ATARI ST COMPUTER

Die Fachzeitschrift für ATARI ST- und TT-Anwender

DM 14,-
Ös. 112,-
Sfr. 14,-
Lit. 10000,-

SONDERHEFT PROGRAMMIER-PRAXIS

ASSEMBLER

BASIC

- GEM-Programmierung
  - Grundlagen
  - Tips & Tricks
- Timer & Interrupts
- Schnelle Grafik
- Sprachen-erweiterungen
- XBRA & Cookie-Jar
- CPX-Module
- Programmier-Utilities

Zum Verständnis werden zusätzlich 2 Disketten (DM 15,-) benötigt!
(Siehe Seite 3)

Die Disketten-frage

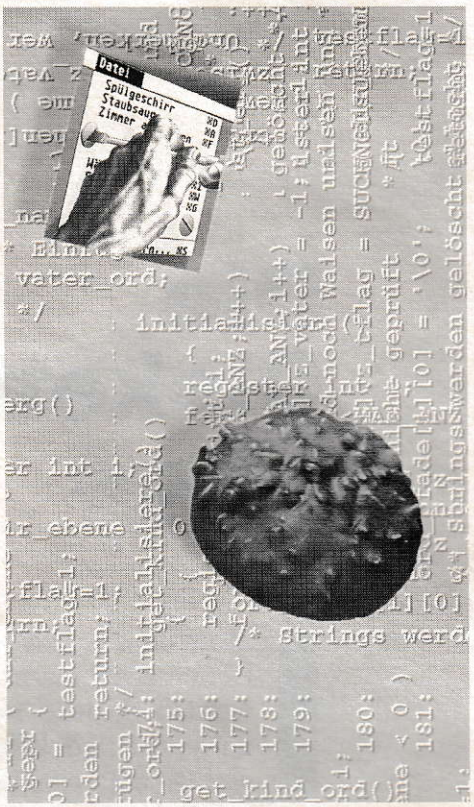
Im Laufe der Zeit sammeln sich beim Programmieren eine Reihe von Routinen an. Manche sind nur auf eine spezielle Aufgabe ausgelegt und werden deshalb auch nur ein einziges Mal eingesetzt. Andere sind universell und können in fast allen Programmen wieder verwendet werden (z.B. eine Window-Bibliothek). Wir haben Ihnen in diesem Sonderheft eine Reihe von nützlichen Routinen und Bibliotheken zusammengestellt. Damit Sie diese sofort, ohne sie erst lange abtippen zu müssen, in Ihren eigenen Programmen anwenden könnnen, haben wir auf einen Abdruck der Listings verzichtet. Der dadurch gewonnene Platz kam weiteren Programmiertips zugute. Die einzelnen Routinen finden Sie, nach den Sprachen sortiert, auf den Disketten zum Heft. Wir haben uns lange überlegt, ob wir die Disketten dem Heft beilegen sollen. Aus mehreren Gründen haben wir uns dagegen entschieden. Die Routinen sind für drei Programmiersprachen (C, PASCAL und BASIC) vorhanden; es müßten demnach sechs Disketten dem Heft beigelegt werden. Abgesehen von dem zusätzlichen Gewicht, wäre es sicherlich nicht in jedermans Interesse, für sechs Disketten zu zahlen, wenn man die Routinen nur für eine Programmiersprache benötigt. Ein weiterer Grund liegt darin, daß nie alle Hefte einer Auflage verkauft werden. Die zurückgehenden Examplare werden normalerweise beim Zwischenhändler gesammelt und nur die Deckblätter an den Verlag zurückgeschickt. Bei beiliegenden Disketten müßte aber jeweils das gesamte Heft an den Verlag zurückgehen! Diesen Mehrkostenaufwand müßte auf den Preis dieses Sonderheftes umgerechnet werden, womit sicherlich keinem gedient wäre.

Sie finden im Heft einen Bestellschein zum Bezug der Disketten. Alle Routinen (mit Ausnahme der Routinen aus dem Spracherweiterungsteil) sind in den drei Programmiersprachen erhältlich. Außerdem befinden sich die vorgestellten Utilities auf den Sprachendisketten. So können Sie die Routinen direkt bei der Entwicklung Ihres nächsten Programms verwenden.

