTimerEin(@NeuerTimer,
88:     timer_per_second);
89: End;
90:
91: Procedure timer_off;
92: {
93: Schaltet den Timer wieder aus
94: }
95: Begin
96:   StelleTimerAus;
97: End;
98:
99: begin;
100: end.

```

»timer.pas« ist eine Unit zur Manipulation des Timer-Interrupts 1Ch.



BERGWALDSTERBEN, WIESO? DA WIRD DOCH ÜBERTRIEBEN.

Irrtum. Der Kollaps steht in den Bergen unmittelbar bevor.

Giftige Luft macht Bäume krank, viele gesunde müssen Skipisten weichen, Wild nagt an den Setzlingen. Erdreich und Schnee finden keinen Halt mehr. Immer häufiger gehen vernichtende Lawinen zu Tal.

Wenn Sie wissen wollen, was zu tun ist, damit die Berge bewohnbar bleiben, dann füllen Sie den Coupon aus. Wir beantworten Ihre Frage, sagen Ihnen, wo wir uns noch engagieren, warum Sie uns dabei helfen können.

Ja, ich will von ROBIN WOOD wissen, was zu tun ist, damit die Berge bewohnbar bleiben.

Name, Vorname

Straße

Wohnort

Weil ich helfen will, lege ich 3 Mark in
Briefmarken bei. Coupon einsenden an

ROBIN WOOD,
Postfach 102122, 28021 Bremen

ROBIN WOOD

Gewaltfreie Aktionsgemeinschaft für Natur und Umwelt e.V.



Animierte Demos mit Pep

Gute Demos mit Grafik- und Tonunterstützung, animierten Bildschirmbereichen und Laufschrift sind allseits gerne betrachtete (Werbe-)Botschaften. Nur wenige wissen allerdings, welcher technische Aufwand dahintersteht.

Um die Programmierung des mit der VGA-Karte eingeführten neuen 256-Farben-Grafikmodus weitgehend einfach zu halten, erfand IBM vor Jahren den Grafikmodus 13h (320 x 200 Punkte bei 256 Farben). Diese Betriebsart erlaubt – wie der eine oder andere vielleicht schon zu schätzen gelernt hat – eine lineare Adressierung des kompletten Bildschirmspeichers, das heißt, jeder Bildschirmpunkt wird durch genau ein Byte repräsentiert. Da dadurch bereits mit einem Bild der gesamte der Grafikkarte zur Verfügung stehende Adreßraum von 64 KByte erschöpft ist (320 x 200 = 64 000), bleibt dem Programmierer der Zugriff auf verschiedene Bildschirmseiten verwehrt. Dadurch schränken sich die Anwendungsgebiete dieses Modus doch stark ein. Zum Ansehen oder eventuell auch zum Editieren von Bildern benötigt man keine flüssigen Animationen. Will man jedoch den Monitor zum Leben erwecken und beispielsweise Sprites über den Bildschirm laufen lassen, stößt man unweigerlich auf das Problem der einen Bildschirmseite, da es ausgeschlossen ist, ein komplexes Bild innerhalb der minimalen Zeit aufzubauen, die einem bei jedem Bildaufbau zur Verfügung steht.

Wird dennoch versucht, ein neues Motiv aufzubauen, während sich der Strahl noch auf dem Schirm befindet, kommt es sehr leicht zu geringfügig versetzten Bildteilen, die sich dem Betrachter als Flimmern und Ruckeln offenbaren. Daher darf man das Bild nur während des vertikalen Strahlrücklaufs verändern. Die entsprechende Berechnung gestaltet sich unkompliziert: Die VGA-Karte baut ihr Bild 70mal in der Sekunde auf dem Monitor auf. Davon entfällt aber leider nur ein verschwindend geringer Anteil (zirka 0,5 Prozent) auf den sogenannten Strahlrücklauf. Er bezeichnet die Zeit, die der Kathodenstrahl nach dem Aufbau des kompletten Bildes benötigt, um vom unteren Bildschirmrand in ausgeschaltetem Zustand zum oberen Rand zu wandern, um von dort aus den nächsten Bildaufbau zu starten. Somit stehen für eine vernünftige Bildmanipulation nur zirka 60 µs zur Verfügung, was auf einem mit 33 MHz getakteten Rechner etwa 2100 Taktzyklen entspricht. In dieser Zeit kann

die 386-CPU etwa 500 Byte kopieren. Weil der gute alte ISA-Bus, an dem auch im beginnenden Zeitalter des Local Bus noch die meisten Grafikkarten ihren Dienst verrichten, jedoch nur mit 8 MHz getaktet ist, verringert sich der Datendurchsatz nochmals mindestens um den Faktor 4. Daraus läßt sich erkennen, daß für eine flüssige und vor allem flimmerfreie Animation mindestens zwei Bildschirmseiten nötig sind. Doch genau dies bietet der zuvor erwähnte Modus 13h eben nicht. Mit ein wenig Trickserei können Sie die VGA-Karte jedoch in einer Weise umprogrammieren, daß sie sich zwar nicht mehr ganz so einfach einsetzen läßt, dafür aber die Verwendung mehrerer Bildschirmseiten erlaubt. Doch wie ist das bei 64 KByte Adreßraum zu bewerkstelligen? Hierzu muß auf ähnliche Techniken zurückgegriffen werden wie zur direkten Programmierung der 16-Farben-Modi. Der Bildschirmspeicher der VGA-Karte setzt sich aus vier Bitplanes zusammen, die praktisch parallel im Adreßraum liegen. Hinter jedem Bit im Bildschirmspeicher verbergen sich in Wahrheit vier Bits, die über bestimmte Register angesteuert werden. Dabei repräsentieren die einzelnen Planes im 16-Farben-Modus jeweils ein Bit der Farbe eines Bildpunktes. Das heißt, um die Farbe des Punktes (0/0) zu bestimmen, muß man aus allen vier Planes jeweils das erste Bit lesen und zu einem neuen Byte kombinieren. Das Bit aus Plane 0 erscheint im Farbwert als Bit 0 und so weiter.

Ein neuer Grafikmodus entsteht

In dem im folgenden beschriebenen neuen Grafikmodus (künftig als »Modus x« bezeichnet) besitzen die Planes eine etwas andere Bedeutung: Jeweils ein komplettes Byte einer bestimmten Plane steht für ein Pixel; das heißt, um einen bestimmten Punkt zu manipulieren, muß der Programmierer die entsprechende Bitplane wählen und deren Byte auslesen oder beschreiben. Eine Aufteilung der Bits dieser Farbe ist nicht erforderlich, wäre auch bei 8 Bit Farbtiefe ein wenig komplizierter. Dabei verteilen sich die Planes wie folgt über das Bild: Das erste Byte von Plane 0

enthält den ersten Punkt (0/0), das erste Byte von Plane 1 enthält den zweiten Punkt (1/0), das zweite Byte spricht schließlich über die Planes 0 bis 3 die Koordinaten (4 bis 7) an. Also befinden sich an einer Adresse im Bildschirmspeicher jeweils vier aufeinanderfolgende Punkte. Dies ist die grundlegende Organisation, der offizielle 320 x 200-Modus (13h) faßt nun diese vier Byte jeweils zu einem zusammen, wodurch ein Byte wieder ein Pixel darstellt. Diesen Vorgang nennt man »Chaining«.

Der eigentliche Aufbau des Bildschirmspeichers wird in den Prozeduren *Get_Point* (der Einfachheit halber Pascal) und *Point* (komplett Assembler) der Unit »s_unit.pas« (Listing 1) deutlich, die einen Punkt lesen beziehungsweise setzen, wobei die Koordinaten zunächst unter Berücksichtigung der Seite in einen Offset umgerechnet werden, die Plane ermittelt sowie gesetzt und schließlich der Punkt selbst geschrieben wird.

Grundlage für den neuen Grafikmodus ist der Modus 13h, den Sie auch zunächst über das BIOS (Interrupt 10h) setzen, womit die meisten Register (vor allem die zum Kathodenstrahl-Timing) bereits korrekte Werte erhalten. Einige Register sind jedoch nachträglich von Hand zu initialisieren, denn hier bietet das BIOS keine Unterstützung an. Zunächst müssen Sie dem sogenannten Sequenz-Controller (SC, Indexport an 03c4h, Datenport an 03c5h) mitteilen, daß die Planes wieder einzeln angesprochen werden sollen (Chaining aufheben). Dies erfolgt über das Speichermodus-Register 04h, in dem man Bit 3 löscht und Bit 2 setzt (den entsprechenden Quellcode finden Sie in der Prozedur *Init_VGA* in Listing 1).

Aufgabe des Sequenzers ist im Prinzip die Speicherverwaltung der VGA-Karte, nebenbei steuert er auch das Timing der Bilddarstellung. Außer dem Speichermodus-Register ist bei diesem Chip vor allem noch das Map-Maskierungs-Register von Bedeutung, anhand dessen festgelegt wird, auf welche Planes sich die folgenden Schreibzugriffe der CPU beziehen. So ist es beispielsweise für ein Löschen beziehungsweise Füllen des Bildschirms oder für das Kopieren von einer Seite auf eine andere sinnvoll, alle vier Planes gleichzeitig anzusprechen, wodurch eine Anweisung vier Byte gleichzeitig manipuliert. Zur Veränderung einzelner Pixel – wie für die Sprite-Programmierung unabdingbar – ist ein anderer Ansatz gefordert. Hier sind einzelne Pixel gezielt zu beeinflussen, also darf auch nur eine Plane auf einmal beschrieben werden.

An dieser Stelle ist eine nähere Erklärung der Plane-Verwaltung erforderlich. Dazu sind zunächst einmal die Latches von Bedeutung. Hierbei handelt es sich um vier interne Register der VGA-Karte, die direkt mit den entsprechenden Planes korrelieren. Bei einem Lesezugriff seitens der CPU

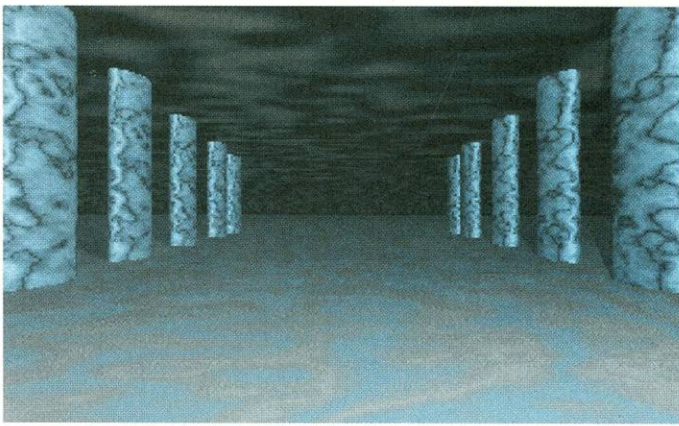


Bild 1. »phint.gif« stellt das Hintergrundmotiv für das Demo »simple_s.pas« dar.

liest das VGA-BIOS an der vom Prozessor auf dem Adreßbus übergebenen Adresse alle vier Planes aus und legt sie in den Latches ab – unabhängig vom jeweiligen Lese-Modus und der dort selektierten Plane. In diesem Fall bleiben die Werte bis zum nächsten Lesebefehl erhalten.

Der Schreibzugriff ist in verschiedene Schreibmodi unterteilt, wobei für Sie nur Modus 0 und 1 von Interesse sind. Im Schreibmodus 0 werden selektierte Latches zunächst mit den ursprünglichen Werten an dieser Adresse geladen und dann mit dem von der CPU übergebenen Byte verknüpft (oder auch ersetzt, was bei den Sprite-Routinen Anwendung findet) sowie schließlich in die Planes zurückgeschrieben. Schreibmodus 1 ist dagegen sehr viel einfacher. Er schreibt nur den aktuellen Inhalt der Latches (geladen durch einen der beiden Lese-Modi) an die am Adreßbus anliegende Adresse in den Bildschirmspeicher, das auf dem Datenbus anliegende CPU-Byte kümmert ihn überhaupt nicht.

Optimale Bildschirm-synchronisation

Mit dem Auflösen des Chaining ist die Initialisierung leider noch nicht ganz abgeschlossen. Auch der CRTC (Cathod Ray Tube Controller = Kathodenstrahlröhren-Controller, Port 3d4h/3d5h), der hauptsächlich die Synchronisation des Bildes steuert und außerdem für den RAM-Refresh des Bildschirmspeichers zuständig ist, muß noch mit passenden Werten versorgt werden. Zunächst schalten Sie die Doppelwort-Adressierung des Bildschirmspeichers aus, wozu Sie in Register 14h (Underline-Location) Bit 6 durch eine AND-Maskierung mit 0bfh löschen. Zum Abschluß erhält der CRTC durch Setzen von Bit 6 im Mode-Control-Register (17h) die Mitteilung, daß Sie den Speicher als lineares Bitfeld adressieren möchten. Ist der Modus auf diese Weise initialisiert, sollte anschließend der Bildschirmspeicher komplett gelöscht werden. An dieser Stelle kommt das Map-Mask-Register (02h) des Sequenzers ins Spiel. In diesem Register korrespondieren die einzelnen Bits jeweils mit der entsprechenden Plane beziehungsweise Latch: Bit 0 steuert den Schreibzugriff auf Plane 0, Bit 1 auf Plane 1 und so weiter. Ein gesetztes Bit zeigt hier an, daß bei einem Schreibzugriff seitens der CPU der Inhalt der entsprechenden Latch in die

auch wenig Sinn ergeben würde).

Selektiert man alle vier Planes, so sind bereits die Voraussetzungen für das Kopieren und Füllen von Bildschirmen geschaffen. Nun müssen noch 64 000 Byte (256 000 Byte Bildschirmspeicher / 4 Planes) auf 0 gesetzt werden, da an dieser Stelle der gesamte Speicher zu löschen ist. Doch Vorsicht: Dies ist nur in der Initialisierungsprozedur ohne weiteres realisierbar, da hier die VGA-Karte und die Latches automatisch Standardwerte erhalten (alles gelöscht).

Das Konzept der Bildschirmseiten wurde bereits angesprochen, die Realisierung findet sich in der Prozedur *Switch* in »s_unit.pas« (Listing 1). Sie schalten hier auf die als *page* übergebene Seite um, die somit sichtbar ist. Zunächst errechnet das Programm die Startadresse innerhalb des Bildschirmspeichers, also den Offset zu A000:0000. Die Seitennummer wird mit 16 000, der Bildschirmgröße, multipliziert, da die Seiten im Speicher unmittelbar aufeinanderfolgen. Diese Adresse läßt sich nun direkt an den CRTC in dessen Register 0ch (High-Byte) und 0dh (Low-Byte) übergeben.

Diese Umschaltung sollte man immer mit dem vertikalen Retrace synchronisieren, um Flackern zu vermeiden. Sie erreichen dies durch einen vorangestellten Aufruf der Prozedur *WaitRetrace*. Diese Prozedur fragt lediglich ständig das Bit 3 des Statusregisters 3dah ab.

Zunächst wird – für den Fall, daß sich der Kathodenstrahl bereits fast wieder auf dem Bildschirm befindet und somit die Zeit zum Umschalten zu kurz wäre – auf ein gelöscht Bit gewartet (Kathodenstrahl baut Bild auf, Schleife @1), danach auf ein Eintreten des Strahls in den Retrace, also den Rücklauf (Bit 3 = 1). Damit ist ein einfacher *WaitRetrace* auch schon fertig, es werden erst dann noch Erweiterungen notwendig, wenn der Bildaufbau etwas komplizierter ausfällt und seine Arbeit nicht mehr in 1/70 Sekunde beendet. In diesem Fall kann es zu Verschiebungen kommen, die dazu führen, daß manchmal auf einen physikalischen Bildaufbau gewartet wird, in einer anderen Situation (zum Beispiel durch eine gerade anfallende Soundberechnung) jedoch auf zwei Bildzyklen, also 1/35 Sekunde, was sich in deutlich sichtbaren Ruckeln äußern kann. Um diesen unerwünschten Effekt zu vermeiden, benötigen Sie einen Timer, der dafür sorgt, daß zwischen zwei Frames (Bildschirmseitenwechseln) immer die glei-

jeweilige Plane kopiert wird. Da dieses Register im Gegensatz zum Leseauswahl-Register für alle vier Planes je ein Bit bereitstellt, ist es hier gestattet, mehrere Planes gleichzeitig anzuwählen (was bei einem Lesebefehl

che Zeit vergeht, beispielsweise 1/35 oder 1/23 Sekunde (drei physikalische Bildzyklen beziehungsweise Rückläufe). Dies ist durch Verwendung des Kassetteninterrupts 15h erreichbar. Kassettenzugriffe steuert dieser Interrupt schon lange nicht mehr, dafür enthält er einen sehr komfortabel zu programmierenden Timer, der theoretisch eine Genauigkeit von 1µs zuläßt, leider jedoch erst ab knapp 1 ms zuverlässig funktioniert. Das ist für Sie aber nicht weiter tragisch, da der abzuwartende Zeitraum im Bereich von 14 ms (70 Hz Bildwechselfrequenz, also 70 neu berechnete Bilder pro Sekunde) bis 42 ms (35 Hz) reicht.

Dieser Timer läßt sich nun über Funktion 83h (übergeben in AH, AL = 0!) derart programmieren, daß er nach der in CX:DX in µs übergebenen Zeit ein beliebiges Flag (Adresse übergeben in ES:BX) auf 128 setzt. Dies darf beispielsweise auch eine Pascal-Variable sein, was die Abfrage sehr einfach hält.

Ein Problem zeigt sich jedoch, wenn eine andere Prozedur (zum Beispiel Soundberechnung) Interrupt 8 (den eigentlichen Zeitgeber auch des Interrupts 15h) umprogrammiert. In diesem Fall läßt sich die gezeigte Methodik nicht mehr zuverlässig verwenden. Das Problem ist allerdings sehr einfach in den Griff zu bekommen: Man läßt das Timeflag einfach direkt vom neuen Interrupt-8-Handler setzen, der seine Frequenz ja kennt. So entfällt die Neuprogrammierung des Interrupts 15h, das Programm muß lediglich das Timeflag wieder auf 0 setzen, um den neuen Zyklus zu starten. Um zu kontrollieren, ob die gewünschte Zeit bereits verstrichen ist, genügt es in beiden Fällen, das Flag so lange abzufragen, bis es den Wert 128 annimmt.

Eine solche Abfrage ist in der Prozedur *Mod_WaitRetrace* bereits integriert, die sich in der Unit »modplay.pas« in dieser Ausgabe befindet. Dieses Verfahren wird jedoch nur gebraucht, wenn der Interrupt 8 noch anderen Zwecken dient.

Die Hauptschleife – A und O eines Demos

Die Hauptschleife, der wichtigste Teil eines Demos, wird ständig bis zu einem Abbruchkriterium (zum Beispiel Tastendruck) durchlaufen und ist prinzipiell wie folgt aufgebaut:

REPEAT

Hintergrund darstellen
Sprites und ähnliches darstellen
auf Retrace warten
auf fertige Seite umschalten

UNTIL Abbruch

Der erste Punkt (Hintergrund darstellen) läßt sich durch einzelnes Zurückkopieren des Sprite-Hintergrundes erreichen. Dies erfordert jedoch einen enormen Verwaltungsaufwand. Zudem ist diese Technik gerade bei vielen Sprites, wie sie zum Beispiel bei einem Scroller auftreten, langsamer als die eigentlich naheliegende Lö-