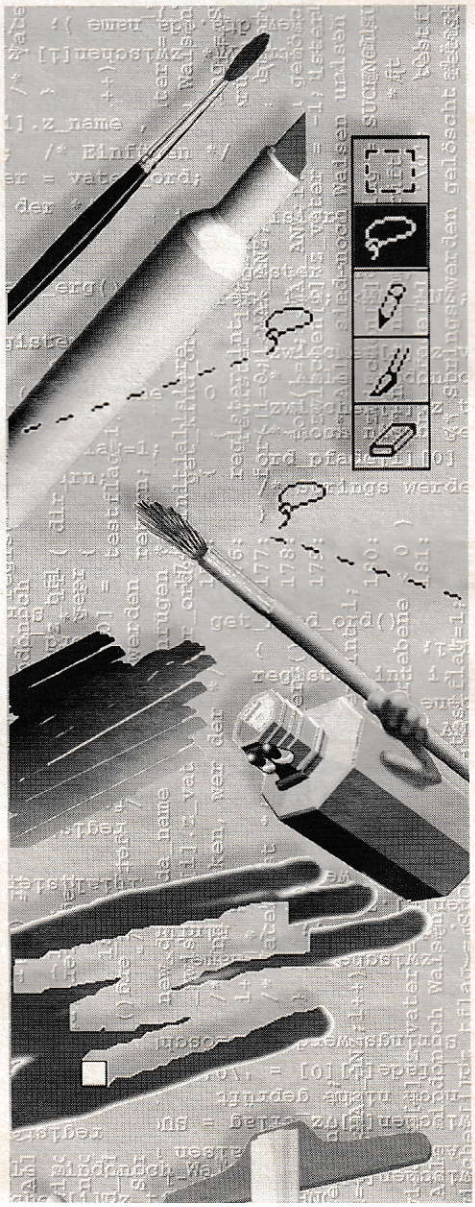
Thomas Werner

Inhalt

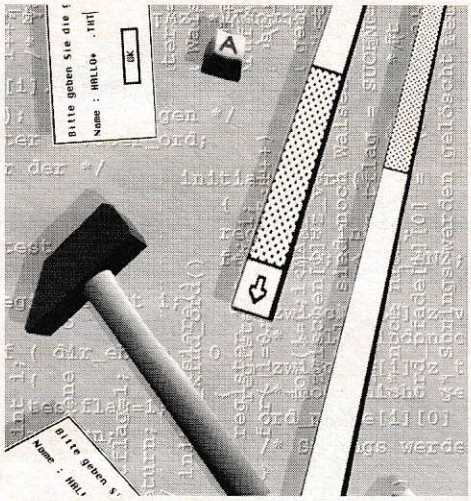
Grundlagen



Grafikroutinen

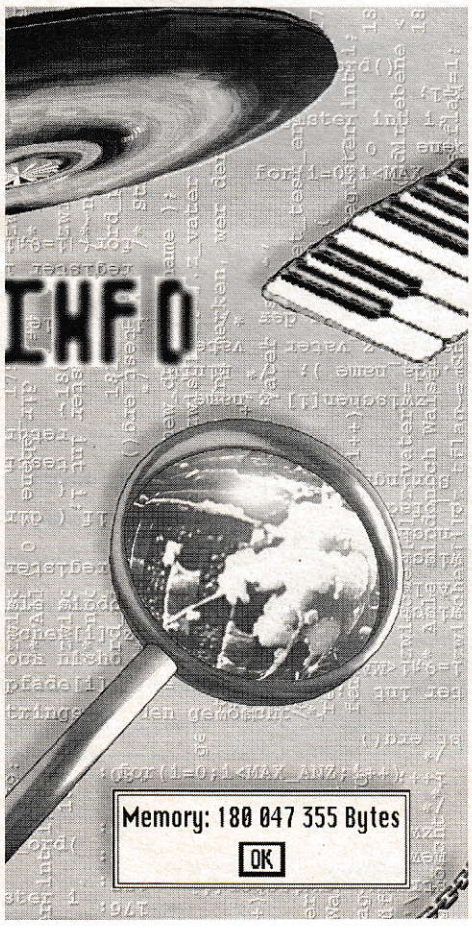


System



Sprachen

Utilities



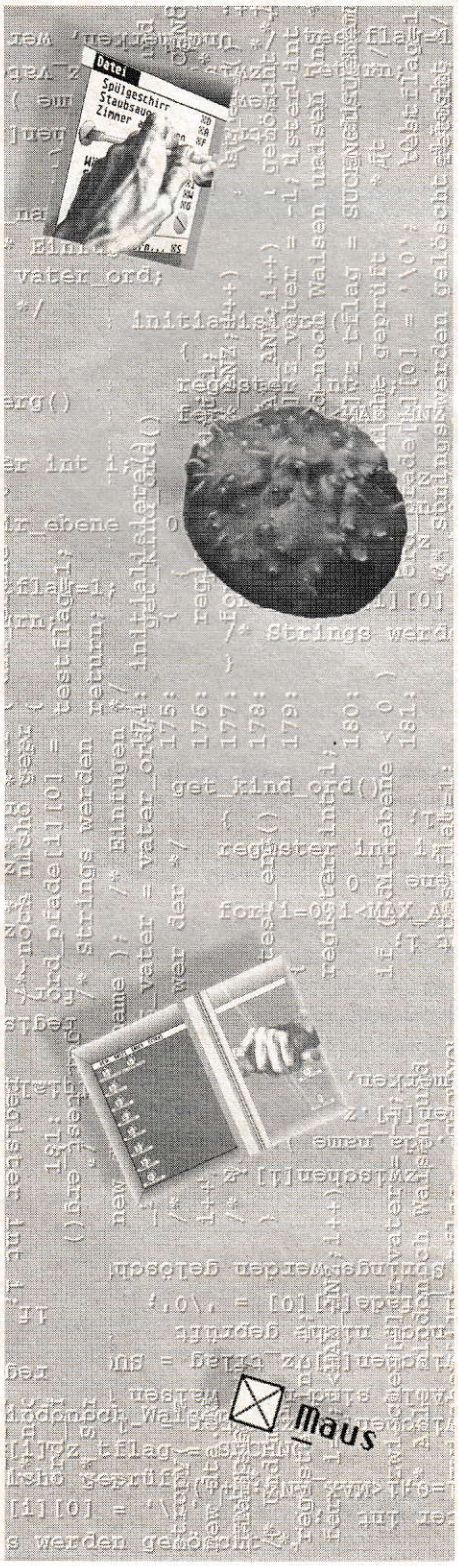
Marktübersicht

Mit der folgenden Übersicht wollen wir Ihnen eine Auswahl über die auf den ATARI-Rechnern verfügbaren Programmiersprachen, Erweiterungen und Utilities geben. In den drei Bereichen finden Sie jeweils kommerzielle und Public-Domain-Programme.

Programm	Art	Akt.Vers.	Anbieter	Preis
Easy Rider	68000/68030er-Assembler	3.00/3.52	Andreas Borchard, Osnabrück	199,- (425,- inkl. Reass.)
GFA-BASIC	BASIC-Interpreter/Compiler	3.6/3.6	GFA Systemtechnik, Düsseldorf	268,- (inkl. Compiler)
Omikron.BASIC	BASIC-Interpreter/Compiler	3.5/3.5	OMIKRON, Pforzheim	69,-/229,-
Lattice C	C	5.5	CCD, Eltville	398,-
Prospero C-Compiler	C	1.144	EDV-Beratung Plünnecke, Lengede	a.A.
Pure C	C	29.08.91	Application Systems, Heidelberg	398,-
Prospero Fortran	FORTRAN	2.153	EDV-Beratung Plünnecke, Lengede	a.A.
FTL Modula-2	MODULA-2	1.18	CCD, Eltville	299,-
Hänisch Modula-2	MODULA-2	4.0	Modular System GbR, Würzburg	349,-
SPC Modula-2	MODULA-2	2.0	ATARI, Raunheim	399,-
Maxon PASCAL	PASCAL	1.5	MAXON Computer, Eschborn	259,-
Prospero PASCAL	PASCAL	2.153	EDV-Beratung Plünnecke, Lengede	a.A.
ST Pascal plus	PASCAL	2.10	CCD, Eltville	149,-
Maxon PROLOG	PROLOG	1.1	MAXON Computer, Eschborn	298,-
TurboASS	Assembler/Debugger	1.70	ST-PD 283	Shareware
GNU C++	C	1.39/1.4	ST-PD 438-442	PD
SOZOBON C	C	11/12/88	ST-PD 240/241	PD
VolksFORTH83	FORTH	3.80	ST-PD 49	PD
FORTRAN 77	FORTRAN	1.3C	ST-PD 347	PD
XLISP	LISP	1.7	ST-PD 7	PD
MODULA 2	MODULA-2	1.4	ST-PD 225	PD
Edison	Editor	1.00	KNISS Soft, Aachen	169,-
Emacs	Editor	3.9+	ST-PD 226	PD
Tempus	Editor	2.11	CCD, Eltville	97,-
ACS	Application Construction Set	1.02	MAXON Computer, Eschborn	198,-
BASIC nach C PRO	Sourcecode-Konverter	2.02	CICERO, Mandelbachtal	333,-
ergo!	GFA-BASIC-Shell		Heim Verlag, Darmstadt	148,-
Interface	Resource Construction Set	1.07	Shift, Flensburg	99,-
OMLib professional	Dialogbox-Library (Omikron)		Hüthig Buch Verlag, Heidelberg	129,-
Roger-Tools	GFA-BASIC-Library		Bela-Computer, Eschborn	59,-
Structo	GFA-BASIC-Cross-Referenz	1.01	Ulli Ramps, Berlin	
Wega-Developer-Kit	GEM-Extended-Library (C)		Dietmar Rabich, Dülmen	50,-

Grundlagen

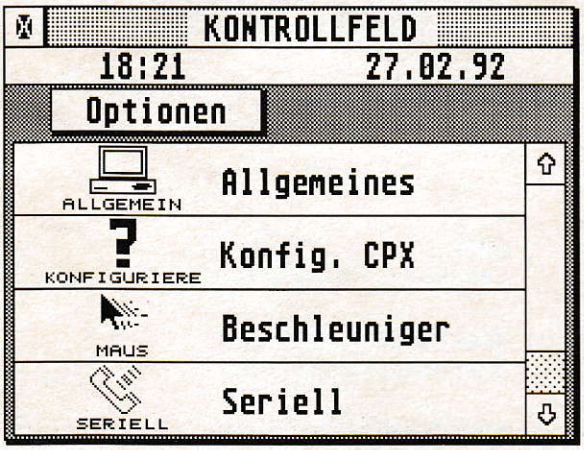
Bevor man sich an das Entwickeln eigener Programm begibt und dazu die in diesem Heft veröffentlichten Programmiertips und -kniffe verwendet, sollte man sich ein Grundwissen über die Programmierung des Rechners angeeignet haben. In dieser ersten Rubrik finden Sie deshalb einige Grundlagenartikel. Sie behandeln neben rechnerspezifischen Themen (CPX, Cookie-Jar, Timer und Interrupts, XBRA, ...) auch das Betriebssystem der ATARI-Rechner (Programme unter GEM, Window-Bibliothek, Submenüs). Die Beiträge können Ihnen zum Schreiben eigener Programme interessante Hilfestellungen und Tips vermitteln.