sung: Kopieren des Hintergrundbildes von einer ansonsten nicht benötigten Bildschirmseite (2 oder 3), in die er geladen wurde, auf die aktuelle Bildschirmseite (0 oder 1). Dazu dient die Prozedur *CopyScreen*, die allgemein die in *qpage* übergebene Seite auf die in *zpage* kopiert. Setzen Sie hier Hintergrundseite und die aktuelle Grafikseite (*PAkt*) ein, wird der Hintergrund neu dargestellt.

Die Prozedur ruft zunächst *enable4* auf, die nichts weiter tut, als alle vier Planes zum Schreiben auszuwählen (Map-Mask-Register) und den Schreibmodus 1 einzuschalten. Damit wird zugleich der schnelle Zugriff auf vier Byte Bildschirmspeicher erreicht. Nun kopiert das Programm 16 000 Byte in einer einfachen Schleife vom Offset der *qpage* zum Offset der *zpage*. Wichtig ist hier, wie auch in allen anderen Assembler-Prozeduren, daß Sie das DS-Register sichern, das prinzipiell in jeder Kopierschleife benötigt wird, und es natürlich am Ende der Prozedur wieder restaurieren.

Ein wichtiger Teil jedes Grafikprogramms (Demo, Spiel) sind die Sprites. Ein Sprite ist genaugenommen ein frei beweglicher Bildschirmausschnitt, der jedoch einige Besonderheiten aufweist. Zum einen besitzt ein Sprite gewöhnlich durchsichtige Stellen, sei es im Innern oder auch am Rand, wo die Form selten exakt rechteckig ist und somit mit durchsichtigen Pixeln aufgefüllt wird. Zum anderen muß das entsprechende Programm das sogenannte »Clipping« unterstützen. Hierunter versteht man das Abschneiden des am Bildschirmrand verschwindenden Sprite-Randes, der bei einer einfachen Kopierschleife in die nächste Zeile rutschen würde. Hinzu kommt die spezielle Speicheraufteilung dieses Grafikmodus. Ein Sprite darf zum Beispiel zehn Pixel breit sein, wodurch die ersten beiden Plane-Datenbereiche drei Pixel umfassen und die letzten beiden nur zwei. Die einfachste Technik zum Lesen und Schreiben von Sprites ist, immer ein Pixel weiter zu gehen und die jeweilige Plane jedesmal zu selektieren. Dies ist jedoch sehr zeitaufwendig, da die VGA-Karte eine gewisse Zeit benötigt, um diese Umschaltung vorzunehmen. Daher ist es günstiger, die Planes nur viermal pro Sprite zu setzen und dann jeweils diese eine Plane komplett zu kopieren. Die Prozeduren *GetSprite* und *PutSprite* aus der Unit »s_unit.pas« (Listing 1), auf die im folgenden Bezug genommen wird, werden ähnlich wie die Turbo-Pascal-Äquivalente *GetImage* und *PutImage* aufgerufen. Zur x- und y-Position kommt hier allerdings noch die Bildschirmseite (*pg*) hinzu, *dx* und *dy* sind Breite beziehungsweise Höhe des Sprites. Als Variable zur Sprite-Datenübergabe dient hier allerdings kein einfacher Pointer, sondern ein Record des Typs *SpriteTyp*, der neben einem Zeiger auf die eigentlichen Sprite-Daten (*Adr*) noch die an *GetSprite* übergebenen Maße des Sprites enthält. Nach dem Setzen einiger Variablen und dem Berechnen der Ziel-

adresse bestimmt diese Prozedur zunächst die Breitenverteilung auf die einzelnen Planes (in der Variablen *ppp* (Pixel pro Plane) gespeichert). Die Sprite-Breite ist in den einzelnen Planes unterschiedlich: Hier sind die Planes 0, 1 und 3 jeweils drei Pixel breit und Plane 2 vier. Zunächst erhalten alle Planes die gleiche Anzahl, auf die dann noch der jeweils überstehende Teil (maximal ein Pixel) addiert wird. Danach folgt direkt die Kopierschleife. Sie beginnt mit dem Label *@lplane*, das die äußerste Schleife kennzeichnet, die nur einmal pro Plane zum Einsatz kommt. Hier ist wichtig, daß man möglichst viele Berechnungsschritte, wie die Sprungweite zum nächsten Zeilenanfang (*Skip*), weit außen in selten durchlaufenen Schleifen ausführt. Das wäre in diesem Fall die Schleife *lplane*. Sie greift auch auf das Feld *ppp* zurück und lädt pro Plane jeweils die »Anzahl« zu kopierender Bytes.

Die am häufigsten durchlaufene Schleife ist auf ein simples *rep movsb* geschrumpft, und auch die für die einzelnen Zeilen zuständige Schleife *lcopy* ist auf ein Minimum (Weiterbewegen des Quellzeigers auf nächste Zeile, CX neu laden) reduziert. An dieser Stelle sei angemerkt, daß die Verwendung von *movsb* nicht ohne Grund geschah. Ein *movsw* führt auf den meisten Computern beziehungsweise Grafikkarten zu falschen Resultaten, daher ist es besser, den langsamen, aber sicheren Weg zu gehen und byteweise zu kopieren.

Zum Schluß der *lplane*-Schleife wird noch die Plane-Nummer weitergeschaltet und im Fall eines Überlaufs der Zieloffset erhöht, da man sich dann bereits im nächsten Vier-Pixel-Block befindet. Damit ist die Schleife abgeschlossen, sie endet nach vier Durchläufen (gezählt in *Planeindex*), das Sprite liegt dann im Speicher vor.

Die *PutSprite*-Prozedur arbeitet zum großen Teil genauso wie *GetSprite*, das Kopieren findet allerdings vom Haupt- in den Bildschirmspeicher statt und nicht umgekehrt. Nun stellen jedoch die Bildschirmränder ein Problem dar, die beim Laden eines Sprites keine so wichtige Rolle spielten, da in diesem Fall die Koordinaten meist schon bei der Programmierung feststehen. Anders ist dies beim Setzen, denn hier ist ein glattes Verschwinden der Sprites am Bildschirmrand unbedingt erforderlich. Man denke nur an ein Action-Spiel, bei dem Raumschiffe in das Bild hinein- und wieder herausfliegen. Auch bei einem animierten Grafikdemo, dessen Laufschrift – soll sie denn etwas fantasievoller sein – sich nur mit Sprites realisieren läßt, kommt man nicht umhin, sie am Bildschirmrand abzuschneiden und so in das Bild hineinzufahren. Würde man sich in der Kopierschleife an dieser Problematik nicht stören, würden Teile des Sprites in unerwünschte Bereiche rutschen, zum Beispiel in die nächste Zeile. Darum muß eine Abfrage erfolgen, die sicherstellt, daß nur der sichtbare Teil auf den Bildschirm gelangt (Clipping).

Am rechten und unteren Rand ist es relativ einfach, Teile auszulassen, da hier nur die Sprite-Breite beziehungsweise -Höhe vermindert und der Quellzeiger nach jeder Zeile (Clipping am rechten Rand) beziehungsweise nach jeder Plane (unterer Rand) um den zu überspringenden Betrag erhöht werden muß. Der Zeiger steht somit wieder auf der nächsten Zeile oder Plane und gibt die richtigen Daten aus.

Am linken und oberen Rand funktioniert das Ganze praktisch genauso, jedoch wird der Zeiger vor der entsprechenden Zeile oder Plane erhöht. Beim linken und rechten Rand taucht allerdings auch noch das Problem auf, daß der zu überspringende Teil je nach Plane unterschiedlich ist, da diese jeweils eine andere Breite aufweisen (siehe *GetSprite*, *ppp*-Variable). Beachten Sie bei der Prozedur *PutSprite* noch, daß die Auswahl der gerade zu beschreibenden Plane etwas anders erfolgt als bei *GetSprite*. Zum einen benutzen Sie Port 3c4h, zum anderen korrespondieren in diesem Fall die übergebenen Bits mit den entsprechenden Planes, so daß das Weiterschalten auf die nächste Plane durch Rotieren des entsprechenden Bytes (*PlaneMask*) um eine Einheit nach links rotiert.

Skalierung von Sprites

Neben dem Clipping gibt es bei Sprites ein weiteres wichtiges Gestaltungsmittel: die Skalierung. Dies wird meist über sogenannte Texturen gelöst, die über mathematische Formeln jedem Quellpunkt einen Zielpunkt zuordnen und somit ein Sprite in jede beliebige Form bringen können (unter anderem auch auf drehende dreidimensionale Körper). Dieses Verfahren ist jedoch schon bei den verhältnismäßig einfachen Skalierungen sehr rechenintensiv und somit für schnelle Animationen nur bedingt geeignet. Vom Prinzip her ähnlich, von der Umsetzung her jedoch sehr verschieden arbeitet das in der Prozedur *PutTexture* verwendete Verfahren. Da hier ebenfalls der Rechenaufwand recht groß ist und eine Skalierung bei vielen Sprites weitgehend entfällt, wurde eine neue Prozedur geschrieben, die als Ausgleich kein Clipping unterstützt. *PutTexture* errechnet, in welchem Abstand Zeilen herauszunehmen sind, deren Darstellung dann einfach unterbleibt. Dazu verwenden Sie eine Festkommarechnung, die zwar nicht so genau, aber um ein Vielfaches schneller als die Gleitkomma-Arithmetik arbeitet.

Die Variable *Add_Y* enthält zu diesem Zweck den Nachkommateil, der nach jeder Zeile auf den Nachkommateil der (Sprite-bezogenen) y-Koordinate (*Rel_Y*) aufaddiert wird. Tritt ein Überlauf ein, muß die Prozedur die Zeile herausnehmen und den Nachkommateil auf den Bereich von 0 bis 100 reduzieren, um wieder nur den Nachkommateil darzustellen.

Das Überspringen der errechneten Zeilen ist sehr einfach: Nachdem die Zeile dar-

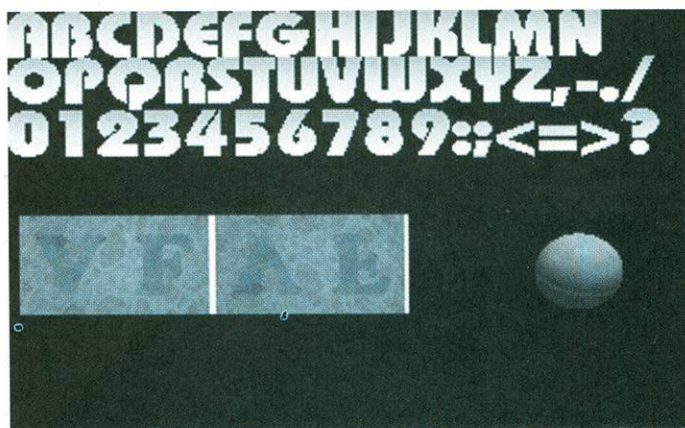


Bild 2. In »pfont.gif« sind Zeichensatz, Textur und Ball enthalten. Mit einem Grafikprogramm lassen sich beliebige andere Kreationen erzeugen.

gestellt ist, wird sie sofort durch die folgende Zeile überschrieben, da wir den Zielzeiger wieder eine Zeile zurückbewegt haben. Natürlich läßt es sich auch realisieren, die Zeilen gar nicht erst darzustellen – der Verwaltungsaufwand ist dann allerdings immens und der entstehende Vorteil zweifelhaft, da gerade bei großflächigen Sprites die Darstellungs geschwindigkeit stark schwanken würde. Zur Verwendung der PutTexture-Prozedur sei gesagt, daß zu den Koordinaten noch der Skalierungsfaktor in Prozent (–100 bis 100) angegeben wird, wobei negative Werte eine Spiegelung in y-Richtung bewirken, was rotierende Logos erlaubt. In diesem Fall ist prinzipbedingt ein Löschen des Sprites durch Angabe einer Skalierung von 0 Prozent nicht unterstützt, vielmehr werden alle Zeilen übereinandergeschrieben, was eine minimale Sprite-Höhe von einem Pixel ergibt.

Mit diesen Routinen lassen sich Sprites im eigentlichen Sinn bereits komfortabel erzeugen und über den Bildschirm bewegen. Sie bieten sich aber auch geradezu zum Erzeugen eines »Scrollers« an, das heißt einer Laufschrift, die von rechts nach links durch das Bild läuft und die gewünschte Botschaft übermittelt (unter anderem auch die beliebten »Greetings«).

Eine einfache Laufschrift ist sehr schnell realisierbar, indem man eine Prozedur schreibt, die einen Bildausschnitt um vier Punkte nach links bewegt und rechts eine vier Pixel breite Spalte anhängt, wobei dem Programmierer der Schreibmodus 1 (vier Planes auf einmal) sehr zugute kommt. Dieses Verfahren besitzt jedoch den Nachteil, daß erstens der Zeichensatz ständig im VGA-RAM vorhanden sein muß, der schnell mit Hintergrundbildern und ähnlichem gefüllt ist, und sich zweitens viele interessante Effekte so nicht umsetzen lassen. Zum Erzeugen einer Sinuslinie, auf der sich der Scroller entlangbewegt, oder zum Bewegen über den Hintergrund hinweg (durchscheinende Buchstaben) kann man nur Sprites verwenden. Dazu dienen in »s_unit.pas« die Prozeduren *Init_Scroller*, die nur Variablen initialisiert und die Sprites anhand einer Koordinatentabelle (*C_Pos*, mit Hilfe jedes gängigen Malprogramms erzeugbar) in den Hauptspeicher lädt, und die Prozedur *Draw_Scroller*. Letztere bewegt die beiden Textzeiger (*txtnr1* : *txt_pos1* und *txtnr2* : *txt_pos2*) fort,

angezogen, die eine Sinuskurve der Amplitude 30 enthält, deren Werte man nur noch auf die feste y-Koordinate des Scrollers (*Sc_Y*) addiert. Die Sprites bewegen sich nun in einem Bereich von 30 Punkten herauf und herunter. Mit anderen Daten ließe sich natürlich auch beispielsweise eine Dreieck- oder Rechteckkurve erzeugen.

Die Tabelle kommt zum Einsatz, um die (äußerst langsame) Echtzeitberechnung der Sinuswerte zu umgehen. Den Text selbst entnimmt das Programm dem Array *txt*, das, da Turbo Pascal nur 255-Zeichen-Strings zuläßt, aus mehreren (*maxtext*) Zeichenketten besteht, die direkt (ohne Lücke!) nacheinander ausgegeben werden. Der Zeichensatz ist mit jedem beliebigen Malprogramm erzeug- und als Gif-Grafik importierbar, wobei die Koordinaten der linken oberen Ecken und die Breiten der Zeichen in die *C_Pos*-Tabelle (in ASCII-Reihenfolge) einzutragen sind.

Zur Gestaltung von Sprites und Hintergrundbildern bedient man sich häufig eines speziell auf die Demoprogramm zugeschnittenen Editors, der im vorliegenden Fall jedoch erst noch geschrieben werden müßte und auch so viele Leistungsmerkmale benötigt, die in Malprogrammen ohnehin zuhauf vorkommen. Daher beschreiten wir einen anderen Weg: Der Demodesigner malt seine Bilder mit einem beliebigen Grafikprogramm (oder läßt sie per Raytracer berechnen) und speichert sie als Gif-Bild, das mit Hilfe der ebenfalls in »s_unit.pas« befindlichen Gif-Routinen verwendet wird. Die Sprites sind anhand von *GetSprite* in den Hauptspeicher ladbar und stehen als Sprite, Texture oder Zeichensatz bereit. *CopyScreen* kopiert schließlich die Hintergrundbilder auf die gewünschte Seite.

Das Laden von Grafiken ist immer sehr zeitkritisch, besonders bei gepackten Formaten wie etwa Gif – daher sollte es generell am Anfang des Programms erfolgen. Ein später benötigtes Bild läßt sich zum Beispiel auch auf einer nicht verwendeten Bildschirmseite unterbringen.

Um die hohe Geschwindigkeit beizubehalten, kommen in unseren Prozeduren keine Dateifunktionen von Turbo Pascal zum Einsatz, die bei einzeln zu ladenden Bytes zu langsam arbeiten und eine großangelegte Pufferung andererseits nicht sinnvoll ist. Da es bei Pascal im Gegensatz zu C keinen Weg gibt, nach einem Öffnen der Datei deren Handle zu erfahren, mußten auch

Routinen zum Öffnen und Schließen geschrieben werden, die den Handle in der gleichnamigen Variable speichern.

GifRead liest »n« Byte aus der Datei, und *GifSeekDelta* bewegt den Dateizeiger um *delta* vorwärts. *ShiftPal* gleicht das Gif-Palettenformat (Bits 2 bis 7) an das VGA-Format (Bits 0 bis 5) durch eine Rechtsrotation um zwei Bit an. Die eigentliche Laderoutine öffnet zunächst die Grafik und überliest den Kopf, der in diesem Fall zum größten Teil unwichtig ist, da ohnehin nur Bilder im Format 320 x 200 bei 256 Farben erlaubt sind. Da eine Darstellung anderer Formate mit dieser Unit ausscheidet, müssen Sie solche Bilder zuerst mit einem Malprogramm konvertieren.

Außerdem kümmert sich das Programm an dieser Stelle um die Reservierung von Speicher für eine Zwischenpufferung des Bildes. Diese ist wiederum notwendig, da ein Setzen einzelner Punkte sehr lange dauert. Daher laden Sie zunächst das ganze Bild komplett im gewohnten 320 x 200-Format in den Hauptspeicher, von wo aus es Plane-weise in das VGA-RAM gelangt. Am Ende folgt die Freigabe des verwendeten Speichers.

Der folgende Assembler-Teil liest das Bild ein, genauere Informationen zum Aufbau des Headers und der eigentlichen Daten (gepackt nach dem LZW-Verfahren) finden sich zur Genüge in der entsprechenden Fachliteratur.

Danach wird nur noch das temporär im Arbeitsspeicher vorliegende Bild in den Bildschirmspeicher übertragen und *temppic* wieder freigegeben.

Das Programm »simple_s.pas« (Listing 2) ist eine Demonstration der Fähigkeiten der Unit und stellt ein Demo beziehungsweise Intro dar, das Scroller, Texture sowie Einzel-Sprites kombiniert und – in Verbindung mit der Unit »modplay.pas« aus dieser Ausgabe – gleichzeitig eine MOD-Datei abspielt. Dazu ist lediglich die Compiler-Variable *Sound* am Anfang zu setzen und der Unit-Name zur Uses-Anweisung hinzuzufügen.

Beim Einsatz des Demoprogramms »simple_s.pas« sowie der damit verbundenen Units »s_unit.pas« und »modplay.pas« sollten Sie den darauf aufbauenden Applikationen ausreichend Hauptspeicher zur Verfügung stellen und auch weitgehend auf Speichermanagement-Software (beispielsweise Qemm) oder nicht unbedingt erforderliche Treiber verzichten, da es sonst zu Interrupt-Kollisionen oder Beeinträchtigungen im Ablauf kommen könnte. Weiterhin erwartet »simple_s.pas« im aktuellen Verzeichnis die beiden Grafikdateien »phint.gif« (Hintergrundmotiv, Bild 1) und »pfont.gif« (Zeichensatz, Textur und Ball, Bild 2) sowie die vierstimmige MOD-Datei »hurrica2.mod«, die Sie entweder auf der Diskette zu dieser Ausgabe vorfinden oder durch eigene Dateien des entsprechenden Formats ersetzen können.

(Matthias Rasch/bj)

Listing 1: s_unit.pas