Programmierung von CPX-Modulen

Neben der Überarbeitung der Hardware und des Betriebssystem hat ATARI den STE- und TT-Rechnern auch ein neues Kontrollfeld spendiert. Es ist modular aufgebaut und kann somit auch vom Anwender erweitert werden.

tw



Zusammen mit dem Steuerprogramm XControl, welches für die Verwaltung der einzelnen Module zuständig ist und Funktionen zur Verfügung stellt, werden von ATARI bereits Module zur Rechnerkonfiguration mitgeliefert. Das modulare Kontrollfeld kann auch auf den 'alten' ST-Modellen verwendet werden; ATARI gibt dafür allerdings keine 100% Funktionsgarantie.

Das Kontrollfeld und die zusätzlichen Module (bis zu 99 können von XControl verwaltet werden) dienen außschließlich der Konfiguration des Rechners, von Programmen (z.B. Mausbeschleuniger) und Zusatz-Hardware (z.B. Grafikkarten) und sollten auch zu keinem anderen Zweck verwendet werden.

Wie funktioniert CPX?

Nach dem Start des Rechners wird XControl geladen. Dieses sucht im CPX-Verzeichnis nach aktiven CPX(**C**ontrol **P**anel e**X**tension)Modulen und lädt deren Header ein. Eventuell wird anschließend eine Initialisierungsroutine aufgerufen, die jedem Modul die Möglichkeit zur Konfiguration gibt. Danach wartet XControl, wie jedes Accessory, auf seine Aktivierung durch Anwahl in der Menüzeile. Wird ein Modul durch Auswählen in XControl aktiviert, wird zuerst seine Initialisierungroutine eine zweiten Mal aufgerufen. Hierbei müssen globale Variablen initialisiert werden. Darunter fallen auch sämtliche Resource-Informationen, da diese im Modul enthalten sein müssen [ein späteres Laden mittels *rsrc_load()* ist, durch die damit verbundene Zerstörung der Resource-Daten des momentan aktiven (Haupt-) Programms, nicht möglich]. Anschließend übergibt XConrol die Kontrolle an das Modul durch Aufruf seiner Hauptroutine.

Modul-Aufbau

Wie der obigen Beschreibung entnommen werden kann, unterscheidet sich der Aufbau eines Moduls von dem eines normalen Programms. Bei einer Modul-Datei befinden sich an erster Stelle der CPX-Header, 512 Bytes lang (Aufbau siehe Bild 1). Danach folgen der GEM-DOS-Programmheader, das Text- und Daten-Segment sowie Relozierungsinformationen. Der CPX-Header enthält wichtige Information über das Modul, die von XControl benötigt werden. Neben den Informationen über die Darstellung innerhalb des Kontrollfeldes befinden sich auch Daten über die Art des Moduls darin. Ein Flag zeigt z.B. an, ob das Modul resident im Speicher gehalten werden soll (dann wird bei der Initialisierung nicht nur der Header, sondern das ganze Modul in den Speicher geladen!) oder nicht. Wenn es sich um ein Set-Only-Modul handelt (es werden nur bei der Initialisierung Daten an eine Hardware oder ein Programm übergeben und das Modul danach nicht mehr aufgerufen) oder bei der Initialisierung die Routine des Moduls nicht mit aufgerufen werden soll, findet XControl auch diese Informationen innerhalb des Headers. Dieser Header muß für jedes Modul erzeugt werden. Von ATARI können eingetragene Entwickler ein Tool zur Erzeugung des Headers beziehen. Auf den Disketten zum Heft befindet sich ein von Uwe Hax und Oliver Scholz entwickelter Header-Generator.

CPX-Programmierung

Bei der Entwicklung eigener Module sollte man einige Programmierrichtlinien beachten:

- Die Module werden zusätzlich zu einem momentan laufendem Programm aufgerufen; sie sollten deshalb reservierten Speicher möglichst schnell wieder freigeben. Eventuell genügt ja der von XControl reservierte und dem Modul als einziger nicht flüchtiger Speicher zur Verfügung stehende Puffer von 64 Bytes Größe.
- Der Aufbau und das Aussehen der Module sollte sich an den bereits existierenden orientieren, damit dem Benutzer die Bedienung, durch eine einheitliche Oberfläche, leichter fällt.
- Buttons, die zur Darstellung von Pop-Up-Menüs führen, sind als schattierte Rechtecke darzustellen.
- Das erste Objekt eines Moduls kann die maximale Größe von 256 * 176 Pixeln besitzen (Arbeitsgröße von XControl).
- Interrupt-Vektoren dürfen nicht verändert werden.

Zuerst spielte er nach Noten dann seine eigenen Melodien und jetzt sahnt er bei der GEMA ab.

Daß ATARI ST Computer die Nr.1 im Musikbereich sind, ist unbestritten. Das MIDI-Interface gehört nun einmal zur serienmäßigen Ausstattung dieses auch in anderen Bereichen erfolgreichen Computers. Es gibt noch eine ganze Reihe weiterer guter Gründe. Da ist die beispielhafte Monochromdarstellung, die Noten exakt lesbar macht. Die Vielzahl an hervorragenden Programmen, denen nur Ihre musikalische Kreativität die Grenzen setzt. So steuern Sie mit Hilfe der gängigen Multitasking-Betriebssysteme gleichzeitig z.B. eine Mischpultautomation, lassen den Sequenzer laufen und verwalten zudem Ihre Sample-Sounds. Und über die Druckerschnittstelle erhalten Sie als Noten, was Sie somit Schwarz auf Weiß als Ihr eigenes Werk vielleicht zu hohen Ehren kommen läßt.

Wann erweitern Sie Ihr musikalisches Repertoire mit einem ATARI STE Computer? Sprechen Sie mit Ihrem ATARI MIDI/Musik-Fachhändler darüber. Das ist:

ATARI
und Musik

ATARI MEGA STE
1 oder 2 oder 4 MB RAM
integrierte Floppy und Festplatte
serienmäßig mit flimmerfreiem
s/w Monitor 71 Hz
ATARI SLM 605
Laserdrucker für
gestochen scharfe s/w Ausdrucke

Für ATARI STE Computer
gibt es Sequenzer und
Editoren von C-Lab,
Soft Arts und Steinberg,
Lernsoftware von Schott
und Harddiskrecording
von Hybrid Arts,
um nur einige zu nennen.

ATARI Computer GmbH
Postfach 12 13
6096 Raunheim

ATARI
...wir machen Spitzentechnologie preiswert

ATARI und Musik • ATARI und Textverarbeitung • ATARI und Datenbanken • ATARI und Spaß mit Grips •
ATARI und DeskTop Publishing • ATARI und Büro • ATARI und Studium • ATARI und Wissenschaft • ATA

- Die Buttons 'OK' und 'Abbruch'/'Cancel' sind, soweit dies sinnvoll ist, immer zu implementieren.
- Die CPX-Id muß eindeutig sein, d.h. es darf keine zwei Module mit derselben Kennung geben (wenn Sie eine neue Version Ihres Moduls entwickelt haben, behalten Sie die ID bei und ändern die Versionsnummer. XControl aktiviert bei Modulen mit derselben ID jenes, welches die neueste Versionsnummer besitzt).
- Die erste Routine in Ihrem Modul muß die Initialisierungs-Routine sein (Sie können auch einen Sprungbefehl auf diese Routine an die erste Stelle setzen). Diese muß, unter Auswertung des bootinit-Flags, zwei Arten der Initialisierung durchführen können: zum einen die Initialisierung beim Booten und zum anderen die eigentliche Modul-Initialisierung vor dem Aufruf der Hauptroutine. Der Initialisierung-Routine wird ein Zeiger auf die von XControl bereitgestellten Funktionen übergeben (siehe Bild 2; XCPB-Struktur). Dieser muß zwischengespeichert werden, da nur über diese Struktur auf die XControl-Routinen zugegriffen werden kann. Als Rückgabewert liefert die Initialisierungs-Routine einen Zeiger auf die vom Modul bereitgestellten Funktionen (siehe unten).

XControl-Funktionen

Bei den vom Kontrollfeld zur Verfügung gestellten Funktionen handelt es sich um Routinen zur Objektanpassung (rsh_...), zur Bearbeitung von Pop-Up-Menüs (Popup), zur einfachen Slider-Bearbeitung (Sl_...), zur Formular- und Dialog-Verwaltung (*Xform_do*,*XGen_Alert*, *Set_Evnt_Mask*, *Get...Rect*, *MFsave*, *CPX_Save*), zur Zwischenspeicherbearbeitung (*Get_Buffer*) und zur Cookie-Jar Abfrage (*getcookie*). Durch Ausnutzung dieser Routine erhält der Programmierer die Möglichkeit kompakte Module zu entwickeln. Besonders die Funktionen zur Slider- und Pop-Up-Menü-Verarbeitung sind sehr hilfreich. Letztere übernimmt z.B. die komplette Abwicklung von Pop-Ups. Sie wird nach Anwahl des Pop-Up-erzeugenden Buttons mit einer Liste der Menüeinträge aufgerufen. Sie verwalten nun vollständig die Selektion eines Eintrags. Bei mehr als 5 Einträgen wird der erste und letzte durch Pfeile ersetzt, und die Liste kann gescrollt werden.

Modul-Funktionen

Die Modul-Initialisierungs-Routine (siehe Tabelle 2) muß, wie oben bereits erwähnt, einen Zeiger auf eine Liste der modul-eigenen Routinen zurückliefern. Da XControl nicht bekannt ist, wo die einzelnen Routinen eines Moduls sich befinden (abgesehen von der Initialisierungs-Routine, die direkt am Anfang des Text-Segments stehen muß), kann es nur über diese Liste auf die Routine zugreifen. Die vom CPX-Modul zur Verfügung zu stellenden Routinen können Tabelle 3 entnommen werden. Es handelt sich dabei um 'Reaktions'-Routinen, die dann aufgerufen werden, wenn eine Operation des Moduls notwendig wird. Ein recht praktischer Umstand von XControl, der bei der Entwicklung von neuen CPX-Modulen wirksam wird, ist die Tatsache, daß man XControl auch als Programm starten kann. Dazu muß es von XCONTROL.ACC in XCONTROL.APP umbenannt werden. Nun können Module ausgetestet werden, ohne jedes Mal einen Reset ausführen zu müssen.