```

1: {Programm: s_unit.pas
2: Funktion: Unit zur Realisierung von
3:   Bild-, Animations- und
4:   Sound-Demos
5: Sprache : Turbo Pascal ab 6.0
6: Autor   : Matthias Rasch
7: (c) 1993 DMV Daten- u. Medienverlag}
8:
9: Unit s_unit;
10: Interface
11: uses Crt,dos;
12: Const Maxsprites=2;
13:   o_dtx=4; o_dty=6;
14:   dly=35000;
15:   dlyhigh=$0;
16: Type SpriteTyp=Record
17:   Adr:Pointer;
18:   dtx,dty:Word;
19: End;
20: const
21:   sin_sng:Array[0..319] of shortint =
22:   (1,1,2,2,3,4,4,5,5,6,6,7,8,8,9,9,10,
23:   10,11,11,12,13,13,14,14,15,15,16,16,
24:   17,17,18,18,19,19,20,20,21,21,22,
25:   22,22,23,23,24,24,25,25,26,26,
26:   26,26,27,27,27,28,28,28,28,29,29,
27:   29,29,29,29,29,30,30,30,30,30,30,
28:   30,30,30,30,30,30,30,30,30,30,30,
29:   29,29,29,29,29,29,28,28,28,28,27,
30:   27,27,27,26,26,26,26,25,25,24,24,
31:   24,23,23,22,22,22,21,21,20,20,19,19,
32:   19,18,18,17,17,16,16,15,15,14,14,13,
33:   13,12,11,11,10,10,9,9,8,8,7,6,6,5,5,
34:   4,4,3,2,2,1,1,0,-1,-1,-2,-2,-3,-4,-4,
35:   -5,-5,-6,-6,-7,-8,-8,-9,-9,-10,-10,
36:   -11,-11,-12,-13,-13,-14,-14,-15,-15,
37:   -16,-16,-17,-17,-18,-18,-19,-19,-19,
38:   -20,-20,-21,-21,-22,-22,-22,-23,-23,
39:   -24,-24,-24,-25,-25,-25,-26,-26,-26,
40:   -26,-27,-27,-27,-27,-28,-28,-28,-28,
41:   -29,-29,-29,-29,-29,-29,-30,-30,-30,
42:   -30,-30,-30,-30,-30,-30,-30,-30,-30,
43:   -30,-30,-30,-30,-30,-30,-30,-29,
44:   -29,-29,-29,-29,-29,-28,-28,-28,
45:   -28,-27,-27,-27,-27,-26,-26,-26,-26,
46:   -25,-25,-25,-24,-24,-24,-23,-23,-22,
47:   -22,-22,-21,-21,-20,-20,-19,-19,-19,
48:   -18,-18,-17,-17,-16,-16,-15,-15,-14,
49:   -14,-13,-13,-12,-11,-11,-10,-10,-9,
50:   -9,-8,-8,-7,-6,-6,-5,-5,-4,-4,-3,-2,
51:   -2,-1,-1,0);
52:
53: var time:Integer;
54: Const C_Pos:Array['..'Z',0..3]
55:   of Word =
56:   ((256,22,11,22),(268,22,10,22),
57:   (279,22,9,22),(289,22,16,22),
58:   {,-/})
59:   (0,47,19,22),(24,47,11,22),
60:   (42,47,19,22),(64,47,17,22),
61:   {0-3}
62:   (83,47,20,22),(107,47,16,22),
63:   (127,47,18,22),(148,47,17,22),
64:   {4-7}
65:   (168,47,18,22),(190,47,18,22),
66:   (209,47,9,22),(219,47,9,22),
67:   {8-;})
68:   (230,47,18,22),(250,47,18,22),
69:   (270,47,18,22),(289,47,18,22),
70:   {<-?}
71:   (0,0,19,22),
72:   {@}
73:   (0,0,19,22),(21,0,19,22),
74:   (42,0,16,22),(61,0,20,22),
75:   {a-d}
76:   (83,0,16,22),(101,0,14,22),
77:   (117,0,25,22),(145,0,19,22),
78:   {e-h}

```

```

79: (166,0,8,22),(177,0,13,22),
80: (193,0,19,22),(214,0,14,22),
81: {i-l}
82: (229,0,28,22),(260,0,20,22),
83: (0,22,25,22),(27,22,18,22),
84: {m-p}
85: (47,22,25,22),(74,22,19,22),
86: (94,22,15,22),(111,22,15,22),
87: {q-t}
88: (127,22,19,22),(147,22,20,22),
89: (168,22,29,22),(199,22,19,22),
90: {u-x}
91: (220,22,17,22),(238,22,18,22));
92: {y-z}
93: maxtext=3;
94: txt:Array[1..maxtext] of String=('D'+
95: 'IES IST EIN DEMO DER VISION FACTOR'+
96: 'Y / DMV VERLAG UM DIE FAEHIGKEITEN'+
97: 'DER TURBO PASCAL UNIT - S-UNIT ZU'+
98: 'DEMONSTRIEREN          DIESER SCR'+
99: 'OLLER WURDE MIT HILFE DER ',
100: 'PROZEDUR DRAW-SCROLLER REALISIERT,'+
101: 'WOBEI DIE SINUS-TABELLE VERWENDET'+
102: 'WURDE, UM DEN SCHLANGENVERLAUF ZU'+
103: 'ERZEUGEN ..... DAS DREHENDE'+
104: 'LOGO IST EINE MIT PUT-TEXTURE ERZ'+
105: 'EUGTE TEXTUR, WOBEI SOWOHL GROESSE'+
106: 'NAENDERUNG ALS AUCH Y-SPIEGELUNG B'+
107: 'ENUTZT WERDEN ',
108: '/ SIEHE HAUPTSCHLEIFE / DIE VI'+
109: 'ER BAELE AM BODEN STELLEN BEI VER'+
110: 'WENDUNG DER MOD-UNIT AUS DIESER AU'+
111: 'SGABE DIE ANSTEUERUNG DER VIER KAN'+
112: 'AELE GRAFISCH DAR UND SPRINGEN SOM'+
113: 'IT IM TAKT DER MUSIK ..... SO, GE'+
114: 'NUG, DER SCROLLER STARTET JETZT WI'+
115: 'EDER VON VORN          ');
116:
117:   sc_step=4;
118:   sc_abst=2;
119: {x-Abstand der Buchstaben}
120:   sc_space=10;
121:   sc_y:Word=33;
122: Var txtnr1,
123:   txtnr2,
124:   Txt_Pos1,
125: {Zeiger, zum Text auf dem Bildschirm}
126:   Txt_Pos2:Word;
127:   sc_x1,
128: {X-Koordinate des ersten und letzten
129:   Buchstabens}
130:   sc_x2:Integer;
131: {müssen auch negativ werden können}
132:   sc_i,
133: {Schleifenindex}
134:   sc_txtnr,
135: {Zaehler für Textnr}
136:   sc_x:Integer;
137: {temporärer x-Merker}
138:   sc_numb:Integer;
139: {temporärer Spritenummernzähler}
140:   Chars:Array['..'Z'] of SpriteTyp;
141:
142: Var palette:Array[0..767] of Byte;
143: Sprite:Array[1..maxsprites] of
144:   SpriteTyp;
145: texture:Array[1..1] of SpriteTyp;
146: pakt:Byte;
147: {aktuelle (unsichtbare Seite)}
148:   dir,
149:   i:Integer;
150:   timeflag:Byte;
151: {
152: für Zeitmessung zwischen der
153: Spritedarstellung

```

»s_unit.pas« läßt Intros und Demos mit animierter Laufschrift, Grafikeinbindung und Soundausgabe zum Kinderspiel werden.