Aufruf:	rsh_fix(num_objs, num_frst, num_frimg, num_tree, rs_object, rs_tedinfo, rs_string[], rs_iconblk, rs_bitblk, rs_frstr, rs_frimg, rs_trindex, rs_imdope)
Funktion:	Umwandlung der Koordinatendarstellung eines Objektbaums von Zeichen- in Pixel-Darstellung auf der Basis von 8 * 16 Pixel großen Zeichen.
Parameter:	num_...: die entsprechenden Konstanten aus *.RSH (wird vom RCS angelegt) rs_...: Zeiger auf die gleichnamigen Strukturen aus *.RSH
Rückgabe:	keine
Aufruf:	rsh_obfix(tree, curob)
Funktion:	Umwandlung der Koordinatendarstellung eines Objekts von Zeichen- in Pixel-Darstellung auf der Basis von 8 * 16 Pixel großen Zeichen [entspricht rsrc_obfix()]
Parameter:	tree: Zeiger auf umzuwandelnden Objektbaum curob: Nummer des zu konvertierenden Objekts
Rückgabe:	keine
Aufruf:	Popup(items[], num_items, default_item, font_size, button, world)
Funktion:	Automatische Verwaltung eines Pop-Up-Menüs. Bei mehr als 5 Einträgen wird selbständig gescrollt.
Parameter:	items[]: Zeiger auf Liste der Menüeinträge. Alle Einträge müssen gleich lang sein. Außerdem sollten jedem Eintrag zwei Leerzeichen voran- (für Haken) und ein Leerzeichen nachgestellt werden. num_items: Anzahl der Menüeinträge default_item: Nummer des vorgewählten (abgehakten) Eintrags (Zählung beginnt mit 0; bei -1 wird kein Eintrag vorgewählt). font_size: Auswahl der Zeichengröße: 3: groß (8*16) 5: klein (8*8) button: Zeiger auf Rechteckstruktur GRECT) des Pop-Up-Menü-Buttons world: Zeiger auf Rechteckstruktur des CPX
Rückgabe:	angeklickter Eintrag; -1, wenn keiner angewählt wurde
Aufruf:	Sl_size(tree, base, slider, num_items, visible, direction, min_size)
Funktion:	berechnet Größe des darzustellenden Sliders im Verhältnis zur Gesamtzahl der Listeneinträge
Parameter:	tree: Zeiger auf Objektbaum base: Nummer des Slider-Hintergrundobjekts slider: Nummer des Slider-Objekts (Child von base) num_items: Gesamtanzahl der Listeneinträge visible: Anzahl der darstellbaren Einträge

→

direction: Richtung:
0: vertikal
1: horizontal
min_size: minimale Pixel-Größe des Sliders

Rückgabe: keine

Aufruf: SI_x(tree, base, slider, value, min, max, foo)
SI_y(tree, base, slider, value, min, max, foo)

Funktion: Positionierung des Sliders

Parameter: tree: Zeiger auf Objektbaum
base: Nummer des Slider-Hintergrundobjekts
slider: Nummer des Slider-Objekts (Child von base)
value: neuer Wert für Slider(-Position)
min: minimaler Wert
max: maximaler Wert
foo: Zeiger auf eine Funktion, die bei der Slider-Neupositionierung mit aufgerufen wird

Rückgabe: keine

Aufruf: SI_arrow(tree, base, slider, obj, inc, min, max, value, direction, foo)

Funktion: bearbeiten die Pfeil, bzw. Hintergrundanwahl innerhalb der Slider-Bearbeitung

Parameter: tree: Zeiger auf Objektbaum
base: Nummer des Slider-Hintergrundobjekts
slider: Nummer des Slider-Objekts (Child von base)
obj: Nummer des angewählten Objekts (Pfeil, Hintergrund)

→

```
typedef struct
{
   WORD  magic    /* CPX-Kennung = 100 */
   struct
   {
      unsigned reserved:   13; /* reserviert */
      unsigned resident:   1;/* Modul ist resident */
      unsigned booting: 1; /* Boot-Initialisierung */
      unsigned setonly: 1; /* Set-Only-Modul */
   } flags;
   LONG  cpx_id;            /* eindeutige CPX-ID
                               (4 ASCII-Zeichen) */
   WORD  cpx_version /* CPX-Versionsnummer */
   char  i_text[14]; /* Icon-Name */
   WORD  sm_icon[48]; /* Icon-Bitmap (32*24 Pixel) */
   WORD  i_color;    /* Icon-Farbe */
   char  title_text[18];/* Texte neben Icon */
   WORD  t_color:    /* Textfarbe */
   char  buffer[64]; /* nicht flüchtiger Speicher */
   char  reserved[306]; /* reserviert */
} CPXHEAD;
```

Bild 1: Struktur des CPX-Headers

```
typedef struct
{
   WORD  handle;     /* von graf_handle()-Aufruf */
   WORD  booting;    /* Boot-Initialisierung */
   WORD  reserved;      /* reserviert */

   void  *reserve1;  /* reserviert */
   void  *reserve2;  /* reserviert */
   ...
   /* Definition der von XContol bereitgestellten
            Funktionen */
   ...
   WORD  Country_Code;  /* Länderkennung */
} XCPB;
```

Bild 2: Aufbau der XCPB-Struktur

inc: Anzahl der zu 'blätternden' Listeneinträge
min: minimaler Wert
max: maximaler Wert
value: Adressen für aktuellen Wert
direction: Richtung:
0: vertikal
1: horizontal
foo: Zeiger auf eine Funktion, die bei der Slider-Neupositionierung mit aufgerufen wird

Rückgabe: keine

Aufruf: Sl_dragx(tree, base, slider, min, max, value, foo)
Sl_dragy(tree, base, slider, min, max, value, foo)
Funktion: Verwaltung des Verschiebens eines Sliders bei gedrückter Maustaste
Parameter: tree: Zeiger auf Objektbaum
base: Nummer des Slider-Hintergrundobjekts
slider: Nummer des Slider-Objekts (Child von base)
min: minimaler Wert
max: maximaler Wert
value: Adressen für aktuellen Wert
foo: Zeiger auf eine Funktion, die bei der Slider-Neupositionierung mit aufgerufen wird
Rückgabe: keine

Aufruf: Xform_do(tree, startob, puntmsg)
Funktion: Formular-Verwaltung [ähnlich form_do()]
Parameter: tree: Zeiger auf zu verwaltenden Obkjektbaum
startob: Startobjekt
puntmsg: Zeiger auf Mitteilungspuffer
Rückgabe: Nummer des angeklickten Objekts; bei -1 enthält puntmsg eine der folgenden vier Mitteilungen:
WM_REDRAW(20): Objekte, die nicht zum Objektbaum gehören (und deshalb nicht automatisch neugezeichnet werden), müssen selbst erneuert werden [die dazu notwendige Rechteckliste liefern die Funktionen GetFirstRect() und GetNextRect(); siehe unten]
WM_CLOSE(22): Das Modul wurde mit OK beendet.
AC_CLOSE(41): Das Modul wurde mit Abbruch beendet.
CT_KEY(53): Die Tasten HELP, UNDO oder eine Funktionstaste wurden gedrückt (High-Byte von puntmsg[3] enthält Scan- und Low-Byte ASCII-Code).

Aufruf: GetFirstRect(prect)
Funktion: liefert erstes Element der Rechteckliste des neuzuzeichnenden Bereichs
Parameter: prect: Zeiger auf Rechteckstruktur (GRECT) des zu aktualisierenden Bereichs
Rückgabe: Zeiger auf Rechteckstruktur des zu restaurierenden Bereichs; NULL: kein Bereich vorhanden

Aufruf: GetNextRect(prect)
Funktion: liefert nächstes Element der Rechteckliste des neuzuzeichnenden Bereichs
Parameter: keine
Rückgabe: Zeiger auf Rechteckstruktur des zu restaurierenden Bereichs; NULL: keine weiteren Bereiche vorhanden

Aufruf: Set_Evnt_Mask(mask, m1, m2, time)
Funktion: bestimmt, auf welche Events ein Event-CPX reagieren soll
Parameter: mask: Events [wie evnt_multi()]
m1, m2: Zeiger auf Struktur, die Mausrechteck und -richtung für Event angibt
time: Zeit (Millisekunden) für Timer-Event [damit keine Programme blockiert werden (das CPX-Modul läuft zusätzlich zu einem anderen Programm (oder zum Desktop), erfolgt spätestens nach 30 Sekunden ein Timerevent]
Rückgabe: keine

Aufruf: XGen_Alert(id)
Funktion: Erzeugt vordefinierte Alarmboxen; weitere müssen selbst erzeugt werden. [form_alert() ist dazu nicht sinnvoll, da damit erzeugte

→

	Alarmboxen bezüglich des Bildschirms und nicht der CPX-Box zentriert werden].
Parameter:	id: Nummer der darzustellenden Alarmbox: 0: SAVE_DEFAULTS (Voreinstellungen sichern?) 1: MEM_ERR (Fehler bei Speicheranforderung!) 2: FILE_ERR (Fehler beim Schreiben/Lesen von Datei!) 3: FILE_NOT_FOUND (Datei nicht gefunden!)
Rückgabe:	0: Abbruch wurde angewählt; sonst wurde OK angeklickt
Aufruf:	CPX_Save(ptr, num)
Funktion:	Default-Parameter werden in das DATE-Segment des aktuellen CPX gespeichert
Parameter:	ptr: Zeiger auf die zu speichernden Daten num: Byte-Anzahl der zu speichernden Daten
Rückgabe:	0: ein Fehler ist aufgetreten; sonst: alles ok
Aufruf:	Get_Buffer()
Funktion:	ermittelt Zeiger auf einen 64 Byte großen CPXeigenen Speicherbereich
Parameter:	keine
Rückgabe:	Zeiger auf Speicherbereich
Aufruf:	getcookie(cookie, p_value)
Funktion:	sucht nach gewünschtem Cookie und liefert, sofern vorhanden, seinen Wert
Parameter:	cookie: Cookie-ID (vier ASCII-Zeichen) p_value: Zeiger zur Ablage des Cookie-Wertes
Rückgabe:	0: Cookie nicht gefunden; sonst: Cookie gefunden
Aufruf:	MFsave(saveit, mf)
Funktion:	Form des Mauszeigers kann zwischengespeichert und wieder hergestellt werden
Parameter:	saveit: bestimmt Operationsart: 0: MFRESTORE (Mausform restaurieren) 1: MFSAVE (Mausform zwischenspeichern) mf: Zeiger auf einen Speicherbereich zur Ablage der Mauszeigerdaten
Rückgabe:	keine