```

154: }
155:     counthi,countlo:Word;
156:     gi : integer;
157:     gidir:Integer;
158:     page : pointer;
159: const clr=256;
160: {gif}
161:     eof=257;
162: Var Handle:Word;
163:     Puf:Array[0..767] of Byte;
164: PufInd:Word;
165: Stack:Array[0..1280] of byte;
166: ab_prfx,ab_tail:Array[0..4096]
167:     of word;
168: Byt:Byte;
169: free,breite,max,
170:     stackp,restbits,restbyte,
171:     sonderfall,
172: code,old_code,readbyt,bits,
173:     bits2get:Word;
174: lbyte:Word;
175: mask:Word;
176: zseg,zofs,
177: GifName:String[15];
178: tempPic:Pointer;
179: Procedure Init_VGA;
180: Procedure Switch(page:Word);
181: Procedure CopyScreen(qpage,
182:     zpage:Word);
183: Procedure GetSprite(pg,x,y,dtx,
184:     dty:word;var zsprite:SpriteType);
185: Procedure PutTexture(pg,x,y,
186:     scale_y:Integer;gsprite:spriteType);
187: Procedure PutSprite(pg,x,y:Integer;
188:     gsprite:spriteType);
189: Procedure WaitRetrace;
190: Procedure Init_Scroller;
191: Procedure Draw_Scroller;
192: Procedure LoadGif(name:String;
193:     bpos:word);
194: procedure disable4;
195:
196: Implementation
197: function get_Point(x,y : integer;
198:     Page : word) : byte;
199: var plane : byte;
200:     offs : word;
201: begin;
202:     offs := y*80+(x Div 4);
203:     plane := x mod 4;
204:     port[$3ce] := 4;
205:     port[$3cf] := plane;
206:     offs := offs + page*16000;
207:     get_Point := mem[$A000:offs];
208: end;
209: procedure Point(x,y : integer; col :
210:     byte); assembler;
211: var plane : byte;
212: asm
213:     mov ax,y
214:     mov bx,80
215:     mul bx
216:     mov di,ax
217: { y * 80 }
218:     mov ax,x
219:     shr ax,2
220:     add di,ax
221: { + x DIV 4 }
222:     mov ax,16000
223:     mov bh,0
224:     mov bl,pakt
225: { <- PAKT ! }
226:     mul bx
227:     add di,ax
228: { + 16000 * Pakt }
229:     mov ax,x
230: { Plane des Punktes bestimmen }
231:     mov bx,4
232:     div bl
233:     mov cl,ah

```

```

234:     mov dx,1
235:     shl dx,cl
236:     mov plane,d1
237:     mov dx,$3c4
238: { Plane einschalten }
239:     mov al,2
240:     mov ah,plane
241:     out dx,ax
242:     mov si,$A000
243: { Punkt Adressieren & Setzen }
244:     mov es,si
245:     mov al,col
246:     stosb
247:     inc dx
248: { Alle Planes ein }
249:     mov al,15
250:     out dx,al
251: end;
252: Procedure SetPal;assembler;
253: asm
254:     push ds
255:     pop es
256:     mov dx,offset palette
257:     mov ax,1012h
258:     xor bx,bx
259:     mov cx,0100h
260:     int 10h
261: End;
262: Procedure Init_VGA;assembler;
263: asm
264:     mov ax,0013h
265: { Den normalen Mode 13h setzen }
266:     int 10h
267:     mov dx,3c4h
268: {
269: Verknüpfung aufheben,
270: Einzelzugriff erlauben
271: }
272:     mov al,4
273:     out dx,al
274:     inc dx
275:     in al,dx
276:     and al,0f7h
277:     or al,4h
278:     out dx,al
279:     dec dx
280:     mov ax,0f02h
281:     out dx,ax
282:     mov ax,0a000h
283: { Bildschirmspeicher löschen }
284:     mov es,ax
285:     xor di,di
286:     xor ax,ax
287:     mov cx,8000h
288:     cld
289:     rep stosw
290:     mov dx,3d4h
291:     mov al,14h
292:     out dx,al
293:     inc dx
294:     in al,dx
295:     and al,0bfh
296:     out dx,al
297:     dec dx
298:     mov al,17h
299:     out dx,al
300:     inc dx
301:     in al,dx
302:     or al,40h
303:     out dx,al
304: End;
305: Procedure enable4;assembler;
306: asm
307:     mov dx,3c4h
308:     mov ax,0f02h

```

»s_unit.pas« läßt Intros und Demos mit animierter Laufschrift, Grafikeinbindung und Soundausgabe zum Kinderspiel werden.


```

309: out dx,ax
310: mov dx,3ceh
311: mov ax,4105h
312: out dx,ax
313: End;
314: procedure disable4; assembler;
315: asm;
316: mov dx,3c4h
317: mov ax,0f02h
318: out dx,ax
319: mov dx,3ceh
320: mov ax,4005h
321: out dx,ax
322: end;
323: Procedure Switch(page:Word);
324: assembler;
325: var ofs:Word;
326: asm
327: mov ax,16000
328: {Startadresse der Page ausrechnen}
329: mul page
330: mov dx,3d4h
331: {Port des Memory Controllers}
332: mov cl,al
333: {Low-Byte der Startadresse sichern}
334: mov al,0ch
335: {Register 0ch erhält High-Byte}
336: out dx,ax
337: mov ah,cl
338: {gesichertes Low-Byte zurück}
339: mov al,0dh
340: {und auf Register 0dh}
341: out dx,ax
342: End;
343: Procedure CopyScreen(qpage,
344: zpage:Word); assembler;
345: asm
346: call enable4
347: push ds
348: mov bx,16000
349: mov ax,qpage
350: mul bx
351: mov si,ax
352: mov ax,zpage
353: mul bx
354: mov di,ax
355: mov ax,0a000h
356: mov es,ax
357: mov ds,ax
358: mov cx,16000
359: rep movsb
360: pop ds
361: End;
362: Procedure GetSprite(pg,x,y,dtx,
363: dty:word;var zsprite:SpriteTyp);
364: var plane:Byte;
365: {aktuelle Bit-Plane}
366: planeindex:Byte;
367: {Zähler der bereits kopierten Planes}
368: ofs:Word;
369: Anzahl:Byte;
370: {Anzahl zu kopierender Bytes in einer
371: Zeile}
372: Skip:word;
373: {Anzahl bei Zeilenwechsel zu
374: überspringender Bytes}
375: ppp:Array[0..3] of Byte;
376: Begin
377: GetMem(zsprite.adr,dtx*dty);
378: ofs:=16000*pg+80*y+(x div 4);
379: Anzahl:=dtx div 4;
380: Plane:=x mod 4;
381: Skip:=0;
382: zsprite.dtx:=dtx;
383: zsprite.dty:=dty;
384: asm
385: call disable4
386: push ds
387: mov dx,03ceh
388: {Lesemodus 1 setzen}

```

```

389: mov ax,4105h
390: out dx,ax
391: mov di,word ptr zsprite
392: {Zieladresse laden in es:di}
393: mov ax,word ptr ds:[di+2]
394: mov di,word ptr ds:[di]
395: mov es,ax
396: mov ax,dtx
397: shr ax,2
398: mov byte ptr ppp[0],al
399: {ppp-Register initialisieren}
400: mov byte ptr ppp[1],al
401: {auf Anzahl mindestens zu kopierender
402: Bytes pro Plane}
403: mov byte ptr ppp[2],al
404: mov byte ptr ppp[3],al
405: mov cx,dtx
406: {
407: "überstehende" Pixel in ppp eintragen
408: }
409: and cx,3
410: je @nodbl
411: mov al,plane
412: {ah enthält hier garantiert 0}
413: mov si,ax
414: @mark_double_planes:
415: inc byte ptr ppp[si]
416: inc si
417: and si,3
418: {Überlauf vorbeugen}
419: loop @mark_double_planes
420: @nodbl:
421: mov planeindex,4
422: mov ax,0a000h
423: {Quellsegment nach ds}
424: mov ds,ax
425: @lplane:
426: mov cl,plane
427: {ch garantiert 0}
428: mov si,cx
429: {
430: si wird erst später wieder gebraucht
431: }
432: mov cl,byte ptr ppp[si]
433: {x-Zähler (cx) laden mit zugehöriger
434: ppp-Nummer}
435: mov anzahl,cl
436: mov al,80
437: stc
438: sub al,cl
439: mov byte ptr skip,al
440: mov si,ofs
441: {Quelloffset nach si}
442: mov ah,plane
443: {Plane auswählen}
444: mov al,04h
445: out dx,ax
446: mov bx,dty
447: {y-Zähler (bx) laden}
448: @lcopy:
449: rep movsb
450: {Copy-Schleife}
451: add si,skip
452: {Anfang nächste Zeile}
453: mov cl,Anzahl
454: {cx neu laden}
455: dec bx
456: {y-Zähler dekrementieren}
457: jne @lcopy
458: inc plane
459: {nächste Plane anwählen}
460: and plane,03h
461: mov cl,plane
462: mov ax,1
463: shr al,cl

```

»s_unit.pas« läßt Intros und Demos mit animierter Laufschrift, Grafikeinbindung und Soundausgabe zum Kinderspiel werden.


```

464: add ofs,ax
465: dec planeindex
466: jne @lplane
467: pop ds
468: End;
469: {asm}
470: End;
471: Procedure PutSprite(pg,x,y:Integer;
472:   qsprite:spritetyp);
473: var planeindex,
474:   planemask:Byte;
475:   Skip,ofs,plane,
476:   Anzahl,dtv:Word;
477:   quelle:Pointer;
478:   clip_lt,
479:   clip_rt:integer;
480:   clipakt_lt,
481:   clipakt_rt,
482:   clipall_lt,
483:   clipall_rt:Word;
484:   clip_dn,
485:   clip_up:Word;
486:   ppp:Array[0..3] of Byte;
487:   cpp:Array[0..3] of Byte;
488: {Skip pro Plane}
489: Begin
490:   if (x > 319) or (x+qsprite.dtx < 0)
491:   {Darstellung überflüssig ?}
492:   or (y > 199) or
493:   (y+qsprite.dty < 0) then exit;
494:   clip_rt:=0;
495:   clip_lt:=0;
496:   clip_dn:=0;
497:   clip_up:=0;
498:   clipall_rt:=0;
499:   clipall_lt:=0;
500:   clipakt_rt:=0;
501:   clipakt_lt:=0;
502:   with qsprite do begin
503:     if y+dtv > 200 then begin
504:       clip_dn:=(y+dtv-200);
505:       dtv:=200-y;
506:     End;
507:     if y<0 then begin
508:       clip_up:=-y;
509:       dtv:=dtv+y;
510:       y:=0;
511:     End;
512:     if x+dtv > 320 then begin
513:       clip_rt:=x+dtv-320;
514:       clipall_rt:=clip_rt shr 2;
515:       dtv:=320-x;
516:     End;
517:     if x<0 then begin
518:       clip_lt:=-x;
519:       clipall_lt:=clip_lt shr 2;
520:       plane:=4-(clip_lt mod 4);
521:       plane:=plane and 3;
522:       ofs:=16000*pg+80*y+((x+1)
523:         div 4) - 1;
524:       x:=0;
525:     End Else Begin
526:       plane:=x mod 4;
527:       ofs:=16000*pg+80*y+(x div 4);
528:     End;
529:   End;
530:   dtv:=qsprite.dty;
531:   Anzahl:=0;
532:   Skip:=0;
533:   Quelle:=qsprite.adr;
534:   cpp[0]:=0;cpp[1]:=0;
535:   cpp[2]:=0;cpp[3]:=0;
536: asm
537:   push ds
538:   call disable4
539:   mov dx,3c4h
540: {Portadresse laden}
541:   mov ax,0a000h
542: {Zieladresse nach es:di}
543:   mov es,ax

```

```

544:   mov ax,qsprite.dtx
545:   shr ax,2
546:   mov byte ptr ppp[0],al
547: {ppp-Register initialisieren}
548:   mov byte ptr ppp[1],al
549: {auf Anzahl mindestens zu kopierender
550:   Bytes pro Plane}
551:   mov byte ptr ppp[2],al
552:   mov byte ptr ppp[3],al
553:   mov cx,qsprite.dtx
554: {
555:   "überstehende" Pixel in ppp eintragen
556: }
557:   and cx,3
558:   je @nodbl
559:   mov al,byte ptr plane
560: {ah enthält hier garantiert 0}
561:   mov si,ax
562: @mark_double_planes:
563:   inc byte ptr ppp[si]
564:   inc si
565:   and si,3
566: {Überlauf vorbeugen}
567:   loop @mark_double_planes
568: @nodbl:
569:   mov cl,byte ptr clip_rt
570: {Anzahl abzuschneidender Pixel holen}
571:   and cl,03h
572: {(für rechten Rand)}
573:   je @nocpp
574:   xor si,si
575: {in cpp-Tabelle eintragen}
576: @lcpp:
577:   inc byte ptr cpp[si]
578:   inc si
579:   loop @lcpp
580:   jmp @nocpp_lt
581: {wenn rechts Clipping, kann links
582:   keines sein}
583: @nocpp:
584:   mov cl,byte ptr clip_lt
585: {das gleiche für den linken Rand}
586:   and cl,3h
587:   je @nocpp_lt
588:   mov si,3
589: @lcpp_lt:
590:   inc byte ptr cpp[si]
591:   dec si
592:   loop @lcpp_lt
593: @nocpp_lt:
594:   lds si,quelle
595: {Quelle nach ds:si}
596:   mov cx,plane
597: {Plane - Maske erstellen}
598:   mov ax,1
599:   shl ax,cl
600:   mov planemask,al
601:   shl al,4
602: {auch in oberes Nibble eintragen}
603:   or planemask,al
604:   mov planeindex,4
605: {4 Planes zu kopieren}
606: @lplane:
607:   mov cl,byte ptr plane
608: {ch garantiert 0}
609:   mov di,cx
610: {di wird erst weiter unten wieder
611:   gebraucht}
612:   mov cl,byte ptr ppp[di]
613: {x-Zähler (cx) laden mit zugehöriger
614:   ppp-Nummer}
615:   mov byte ptr anzahl,cl
616: {Skip jeweils neu ausrechnen}
617:   mov ax,80
618:   sub al,cl

```

»s_unit.pas« läßt Intros und Demos mit animierter Laufschrift, Grafikeinbindung und Soundausgabe zum Kinderspiel werden.


```

619: mov byte ptr skip,al
620: mov al,byte ptr cpp[di]
621: cmp clip_lt,0
622: je @rechts
623: add ax,clipall_lt
624: {clipall als Basis}
625: mov clipakt_lt,ax
626: sub anzahl,ax
627: {Anzahl zu kopierenden
628: Bytes reduzieren}
629: jmp @clip_rdy
630: @rechts:
631: add ax,clipall_rt
632: {das gleiche für rechts}
633: mov clipakt_rt,ax
634: @clip_rdy:
635: mov ax,anzahl
636: add ax,clipakt_rt
637: add ax,clipakt_lt
638: mul clip_up
639: add si,ax
640: {neuen Start bei oberem Clipping
641: festlegen}
642: mov cx,anzahl
643: or cl,cl
644: {Breite 0, dann Plane fertig}
645: je @plane_fertig
646: mov di,ofs
647: {Zieloffset nach di}
648: mov ah,planemask
649: {planemask auf bit [0..3] reduzieren
650: und setzen}
651: and ah,0fh
652: mov al,02h
653: mov dx,3c4h
654: out dx,ax
655: mov bx,dti
656: {x- und y-Zähler initialisieren}
657: @lcopy_y:
658: add si,clipakt_lt
659: add di,clipakt_lt
660: @lcopy:
661: lodsb
662: {Byte holen, wenn 0 überspringen}
663: or al,al
664: je @wert0
665: stosb
666: {ansonsten: setzen}
667: @entry:
668: loop @lcopy
669: add si,clipakt_rt
670: dec bx
671: je @plane_fertig
672: {y-Zähler = 0, dann nächste Plane}
673: add di,skip
674: {sonst auf nächsten Zeilenanfang
675: springen}
676: mov cx,Anzahl
677: {x-Zähler reinitialisieren,}
678: jmp @lcopy_y
679: @wert0:
680: inc di
681: {Sprite-Wert 0 : Zielbyte
682: überspringen}
683: jmp @entry
684: @plane_fertig:
685: mov ax,anzahl
686: {abgeschnittene Zeilen überspringen}
687: add ax,clipakt_rt
688: add ax,clipakt_lt
689: mul clip_dn
690: add si,ax
691: rol planemask,1
692: {nächste Plane maskieren}
693: mov cl,planemask
694: {plane 0 selektiert ?}
695: and cx,1
696: {(Bit 1 gesetzt), dann}
697: add ofs,cx
698: {Zieloffset erhöhen um 1 (Bit 1 !)}

```

```

699: inc plane
700: and plane,3
701: dec planeindex
702: {schon 4 Planes kopiert ?, dann Ende}
703: jne @lplane
704: pop ds
705: End;
706: {asm}
707: End;
708: Procedure PutTexture(pg,x,y,
709: scale_y:Integer;qsprite:spritetyp);
710: var planeindex,
711: planemask:Byte;
712: Skip,ofs,plane,
713: Anzahl,dti:Word;
714: quelle:Pointer;
715: ppp:Array[0..3] of Byte;
716: rel_y,
717: {Nachkommateil (2 Stellen) der rel.
718: y-Position}
719: add_y:Word;
720: {Nachkommawert der Addition}
721: direction:Integer;
722: Begin
723: rel_y:=0;
724: if (x + qsprite.dtx > 319)
725: or (x < 0)
726: or (y + qsprite.dty > 199)
727: or (y < 0) then exit;
728: add_y:=100-abs(scale_y);
729: if scale_y < 0 then direction:=-80
730: else direction:=80;
731: dti:=qsprite.dty;
732: plane:=x mod 4;
733: ofs:=16000*pg+80*y+(x div 4);
734: Anzahl:=0;
735: Skip:=0;
736: Quelle:=qsprite.adr;
737: asm
738: push ds
739: call disable4
740: mov dx,3c4h
741: {Portadresse laden}
742: mov ax,0a000h
743: {Zieladresse nach es:di}
744: mov es,ax
745: mov ax,qsprite.dtx
746: shr ax,2
747: mov byte ptr ppp[0],al
748: {ppp-Register initialisieren}
749: mov byte ptr ppp[1],al
750: {auf Anzahl mindestens zu kopierender
751: Bytes pro Plane}
752: mov byte ptr ppp[2],al
753: mov byte ptr ppp[3],al
754: mov cx,qsprite.dtx
755: {"überstehende" Pixel in ppp
756: eintragen}
757: and cx,3
758: je @nodbl
759: mov al,byte ptr plane
760: {ah enthält hier garantiert 0}
761: mov si,ax
762: @mark_double_planes:
763: inc byte ptr ppp[si]
764: inc si
765: and si,3
766: {Überlauf vorbeugen}
767: loop @mark_double_planes
768: @nodbl:
769: lds si,quelle
770: {Quelle nach ds:si}
771: mov cx,plane
772: {Plane - Maske erstellen}
773: mov ax,1

```

»s_unit.pas« läßt Intros und Demos mit animierter Laufschrift, Grafikeinbindung und Soundausgabe zum Kinderspiel werden.


```

774: shl ax,cl
775: mov planemask,al
776: shl al,4
777: {auch in oberes Nibble eintragen}
778: or planemask,al
779: mov planeindex,4
780: {4 Planes zu kopieren}
781: @lplane:
782: mov cl,byte ptr plane
783: {ch garantiert 0}
784: mov di,cx
785: {di wird erst weiter unten wieder
786:      gebraucht}
787: mov cl,byte ptr ppp[di]
788: {x-Zähler (cx) laden mit zugehöriger
789:      ppp-Nummer}
790: mov byte ptr anzahl,cl
791: mov ax,direction
792: sub ax,cx
793: mov skip,ax
794: mov rel_y,0
795: {start wieder bei y=0}
796: mov cx,anzahl
797: or cl,cl
798: {Breite 0, dann Plane fertig}
799: je @plane_fertig
800: mov di,ofs
801: {Zieloffset nach di}
802: mov ah,planemask
803: {planemask auf bit [0..3] reduzieren
804: und setzen}
805: and ah,0fh
806: mov al,02h
807: mov dx,3c4h
808: out dx,ax
809: mov bx,dy
810: {x- und y-Zähler initialisieren}
811: @lcopy_y:
812: @lcopy:
813: lodsb
814: {Byte holen, wenn 0 überspringen}
815: or al,al
816: je @wert0
817: stosb
818: {ansonsten: setzen}
819: @entry:
820: loop @lcopy
821: mov ax,rel_y
822: {Nachkommateil weiter}
823: add ax,add_y
824: cmp ax,100
825: {Überlauf ?}
826: jb @noaddovfl
827: sub ax,100
828: {für auf den Kopf gestellte Texturen}
829: sub di,direction
830: @noaddovfl:
831: mov rel_y,ax
832: dec bx
833: je @plane_fertig
834: {y-Zähler = 0, dann nächste Plane}
835: add di,skip
836: {sonst auf nächsten Zeilenanfang
837:      springen}
838: mov cx,Anzahl
839: {x-Zähler reinitialisieren,}
840: jmp @lcopy_y
841: @wert0:
842: inc di
843: {Sprite-Wert 0 : Zielbyte
844:      überspringen}
845: jmp @entry
846: @plane_fertig:
847: rol planemask,1
848: {nächste Plane maskieren}
849: mov cl,planemask
850: {plane 0 selektiert ?}
851: and cx,1
852: {(Bit 1 gesetzt), dann}
853: add ofs,cx

```

```

854: {Zieloffset erhöhen um 1 (Bit 1 !)}
855: inc plane
856: and plane,3
857: dec planeindex
858: {schon 4 Planes kopiert ?, dann Ende}
859: jne @lplane
860: pop ds
861: End;
862: {asm}
863: End;
864: Procedure WaitRetrace;assembler;
865: var test:Word;
866: asm
867: push dx
868: mov test,0
869: @waittime:
870: inc test
871: cmp timeflag,0
872: je @waittime
873: mov dx,3dah
874: @l1:
875: in al,dx
876: and al,8h
877: jnz @l1
878: @l2:
879: in al,dx
880: and al,8h
881: jz @l2
882: mov timeflag,0
883: mov ax,8300h
884: mov bx,offset timeflag
885: push ds
886: pop es
887: mov dx,dly
888: mov cx,dlyhigh
889: int 15h
890: pop dx
891: End;
892: Procedure Init_Scroller;
893: var z:char;
894: Begin
895:   sc_x1:=320;
896:   sc_x2:=320;
897:   Txt_Pos1:=1;
898:   Txt_Pos2:=1;
899:   txtnr1:=1;
900:   txtnr2:=1;
901:   For z:=' ' to 'Z' do
902:     GetSprite(3,C_Pos[z,0],
903:   C_Pos[z,1],C_Pos[z,2],C_Pos[z,3],
904:   Chars[z]);
905: End;
906: Procedure Draw_Scroller;
907: Var l_breite:Word;
908: {temporär zum Speichern der
909: Buchstaben-Breite verwenden}
910:   einmalnoch,fertig:Boolean;
911: Begin
912:   einmalnoch:=false;
913:   fertig:=false;
914:   Dec(sc_x1,sc_step);
915:   l_breite:=Chars[Txt[txtnr1,
916:   Txt_Pos1]].dtx;
917:   if txt[txtnr1,txt_pos1] = ' '
918:     then l_breite:=sc_space;
919:   If sc_x1 + l_breite <= 0 Then Begin
920: {Buchst. links raus ?}
921:   Inc(Txt_Pos1);
922: {nächsten Buchst. selektieren}
923:   if txt_pos1 > Length(txt[txtnr1])
924:     then begin
925:       inc(txtnr1);
926:       if txtnr1 > maxtext then
927:         txtnr1:=1;
928:       Txt_Pos1:=1;

```

»s_unit.pas« läßt Intros und Demos mit animierter Laufschrift, Grafikeinbindung und Soundausgabe zum Kinderspiel werden.


```

929: End;
930: Inc(sc_x1,l_breite + sc_abst);
931: {und dessen Koordinate festlegen}
932: End;
933: Dec(sc_x2,sc_step);
934: l_breite:=Chars[Txt[txtnr2,
935: Txt_Pos2]].dtx;
936: if txt[txtnr2,txt_pos2]=' ' Then
937: l_breite:=sc_space;
938: If sc_x2 + l_Breite +sc_abst < 320
939: Then Begin
940: {neuer Buchst. komplett drauen ?}
941: Inc(Txt_Pos2);
942: {nchsten Buchst. selektieren}
943: if txt_pos2 > Length(txt[txtnr2])
944: then begin
945: inc(txtnr2);
946: if txtnr2 > maxtext then
947: txtnr2:=1;
948: txt_pos2:=1;
949: End;
950: Inc(sc_x2,l_breite);
951: {und dessen Koordinate festlegen}
952: End;
953: sc_x:=sc_x1;
954: sc_numb:=0;
955: sc_i:=txt_pos1;
956: sc_txtnr:=txtnr1;
957: Repeat
958: time:=sc_x;
959: while time < 0 do
960: inc(time,320);
961: if Txt[sc_txtnr,sc_i]=' ' then
962: inc(sc_x,sc_space) Else Begin
963: PutSprite(Pakt,sc_x,
964: sc_y+sin_sng[time],
965: Chars[Txt[sc_txtnr,sc_i]]);
966: Inc(sc_x,Chars[Txt[sc_txtnr,
967: sc_i]].dtx);
968: End;
969: Inc(sc_x,sc_abst);
970: Inc(sc_i);
971: If sc_i > Length(txt[sc_txtnr])
972: then begin
973: inc(sc_txtnr);
974: If sc_txtnr > maxtext then
975: sc_txtnr:=1;
976: sc_i:=1;
977: End;
978: if einmalnoch then fertig:=true;
979: if (sc_i=txt_pos2)
980: and (sc_txtnr = txtnr2) then
981: einmalnoch:=true;
982: Until fertig;
983: End;
984:
985: Procedure GifOpen;assembler;
986: asm
987: mov ax,03d00h
988: lea dx,gifname + 1
989: int 21h
990: mov handle,ax
991: End;
992: Procedure GifRead(n:Word);assembler;
993: asm
994: mov ax,03f00h
995: mov bx,handle
996: mov cx,n
997: lea dx,puf
998: int 21h
999: end;
1000: Procedure GifSeekdelta(delta:
1001: Longint);assembler;
1002: asm
1003: mov ax,04200h
1004: mov bx,handle
1005: mov cx,word ptr delta + 2
1006: mov dx,word ptr delta
1007: int 21h
1008: End;

```

```

1009: Procedure GifClose;Assembler;
1010: asm
1011: mov ax,03e00h
1012: mov bx,handle
1013: int 21h
1014: End;
1015: Procedure ShiftPal;assembler;
1016: asm
1017: push ds
1018: pop es
1019: mov si,offset Puf
1020: mov di,offset Palette
1021: mov cx,768
1022: @l1:
1023: lodsb
1024: shr al,2
1025: stosb
1026: loop @l1
1027: End;
1028: Procedure FillPuf;
1029: Begin
1030: GifRead(1);
1031: restbyte:=puf[0];
1032: GifRead(restbyte);
1033: End;
1034: Function GetPhysByte:Byte;assembler;
1035: asm
1036: push bx
1037: cmp restbyte,0
1038: ja @restda
1039: pusha
1040: call fillpuf
1041: popa
1042: mov pufind,0
1043: @restda:
1044: mov bx,PufInd
1045: mov al,byte ptr Puf[bx]
1046: inc pufind
1047: pop bx
1048: End;
1049: Function GetLogByte:Word;assembler;
1050: asm
1051: push si
1052: mov ax,breite
1053: mov si,ax
1054: mov dx,restbits
1055: mov cx,8
1056: sub cx,dx
1057: mov ax,lByte
1058: shr ax,cl
1059: mov code,ax
1060: sub si,dx
1061: @nextbyte:
1062: call getphysbyte
1063: xor ah,ah
1064: mov lByte,ax
1065: dec restbyte
1066: mov bx,1
1067: mov cx,si
1068: shl bx,cl
1069: dec bx
1070: and ax,bx
1071: mov cx,dx
1072: shl ax,cl
1073: add code,ax
1074: sbb dx,breite
1075: add dx,8
1076: jns @positiv
1077: add dx,8
1078: @positiv:
1079: sub si,8
1080: jle @fertig
1081: { <= 0 }
1082: add dx,breite
1083: sub dx,8
1084: jmp @nextbyte

```

»s_unit.pas« lt Intros und Demos mit animierter Laufschrift, Grafikeinbindung und Soundausgabe zum Kinderspiel werden.


```

1085: @fertig:
1086:   mov restbits,dx
1087:   mov ax,code
1088:   pop si
1089: End;
1090: Procedure LoadGif(name:String;
1091:   bpos:word);
1092: Var ziel,
1093:   quelle,qseg:Word;
1094:   pic_size,pic_height,
1095:   pic_width:word;
1096:   x_count:Word;
1097:   plane_pos:Array[0..3] of Pointer;
1098:   plane_pos_i:Word;
1099:   Plane:Byte;
1100: Begin
1101:   gifName:=Name+#0;
1102:   getMem(temp-pic,64000);
1103:   GifOpen;
1104:   gifseekdelta(13);
1105:   gifread(768);
1106:   Shiftpal;
1107:   SetPal;
1108:   gifread(1);
1109:   While Puf[0] = $21 do Begin
1110:   {Erw - Block überlesen}
1111:     gifread(2);
1112:     gifread(puf[1]+1);
1113:   End;
1114:   GifRead(10);
1115:   pic_width:=puf[4]+puf[5]*256;
1116:   pic_height:=puf[6]+puf[7]*256;
1117:   pic_size:=pic_width
1118:     div 4 * pic_height;
1119:   If Puf[8] and 128 = 128 Then Begin
1120:     gifread(768);
1121:     Shiftpal;
1122:     SetPal;
1123:   End;
1124:   Plane_pos[0]:=temp-pic;
1125:   Plane_pos[1]:=Ptr(Seg(temp-pic^)+
1126:     pic_size div 16, 0);
1127:   Plane_pos[2]:=Ptr(Seg(temp-pic^)+
1128:     pic_size div 8, 0);
1129:   Plane_pos[3]:=Ptr(Seg(temp-pic^)+
1130:     (pic_size div 16)*3, 0);
1131:   Plane_pos_i:=6;
1132:   Plane:=1;
1133:   lByte:=0;
1134:   asm
1135:     les di,temp-pic
1136:     mov free,258
1137:   {1. freie Position in Alphabet}
1138:     mov breite,9
1139:   {Zeichenbreite in Bit}
1140:     mov max,511
1141:   {maximaler darstellbarer Wert bei
1142:     aktueller Breite}
1143:     mov stackp,0
1144:     mov restbits,0
1145:     mov restbyte,0
1146:     xor si,si
1147:   {aktuelle Plane
1148:   {für Zieladressierung}}
1149:   @mainloop:
1150:     call getlogByte
1151:     cmp ax,eof
1152:     je @abbruch
1153:     cmp ax,clr
1154:     je @clear
1155:     mov readbyt,ax
1156:     cmp ax,free
1157:     jb @code_in_ab
1158:     mov ax,old_code
1159:     mov code,ax
1160:     mov bx,stackp
1161:     mov cx,sonderfall
1162:     mov word ptr stack[bx],cx
1163:     inc stackp
1164:     @code_in_ab:

```

```

1165:     cmp ax,clr
1166:     jb @konkret
1167:   @fillstack_loop:
1168:     mov bx,code
1169:     shl bx,1
1170:     push bx
1171:     mov ax,word ptr ab_tail[bx]
1172:     mov bx,stackp
1173:     shl bx,1
1174:     mov word ptr stack[bx],ax
1175:     inc stackp
1176:     pop bx
1177:     mov ax,word ptr ab_prfx[bx]
1178:     mov code,ax
1179:     cmp ax,clr
1180:     ja @fillstack_loop
1181:   @konkret:
1182:     mov bx,stackp
1183:     shl bx,1
1184:     mov word ptr stack[bx],ax
1185:     mov sonderfall,ax
1186:     inc stackp
1187:     mov bx,stackp
1188:     dec bx
1189:     shl bx,1
1190:   @readstack_loop:
1191:     mov ax,word ptr stack[bx]
1192:
1193:     mov di,word ptr plane_pos[si] + 2
1194:     mov es,di
1195:     mov di,word ptr plane_pos[si]
1196:     stosb
1197:     inc word ptr plane_pos[si]
1198:     add si,4
1199:     and si,12
1200:     or di,di
1201:     jne @rsnc
1202:     mov ax,es
1203:     add ax,1000h
1204:     mov es,ax
1205:   @rsnc:
1206:     dec bx
1207:     dec bx
1208:     jns @readstack_loop
1209:     mov stackp,0
1210:     mov bx,free
1211:     shl bx,1
1212:     mov ax,old_code
1213:     mov word ptr ab_prfx[bx],ax
1214:     mov ax,code
1215:     mov word ptr ab_tail[bx],ax
1216:     mov ax,readbyt
1217:     mov old_code,ax
1218:     inc free
1219:     mov ax,free
1220:     cmp ax,max
1221:     jbe @mainloop
1222:     cmp byte ptr breite,12
1223:     jae @mainloop
1224:     inc breite
1225:     mov cl,byte ptr breite
1226:     mov ax,1
1227:     shl ax,cl
1228:     dec ax
1229:     mov max,ax
1230:     jmp @mainloop
1231:   @clear:
1232:     mov breite,9
1233:     mov max,511
1234:     mov free,258
1235:     call getlogbyte
1236:     mov sonderfall,ax
1237:     mov old_code,ax
1238:     mov di,word ptr plane_pos[si] + 2
1239:     mov es,di
1240:     mov di,word ptr plane_pos[si]

```

»s_unit.pas« läßt Intros und Demos mit animierter Laufschrift, Grafikeinbindung und Soundausgabe zum Kinderspiel werden.


```

1241:      stosb
1242:      inc word ptr plane_pos[si]
1243:      add si,4
1244:      and si,12
1245:      or di,di
1246:      jne @mainloop
1247:      mov ax,es
1248:      add ax,1000h
1249:      mov es,ax
1250:      jmp @mainloop
1251: @abbruch:
1252: End;
1253: gifclose;
1254: asm
1255:      push ds
1256:      lds si,temp pic
1257:      mov ax,0a000h
1258:      mov es,ax
1259:      mov di,bpos
1260:      mov ziel,di
1261:      mov cx,pic_size
1262: @lpplane:
1263:      mov al,02h
1264:      mov ah,plane
1265:      mov dx,3c4h
1266:      out dx,ax
1267: @lpl:
1268:      rep movsb
1269:      mov di,ziel
1270:      mov si,plane_pos_i
1271:      mov si,word ptr plane_pos[si]
1272:      add plane_pos_i,4
1273:      mov ds,si
1274:      xor si,si
1275:      mov cx,pic_size
1276:      shl plane,1
1277:      cmp plane,10h
1278:      jne @lpplane
1279:      pop ds
1280: End;
1281: Freemem(temp pic,64000);
1282: End;
1283: Begin
1284: End.

```

»s_unit.pas« läßt Intros und Demos mit animierter Laufschrift, Grafikeinbindung und Soundausgabe zum Kinderspiel werden.

Listing 2: simple_s.pas

```

1: {Programm: simple_s.pas
2: Funktion: Demoprogramm zu s_unit.pas
3:      (Animation mit
4:      Sound-Ausgabe)
5: Sprache : Turbo Pascal ab 6.0
6: Autor   : Matthias Rasch
7: (c) 1993 DMV Daten- u. Medienverlag}
8: program simple_sprite;
9: {$m $4000,0,300000}
10: uses s_unit,crt,modplay;
11: {modplay evtl. weglassen}
12:
13: {$Define sound}
14: const samfr : byte = 8;
15:      ballstart : word = 158;
16:
17: {$ifdef sound}
18: procedure Exit_the_mod;
19: begin;
20:   periodisch_off;
21:   ende_mod;
22:   if SbVersMaj<4 then
23:   port[dsp_adr+$c]:=1;
24:   end;
25: procedure init_The_Mod;
26: begin;
27:   mod_Setloopflag(ON);
28:   init_sb;

```

```

29:   mod_Samplefreq(samfr);
30: end;
31: procedure Play_the_Mod(s : string);
32: var error : integer;
33: begin;
34:   mod_Samplefreq(samfr);
35:   error := lade_moddatei(s,AUTO,AUTO);
36:   if error <> 0 then begin;
37:     asm mov ax,0003; int 10h; end;
38:     writeln('Fehler beim Laden der ',
39:       'MOD-Datei !');
40:     if error = -1 then writeln(
41:       'Initialisierung fehlerhaft !');
42:     if error = -2 then writeln(
43:       'Datei nicht gefunden !');
44:     halt(0);
45:   end;
46:   mli := 0;
47:   mlj := 0;
48:   periodisch_on;
49: end;
50: {$endif}
51: label weiter;
52: Begin
53:   Init_VGA;
54:   timeflag:=1;
55:   PAKt:=1;
56:   Switch(0);
57:   LoadGif('pfont.gif',48000);
58:   LoadGif('phint.gif',32000);
59:   Init_Scroller;
60:   GetSprite(3,248,106,41,36,
61:     Sprite[1]);
62:   GetSprite(3,5,97,89,47,texture[1]);
63:   i:=300;
64:   dir:=-1;
65:   time:=0;
66: {$ifdef sound}
67:   init_the_mod;
68:   play_the_mod('hurrica2.mod');
69: {$endif}
70:   gi:=10;
71:   gidir:=-1;
72:   Repeat
73:     copyScreen(2,pakt);
74:     disable4;
75:     PutTexture(PAKt,116,84-gi*2,gi*10,
76:       Texture[1]);
77:     For i:=1 to 4 do
78:     {$ifdef sound}
79:       PutSprite(PAKt,-20+i*64,
80:         ballstart-(Noten_Anschlag[i] div 7),
81:         Sprite[1]);
82:     {$else}
83:       PutSprite(PAKt,-20+i*64,
84:         ballstart-gi*(Integer(Odd(i))+1),
85:         Sprite[1]);
86:     {$endif}
87:     Draw_Scroller;
88:     Switch(PAKt);
89:     {$ifdef sound}
90:     mod_WaitRetrace;
91:     {$else}
92:     waitretrace;
93:     {$endif}
94:     PAKt:=1-PAKt;
95:     inc(i,dir);
96:     if i<0 then dir:=-dir;
97:     if i>300 then dir:=-dir;
98:     Inc(gi,gidir);
99:     if (gi=10) or (gi=-10) then
100:       gidir:=-gidir;
101:   Until KeyPressed;
102:   TextMode(3);
103: {$ifdef sound}
104:   Exit_the_mod;
105: {$endif}
106: End.

```

»simple_s.pas« zeigt eindrucksvoll die Fähigkeiten von »s_unit.pas«.

MIDI – Schlüssel zur digitalen Musikwelt

Wer schon einmal etwas mit modernen, digitalen Musikinstrumenten zu tun hatte, kennt MIDI. Die MIDI-Schnittstellendefinition wurde zu Beginn der Synthesizerära von einem Zusammenschluß verschiedener Elektronikkonzerne gegründet und setzte sich trotz anfänglicher Schwierigkeiten und Ausstieg einiger Firmen schon nach kurzer Zeit bei Musikern durch.

Es handelt sich bei dieser Schnittstellendefinition um den ersten Standard zur Verbindung elektronischer Musikinstrumente, der auf digitaler Basis arbeitet und somit nicht nur Toninformationen überträgt, sondern auch Steuer- und Regelfunktionen ausübt. Dazu verbindet man die einzelnen Komponenten (Keyboards, Synthesizer, Drum-Computer, um nur einige zu nennen) mit Hilfe eines fünfpoligen Kabels mit den bereits in anderer Funktion (Stereoanlagen) bekannten DIN-Steckern. Die Übertragung der Daten erfolgt seriell, was die einfachen Kabel, von denen sogar nur vier Leitungen wichtig sind, erklärt. Auch die Soundblaster-Karte ist von Anfang an mit MIDI-Fähigkeiten ausgestattet gewesen, sie besitzt jedoch nicht direkt die erforderlichen Stecker. Der Anwender muß sie gesondert mit Hilfe eines Adapters an den Joystickport der Karte anschließen. Ganz nach Ausführung stehen dann jeweils mindestens ein MIDI-Eingang und ein MIDI-Ausgang zur Verfügung, über die der PC angeschlossene Instrumente kontrolliert und steuert.

Dazu wird der MIDI-Out-Anschluß mit dem MIDI-In des Zielgerätes verbunden und umgekehrt, so daß Daten, die der PC abschickt, am Eingang des Keyboards oder anderen Geräten eintreffen sowie der PC die auf den angeschlossenen Musikinstrumenten gespielten Informationen entgegennimmt. Der PC schlüpft damit in die Rolle eines Sequenzers, der – von einem geeigneten Programm unterstützt – die ankommenden Daten aufzeichnet, speichert, manipuliert und später wieder mit Hilfe der Originalinstrumente (oder nach einigen Angleichungen auch anderer Geräte) abspielt. In unserem Fall sollen die MIDI-Daten nur gelesen und ausgewertet werden, womit Sie beispielsweise auch bei Verwendung eines Mod-Editors in Berührung kommen.

Die Zusammensetzung der Daten ist relativ einfach: Man unterscheidet zwischen Status- oder Steuer- und Datenbyte, wobei Statusbytes im Gegensatz zu Datenbytes am gesetzten Bit 7 zu erkennen sind. Sta-

tusbytes stellen gewissermaßen einen Befehl dar, dem als Argumente, ähnlich einer Pascal-Prozedur, die Datenbytes folgen. Diese Regel hat allerdings auch Ausnahmen, nämlich immer dann, wenn sehr viele Befehle direkt aufeinanderfolgen. In diesem Fall sparen sich einige Keyboards die Wiederholung des Statusbytes und geben direkt die Datenbytes aus. Beispielsweise wird beim Drücken vieler Tasten gleichzeitig der Befehl zum Anschlagen einer Note (*Note-On*) nur einmal erteilt und die folgenden Tastendrücke nur noch als Datenbyte übermittelt, bis ein anderer Befehl eintrifft oder die Anschläge weniger dicht aufeinanderfolgen.

Um unter MIDI auch mehr als zwei Instrumente miteinander zu verbinden, benutzt man 16 verschiedene Kanäle für die einzelnen Instrumente, jedes sendet und empfängt auf einem bestimmten Kanal. Die Numerierung ist bei Instrumenten meist von 1 bis 16, dennoch enthalten die entsprechenden Steuerbytes die Werte 0 bis 15 (Bits 0 bis 3).

Es gibt verschiedene Arten von Statusbytes, von Interesse sind für uns allerdings lediglich zwei: Zunächst einmal die *Channel-Voice-Messages*, die einzelne Kanäle ansprechen und sich auf die Stimme beziehen, also den Klang, der gerade vom Instrument gespielt wird. Unter ihnen befinden sich die zwei wohl wichtigsten Befehle: *Note-Off* (8xh, x steht für Kanalnummer) und *Note-On* (9xh). *Note-On* schlägt eine Taste an, und *Note-Off* läßt sie wieder los. Als Datenbytes übergibt man zum einen jeweils die Nummer der MIDI-Note, angefangen mit 0 beim tiefsten C bis hin zu 127 für das höchste G, wobei allerdings zu beachten ist, daß die wenigsten Keyboards den vollen Tastenumfang von 128 Tasten besitzen und meist nur einen Ausschnitt dieses Bereichs nutzen (zum Beispiel 12 bis 83). Als zweites Datenbyte wird der Wert der Anschlagsdynamik übermittelt. Hierbei handelt es sich um die »Härte«, mit der die entsprechende Taste gedrückt oder losgelassen wurde. Die Anschlagsdynamik soll dabei den Effekt

nachahmen, daß, beispielsweise beim Klavierspielen, ein fester Druck auch einen lautereren Ton erzeugt. Die Werte für dieses Byte bewegen sich im Bereich von 0 bis 127, wobei 0 teilweise eine besondere Bedeutung zukommt. Manche Instrumente verzichten auf die Verwendung des *Note-Off*-Befehls zum Abschalten des Tons und nutzen statt dessen den *Note-On*-Befehl mit der gleichen Note, allerdings mit einer Anschlagdynamik von 0, was ja auch einem Druck der Stärke 0, also dem Loslassen, entspricht.

Weitere Befehle der *Channel-Voice-Messages* betreffen den sogenannten »Aftertouch«, also das unterschiedlich starke Festhalten bestimmter Tasten nach dem Anschlag, sowie interne Einstellungen, darunter Programm- (also Klang-)Nummer und »Pitch-Bend-Rad« (zur direkten Manipulation der Tonhöhe).

Die *Channel-Mode*- und *System-Exclusive-Messages* sind an dieser Stelle nicht von Interesse. Dagegen stellen die *System-Realtime-Messages* den zuvor erwähnten zweiten Bereich der Statusbytes dar. Diese bestehen zum einen nur aus einem Statusbyte ohne Datenbyte und betreffen zum anderen keinen bestimmten Kanal, sondern das ganze »System«. Außerdem sind es *Realtime-Messages*, das heißt, sie werden unter bestimmten Bedingungen regelmäßig ausgesandt. Von Bedeutung ist in diesem Zusammenhang zum einen die *Timing-Clock-Message* (0f8h), die von einem Mastergerät ausgeht und allen anderen den Takt angibt, sowie außerdem die *Active-Sensing-Message* (0feh). Letztere sendet jedes Gerät, wenn keine anderen Daten zu übertragen sind, um eine Leitungsunterbrechung sofort aufspüren zu können und somit die Sicherheit des Systems zu erhöhen. Der Einsatz der Soundblaster-Karte zur MIDI-Steuerung läßt sich am besten mit Hilfe des Beispielprogramms »mididemo.pas« (Listing) erläutern.

Zunächst wird dabei die Soundblaster-Karte durch Aufruf der Prozedur *Reset_DSP* initialisiert, was durch leicht verzögertes Schreiben von 1 und dann 0 an die Reset-Port-Adresse (Basisadresse +6, also meist 226h) und anschließendes Warten auf 0aah am *Read Data Port* (Basis +0ah) erfolgt.

Das eigentliche Lesen der MIDI-Daten übernimmt die Prozedur *Read_Midi*. Dazu übermittelt man dem DSP zunächst den Befehl 30h, der ihn dazu veranlaßt, MIDI-Daten zu lesen. Dies wiederum geschieht anhand der Prozedur *Write_DSP*, die zunächst auf die Bearbeitung (Bit 7 = 0 am *Write Data Port*, Basis +0ch) einer eventuell zuvor gesendeten Anweisung wartet und dann den neuen Befehl an die gleiche Port-Adresse schreibt. Die ankommenden MIDI-Daten legt die Soundblaster-Karte im Lesebuffer ab (bei der Ausgabe entsprechend im Schreibpuffer). In der folgenden Leseschleife wird ständig der *Read Buffer Status Port* (Basis +0eh) überprüft,

PaintingSystems

PAI386DX/40MHZ	128 KB	2375 DM
PAI486DX/33MHZVL	256 KB	3299 DM
PAI486DX2/66MHZVL	256 KB	3999 DM
PAI-Prentium/60MHZ-Eisa		10899 DM

Alle Systeme inkl. : Weitere Systeme auf Anfrage!

Mainboard inkl. 4 MB Ram ,Tower-Gehäuse,
Festplatte AT-Bus 340 MB , Floppylaufwerke 1.2+1.44 MB
Grafikkarte 1 MB, Monitor 14" VGA-Color 1024x768 ,
Multi I/O-AT-BUS Karte , 2xSer, 1xPar, Cherry-Tastatur ,
Maus , Pad, DOS 6.0, Windows 3.1, Handbuch (800 S.)

Streamer 250MB Conner oder Iomega*	349 DM
CD-Rom Mitsumi oder Philips*	369 DM
Cache-Controller(bis 16MB)*	289 DM
Netzwerkkarte Compex ENET*	154 DM
Notstromversorgung (600 VA)*	399 DM
NOVELL 3.12 5 User (inkl. Einrichtung)*	1200 DM
NOVELL 4.0 5 User (inkl. Einrichtung)*	a.Anfr.

*Aufpreis zu einem der genannten Systeme

FESTPLATTEN

CONNER AT	170 MB	399 DM
CONNER AT	256 MB	479 DM
CONNER AT	343 MB	602 DM
CONNER AT	420 MB	844 DM
CONNER AT	545 MB	1139 DM
PAINTING SCSI	1.05 GB	1790 DM
PAINTING SCSI	2.0 GB	3400 DM

DRUCKER

HP DESKJET 510	780 DM
HP DESKJET 500 Color	849 DM
HP DESKJET 550 Color	1350 DM
HP LASERJET 4	3379 DM
HP LASERJET 4L	1599 DM
OKI Laser OL 400e DIN A4/4 Seiten	1149 DM
OKI Laser OL 810 DIN A4/8 Seiten	2540 DM
OKI Laser OL 850 DIN A4/8 Seiten Ps	3822 DM
NEC Jetmate 800 DIN A4 Tinte	739 DM

MODEMS

TELEJET 14400 *BZT	659 DM
ZYXEL U-1496E	740 DM
ZYXEL U-1496E-Plus	955 DM
ZYXEL U-1496EG*BZT!NEU!	1189 DM
ZYXEL U-1496P !NEU!	999 DM
COURIER-DUAL16.8McK-FAX-BZT	1389 DM
COURIER-V.32terbo-FAX !NEU!	1299 DM
COURIER-DUAL16.8/19.2FAX !NEU!	1599 DM
Elink 301	1599 DM
ISDN-Blaster PC-Karte*	899 DM
ISDN-Master Amiga-Karte*	1119 DM

* inkl. Softwarepaket! Sysoppreise erfragen!
Anschluss der Modems, außer mit "BZT-Zeichen" am
Netz der DBP-Telekom verboten und strafbar!

PaintingSystems

EDV-Design & Service GbR D2 PRIVAT

Inh. Kirch & Reuter

Am Baldhof 67 41464 Neuss

TEL 02131/940144 FAX 940148

Mailboxhotline: 02131/940148

Alle Angeb. freibleibend, technisch. Änderungen
vorbehalten. Alle Preise inkl. MWST zzgl.
Versand und Versicherung.
Alle Namen sind eingetragene Warenzeichen.
Katalog mit über 3000 Artikeln jetzt anfordern!

der mit gesetztem Bit 7 anzeigt, daß Daten
im Puffer bereitstehen.

Um das Einlesen kümmert sich nun der
Read Data Port (0ah). An dieser Stelle
bietet es sich an, die beiden ständig auftre-
tenden System-Realtime-Messages *Active*
Sensing und *Timing-Clock* herauszufiltern,
da sie für uns keine weitere Bedeutung be-
sitzen. Die übrigen Datenbytes sind im
Puffer zur späteren Ausgabe auf dem Bild-
schirm abgelegt.

Auch eine Ausgabe auf einem MIDI-
Gerät, beispielsweise dem Ursprungsgerät,
läßt sich realisieren. Hierzu sind dann al-
lerdings auch die Zeitwerte aller Ereignis-
se zu protokollieren, was man am einfach-
sten über den Interrupt 8 löst. Diese Routi-
ne erhöht in unserem Fall die Variable
Counter alle 1/1000 Sekunde um eine Ein-
heit und ruft außerdem den Original-Inter-
rupt-Handler weiterhin 18,2mal pro
Sekunde auf. Die entsprechenden Werte
befinden sich im Array *TimBuf*.

Beim Abspielen wird bei jedem Code
zunächst einmal die vorgegebene Zeit ab-
gewartet. Das Senden des Codes an sich ist
sehr einfach: Man gibt dem DSP den Be-
fehl 38h zur Ausgabe von MIDI-Daten
und schreibt dann genauso wie zum Ertei-
len eines Befehls die Daten in den *Write*
Data Port. Das MIDI-Kabel übermittelt
die Informationen nun zum Synthesizer-
Keyboard, das sie schließlich abspielt. So-
mit wäre ein einfacher Einkanal-Sequencer
bereits realisiert. Zu beachten gilt noch der
Reset (MIDI-Code 0ffh) des Keyboards,
der vor der eigentlichen Ausgabe über MI-
DI veranlaßt werden sollte, da sonst bei ei-
nigen Synthesizern falsche Klänge auftre-
ten.

(Matthias Rasch/hs/bj)

Listing: mididemo.pas

```

1: {Programm: mididemo.pas
2: Funktion: Veranschaulicht die Aus-
3: gabe von Midi-Daten an
4: externe Midi-Geräte
5: Sprache : Turbo Pascal ab 6.0
6: Autor : Matthias Rasch
7: (c) 1993 DMV GmbH & Co.KG}
8: Program Midi_Demo;
9: Uses Crt,Dos;
10: Var Buffer:Array[0..500] of Byte;
11: Bufpt, BufEnde:Word;
12: TimBuf:Array[0..500] of Word;
13: Counter,
14: TimCount:word;
15: OldInt8:Pointer;
16: procedure Reset_DSP;
17: var
18: status : integer;
19: Begin
20: port[$226]:=1;
21: {Reset}
22: delay(1);
23: port[$226]:=0;
24: Repeat
25: status:=port[$22A];
26: {Warten auf Bestätigung}
27: Until status = $aa
28: End;
29: procedure Write_DSP(befehl : byte);
30: {bereit für Befehl ?}
31: Begin
32: Repeat Until port[$22C] and 128 = 0;
33: port[$22C]:=befehl;
34: {dann schreiben}
35: End;
36: procedure Read_Midi;
37: var
38: i : byte;
39: taste : char;
40: Begin
41: WriteLn('Spielen Sie bitte auf dem
Keyboard');
42: Bufpt:=0;

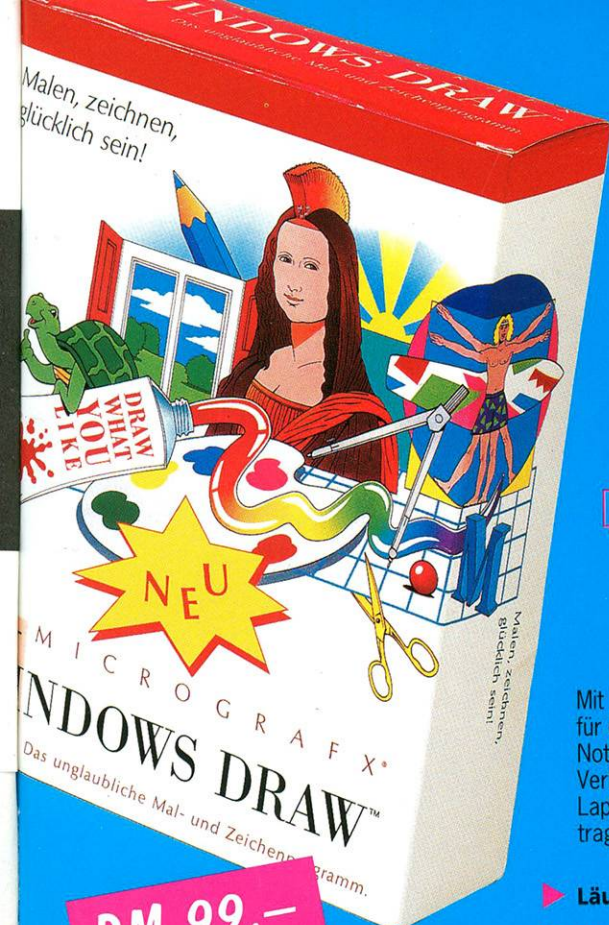
```

```

43: {Variablen laden}
44: BufEnde:=500;
45: write_dsp($30);
46: {Befehl zum Lesen}
47: Repeat
48: If Bufpt<BufEnde Then Begin
49: {Buffer voll, dann Ende}
50: If port[$22E] and 128 <> 0 Then Begin
51: {Daten da ?}
52: i:=port[$22A];
53: {dann lesen}
54: If (i<>$FE) and (i<>$f8) Then Begin
55: {active sensing & timing clock}
56: Buffer[Bufpt]:=i;
57: {aufzeichnen}
58: TimBuf[Bufpt]:=Counter;
59: Counter:=0;
60: inc(Bufpt);
61: End;
62: End;
63: End Else Begin
64: WriteLn('Puffer voll');
65: Exit;
66: End;
67: Until KeyPressed;
68: BufEnde:=Bufpt;
69: End;
70: Function Hex(Dez:Byte):String;
71: {Umwandlung nach Hex-String}
72: Var Stelle,i:Integer;
73: Erg:String;
74: Begin
75: i:=2;
76: Erg:=' ';
77: Repeat
78: Stelle:=Dez mod 16;
79: Erg[i]:=Chr(48+Byte(Stelle > 9)*
39+Stelle);
80: Dez:=dez div 16;
81: Dec(i);
82: Until Dez = 0;
83: Hex:=Erg;
84: End;
85: Procedure Disp_Midi;
86: Var i:Integer;
87: Begin
88: For i:=0 to BufEnde do Begin
89: If Buffer[i] and 128 = 128 Then
90: {Statusbyte, dann neue Zeile}
91: WriteLn;
92: Write(Hex(Buffer[i]),' ');
93: End;
94: End;
95: End;
96: Procedure Write_Midi;
97: Var i:Word;
98: Begin
99: write_dsp($38);
100: {Reset - Code übermitteln}
101: write_dsp($ff);
102: Counter:=0;
103: For i:=0 to BufEnde do begin
104: Repeat Until Counter=TimBuf[i];
105: {Zeit abwarten}
106: Counter:=0;
107: Write_DSP($38);
108: {Befehl zur Ausgabe}
109: Write_DSP(Buffer[i]);
110: {Code - Ausgabe}
111: End;
112: End;
113: End;
114: Procedure Handler8;interrupt;
115: Begin
116: Inc(Counter);
117: {Zeit abzählen}
118: Dec(TimCount);
119: {alten Interrupt aufrufen}
120: If TimCount = 0 Then Begin
121: asm
122: pushf
123: call [oldint8]
124: End;
125: TimCount:=56;
126: End Else Port[$20]:=$20;
127: End;
128: Procedure Start_Timer;
129: Begin
130: Port[$43]:=$36;
131: Port[$40]:=$169;
132: {1000 mal pro sekunde}
133: Port[$40]:=$4;
134: GetIntVec(8,OldInt8);
135: {Vektoren verbiegen}
136: SetIntVec(8,@Handler8);
137: End;
138: Procedure End_Timer;
139: Begin
140: Port[$43]:=$36;
141: Port[$40]:=$0;
142: Port[$40]:=$0;
143: SetIntVec(8,OldInt8);
144: End;
145: Begin
146: clrscr;
147: Reset_DSP;
148: Start_Timer;
149: Read_Midi;
150: Disp_Midi;
151: readln;
152: Write_Midi;
153: End_Timer;
154: End.

```

»mididemo.pas« veranschaulicht die Ausgabe
von MIDI-Daten auf externe MIDI-Geräte.



DM 99,-

WINDOWS DRAW 3.1 das einzigartige Zeichenprogramm von Micrografx!

Windows Draw zum professionellen Gestalten von Vektor- und Präsentationsgrafiken.

- ▶ Kreise, Kreissegmente, Polygone, Linien, Bögen und Rechtecke
- ▶ Freihandzeichnen, Beziérkurven und Objektbearbeitung mit Umformen, Gruppieren, Glätten, Kippen, Ausrichten
- ▶ umfangreiche Zeichen- und Konstruktions-Funktionen
- ▶ WYSIWYG-Darstellung mit stufenloser Vergrößerung und automatischem Ausrichten von Objekten
- ▶ Bis zu 16 Mio. Farben, frei definierbare Farbverläufe und eigene Farb-Creationen
- ▶ Import und Export von WMF-, CGM-, GEM-, PIC-, EPS-, PCX- und TIFF-Grafiken

PLUS

- ▶ 32 neue TrueType Fonts und 8 Outline-Fonts
- ▶ Über 2600 ClipArts und ClipArts-Manager

Versand Deutschland: + DM 6,- bei Vorauskasse, + DM 9,- bei Nachnahme. / Versand Ausland ausschließlich per Nachnahme + DM 15,-.

SOFTWARE EISKALT KALKULIERT!

LAPLINK XL der Connectivity- Klassiker von Travelling Software!

Mit LapLink XL steht Ihnen der schnellste Weg für den Dateientransfer zwischen Laptops, Notebooks, Handheld oder PC's zur Verfügung. Verbinden Sie einfach zwei Rechner mit einem LapLink-Kabel und schon kann die Dateiübertragung beginnen.

- ▶ Läuft unter DOS und Windows
- ▶ Höchste Übertragungsgeschwindigkeit: parallele (bis zu 500 000 Baud) und serielle (200 000 Baud) Datenübertragung
- ▶ Automatische Schnittstellenerkennung
- ▶ Speicherbedarfsanzeige
- ▶ Dateien von einem Rechner auf den anderen verschieben

Deutsche
Vollversion

PLUS

Serielltes Kabel



DM 66,-

▶ Oder als Kombi-Paket
Wordstar für Windows und
Windows Draw DM 144,-



DM 55,-

WORDSTAR FÜR WINDOWS 1.5 der Textverarbeitungs- klassiker nun auch unter Windows!

WordStar für Windows verbindet professionelle Textverarbeitung mit leistungsfähigen DTP-Funktionen. Mit TrueType-Schriften, Objekt Linking und Embedding (OLE).

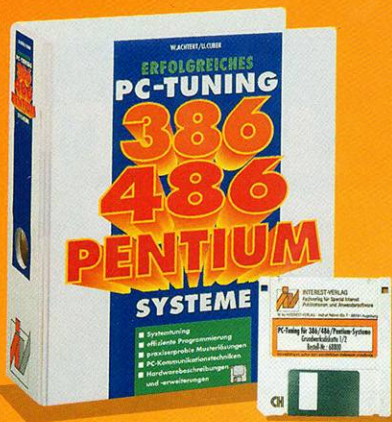
- ▶ WYSIWYG-Darstellung im Maßstab von 25% bis 200% Vergrößerung
- ▶ Exakte Zeilen- und Zeichenabstandskontrolle, Schriftgrößen von 1 bis 792 Punkt
- ▶ Automatische Inhaltsverzeichnis- und Indexerstellung, Querverweise, Fußnoten
- ▶ Rechtschreibprüfung, Trennhilfe und Synonymlexikon
- ▶ Serienbriefe mit bestehenden Dateien (z.B. dBase-Daten)
- ▶ Tabellen und Grafiken importieren, erstellen, bearbeiten
- ▶ TrueType und Adobe Type 1 Schriften
- ▶ vollständige Software mit dem bhv-Einsteigerseminar "Wordstar für Windows 1.5" als Handbuch

Die Bestell-Hotline:
08121/769-102
oder fix faxen:
08121/769-103



IMMER VOLLES PROGRAMM!

Mehr Power für Ihren PC!



Dieses Nachschlagewerk zeigt Ihnen, wie Sie Ihren Rechner für die verschiedensten Anwendungen aufrüsten, Programme bestmöglich installieren und effizient einsetzen.

Aus dem Inhalt:

- ☐ **Leistungssteigernde Utilities:** Windows-Beschleuniger zur Verdoppelung der Geschwindigkeit von Windows; VGA-Tuner, mit dem Sie die Bildwiederholrate auf flimmerfreie 67,5 Hz erhöhen; universeller DOS-Makro-Rekorder für die Automatisierung von Anwendungen...
- ☐ **Optimierung des Betriebssystems:** Tuning-Tips zur Installation und Konfiguration für eine optimale Nutzung von MS-DOS;
- ☐ **Programmierkurse und Musterlösungen:** Kurs „Software-Engineering“ zur strukturierten Programmierung; komplette Musterlösungen für effektive Software-Entwicklung;
- ☐ **Tips und Tricks** zur effizienteren Nutzung gängiger Anwenderprogramme (Excel, Winword...);
- ☐ **PC-Check:** zur Optimierung der System-einstellungen, Ermittlung und Beseitigung von Fehlern und Störungen;
- ☐ **Praxistips zum Systemtuning:** u.a. Einbau eines Scanners, Tips zur Datenkomprimierung und Festplattenoptimierung, „Verwandlung“ Ihres PCs in eine Multimedia-Maschine u.v.m.

„Erfolgreiches PC-Tuning für 386/486/Pentium-Systeme“

Praktischer Ringbuchordner, DIN A5, Grundwerk ca. 500 Seiten, inkl. zwei 5 1/4"-Disketten, Bestell-Nr.: 8800, Preis: DM 99,-, wahlweise zwei 3 1/2"-Disketten (gegen DM 4,- Aufpreis).

Alle 2-3 Monate erhalten Sie Erweiterungen zu Ihrem Grundwerk mit je ca. 100 Seiten, inkl. Diskette, zum Preis von DM 59,50 (Abbestellung jederzeit möglich).

Prüfen Sie

Ihr Nachschlagewerk in Ruhe zu Hause.
Wenn Sie nicht überzeugt sind, schicken Sie
dieses innerhalb von 10 Tagen an uns zurück.
Damit ist die Sache für Sie erledigt.



INTEREST-VERLAG
Fachverlag für Special Interest
Publikationen und Anwendersoftware
Hofrat-Röhler-Straße 7
86161 Augsburg
Tel. 08 21/56 07-0
Fax 08 21/56 07-299

Lösungen für Turbo Pascal 6.0 und 7.0



Mit diesem Nachschlagewerk erstellen Sie komfortabel eigene, leistungsstarke Applikationen und reduzieren Ihren Programmieraufwand ganz erheblich.

Aus dem Inhalt:

- ☐ **Ausgefeilte Turbo-Pascal-Lösungen:** Programme und Routinen für Turbo-Pascal 6.0/7.0, u.a. Lösungen für technisch/mathematische Anwendungen, Datenbanksysteme...
- ☐ **Turbo-Pascal-Kurs:** inklusive einem Schnelleinstieg in objektorientierte Programmierung und weiterführender Konzepte für Turbo-Pascal für Windows und Turbo-Vision;
- ☐ **raffinierte Modulbibliotheken** für Windows und DOS;
- ☐ **Multitasking-Subsystem** als Ergänzung zu DOS: Jetzt lassen Sie über 50 Tasks in drei Prioritätsebenen parallel ablaufen;
- ☐ **Tools und Utilities:** u.a. Turbo-Vision-Programmgenerator; Schnittstellen-Utilities sowie Utilities zur Datensicherung;
- ☐ **Turbo-Pascal und Multimedia:** u.a. Routinen zur Ansteuerung eines CD-ROM-Laufwerks, zum Abspielen einer Audio-CD und zur Programmierung der Soundblaster-Karte.

„Raffinierte Lösungen in Turbo-Pascal“

Stabiler Ringbuchordner, DIN A4, Grundwerk ca. 670 Seiten, inkl. 5 1/4"-Diskette, Bestell-Nr.: 2500, Preis: DM 99,-, wahlweise 3 1/2"-Diskette (gegen DM 2,- Aufpreis).

Alle 2-3 Monate erhalten Sie Erweiterungen zu Ihrem Grundwerk mit jeweils ca. 130 Seiten, inkl. Diskette, zum Preis von DM 59,50 (Abbestellung jederzeit möglich).

Jetzt setzen Sie MS-DOS noch effizienter ein!



Das kompetente Nachschlagewerk für sicheres und gezieltes Programmieren. Neben den dokumentierten erläutert es Ihnen auch die nicht dokumentierten DOS-Funktionen.

Aus dem Inhalt:

- ☐ **Interrupt-Programmierung:** detaillierte Dokumentationen zur BIOS-Interrupts und DOS-Funktionen, z.B. für die Initialisierung der Schnittstellen, zum Verändern der Größe eines Speicherblocks u.v.m.;
- ☐ **das DOS-System im Detail:** DOS-Intensivkurs mit Informationen zur Konfiguration wichtiger Systemkomponenten sowie zu internen Systemkomponenten;
- ☐ **DOS-Kommandos:** alphabetisches Befehlsverzeichnis mit ausführlichen Erläuterungen;
- ☐ **praxisbewährte Musterlösungen,** u.a. komplette Systemprogramme zur Ansteuerung der PC-Peripherie, zur Dateiverwaltung und zur Steuerung interner Abläufe;
- ☐ **Tools und Utilities:** z.B. LASER-SCREEN für Hardcopies auf dem Laserdrucker, LOGAKT zum Protokollieren von Eingaben sowie eine komfortable Mauswerkstatt (mit zahlreichen Mausmakros und -funktionen).

„DOS-Systemprogrammierung für IBM-PCs und Kompatible“

Praktischer Ringbuchordner, Format DIN A4, Grundwerk ca. 880 Seiten, inkl. zwei 5 1/4"-Disketten, Bestell-Nr.: 5000, Preis: DM 99,-, wahlweise 3 1/2"-Diskette (gegen DM 2,- Aufpreis).

Alle 2-3 Monate erhalten Sie Erweiterungen zu Ihrem Grundwerk mit jeweils ca. 130 Seiten, inkl. Diskette, zum Preis von DM 59,50 (Abbestellung jederzeit möglich).

JA, senden Sie mir sofort:

- ☐ Expl. „Erfolgreiches PC-Tuning für 386/486/Pentium-Systeme“
Praktischer Ringbuchordner, DIN A5, Grundwerk ca. 500 Seiten, inkl. zwei 5 1/4"-Disketten, Bestell-Nr.: 8800, Preis: DM 99,-,
☐ wahlweise zwei 3 1/2"-Disketten (gegen DM 4,- Aufpreis)
- ☐ Expl. „Raffinierte Lösungen in Turbo Pascal“
Stabiler Ringbuchordner, DIN A4, Grundwerk ca. 670 Seiten, inkl. 5 1/4"-Diskette, Bestell-Nr.: 2500, Preis: DM 99,-,
☐ wahlweise 3 1/2"-Diskette (gegen DM 2,- Aufpreis)
- ☐ Expl. „DOS-Systemprogrammierung für IBM-PCs und Kompatible“
Stabiler Ringbuchordner, DIN A4, Grundwerk ca. 880 Seiten, inkl. zwei 5 1/4"-Disketten, Bestell-Nr.: 5000, Preis: DM 99,-,
☐ wahlweise 3 1/2"-Diskette (gegen DM 2,- Aufpreis)

Alle 2-3 Monate erhalten Sie Erweiterungsausgaben zu den in der Anzeige angegebenen Preisen (Abbestellung jederzeit möglich).

IN 190235

Bitte ausschneiden, auf eine ausreichend frankierte Postkarte kleben oder im Kuvert
senden an: INTEREST-Verlag GmbH, Hofrat-Röhler-Straße 7, 86161 Augsburg

Meine Anschrift:

Name/Firma

Vorname/Ansprechpartner

Straße, Haus-Nr.

PLZ, Ort

X

Datum

Unterschrift

Bitte unterschreiben Sie auch Ihre Sicherheitsgarantie, mit der Sie folgendes zur Kenntnis nehmen: Sie haben das Recht, Ihr angefordertes Werk innerhalb von 10 Tagen ab Lieferung an den INTEREST-Verlag GmbH, Hofrat-Röhler-Straße 7, 86161 Augsburg, zurückzusenden, wobei für die Fristwahrung das Datum der Absendung genügt. Sie kommen dadurch von allen Verpflichtungen aus der Bestellung frei.

X

Datum

Unterschrift