Tabelle 1: Liste der von XControl bereitgestellten Funktionen

Aufruf:	cpx_init(xcpb)
Funktion:	Funktion muß die erste im Modul sein. Sie wird von XControl zur Initialisierung aufgerufen.
Parameter:	xcpb: Zeiger auf XCPB-Struktur (wird dem Modul geliefert!)
Rückgabe:	Zeiger auf die CPXINFO-Struktur, oder NULL, wenn 'Set-Only'-Modul (liefert das Modul zurück!)

Tabelle 2: Initialisierungs-Funktion der CPX-Module

Aufruf:	cpx_call(word)
Funktion:	Hauptroutine; Aufruf von XControl nach Anwahl des Moduls
Parameter:	word: Rechteckstruktur (GRECT) der Arbeitsfläche des XControl-Fensters
Rückgabe:	0: Ende der Bearbeitung; sonst: CPX soll weiterbearbeitet werden
Nachfolgende Routinen werden nur von Event-CPX benötigt:	
Aufruf:	cpy_draw(clip)
Funktion:	Aufruf bei einem Redraw
Parameter:	clip: Zeiger auf Rechteckstruktur des neuzuzeichnenden Bereichs
Rückgabe:	keine →

Aufruf:	cpx_wmove(work)
Funktion:	XControl-Fenster wurde bewegt
Parameter:	work: Zeiger auf Rechteckstruktur mit den neuen Fensterkoordinaten
Rückgabe:	keine
Aufruf:	cpx_timer(event)
Funktion:	Aufruf bei Timerevent
Parameter:	event: Zeiger auf Ausgabewert: 1: CPX soll verlassen werden
Rückgabe:	keine
Aufruf:	cpx_key(kstate, key, event)
Funktion:	Aufruf bei Keyboardevent
Parameter:	kstate: Status der Umschalttasten key: gedrückte Taste (High-Byte: Scancode, Low-Byte: ASCII-Code) event: Zeiger aus Ausgabewert: 1: CPX soll verlassen werden
Rückgabe:	keine
Aufruf:	cpx_button(mrets, nclicks, event)
Funktion:	Aufruf bei Maustasten-Event
Parameter:	mrets: Zeiger auf Mausparameter nclicks: Anzahl der Maustastenklicks event: Zeiger aus Ausgabewert: 1: CPX soll verlassen werden
Rückgabe:	keine
Aufruf:	cpy_m1(mrets, event) cpy_m2(mrets, event)
Funktion:	Aufruf, wenn Mauszeiger bestimmte Rechtecke verläßt oder betritt
Parameter:	mrets: Zeiger auf Mausparameter event: Zeiger aus Ausgabewert: 1: CPX soll verlassen werden
Rückgabe:	keine
Aufruf:	cpx_hook(event, msg, mrets, key, nclicks)
Funktion:	Aufruf direkt nach event_multi(), bevor XControl den Event bearbeitet
Parameter:	event: aufgetretenes Ereignis msg: Zeiger auf den Ereignispuffer mrets: Zeiger auf Mausparameter key: Tastendruck nclicks: Anzahl der Maustastenklicks
Rückgabe:	bestimmt weitere Verarbeitungsweise: 0: Event-Verarbeitung fortsetzen 1: Event-Verarbeitung abbrechen
Aufruf:	cpx_clos(flag)
Funktion:	wird bei jeder AC_CLOSE- und WM_CLOSE-Mitteilung aufgerufen; das Modul muß sofort eventuell reservierten Speicher freigeben
Parameter:	flag: Art der Mitteilung: 0: AC_CLOSE sonst: WM_CLOSE
Rückgabe:	keine

Tabelle 3: Liste der in CPXINFO aufgeführten Funktionen eines CPX-Moduls

Das XBRA-Verfahren

Stellen Sie sich vor, Sie haben ein Programm geschrieben, sagen wir einen Drucker-Spooler, das sich resident im Speicher installiert. Um die Daten in einen Puffer übernehmen zu können und sie dann aus diesem an den Drucker zu schicken, mußten Sie einige Systemvektoren des Betriebssystems 'verbiegen'.

tw

Sie haben also die Adresse, auf die der Vektor zeigte, ausgelesen und tragen die Startadresse Ihres Programms dort ein. Natürlich müssen Sie nach Beendigung Ihrer Routine zur alten Routine zurückspringen (siehe hierzu 'Timer und Interrupts' in diesem Sonderheft). Dies ist sicher kein großes Problem, da Sie ja die ursprüngliche Adresse ausgelesen haben.

Soweit, so gut. Solange Ihr Programm das einzige ist, das die entsprechenden Vektoren verbiegt, geht auch also gut. Stellen Sie sich nun aber weiter vor, daß ein zweites Programm die gleichen Vektoren verbiegen möchte. Kein Problem, sagen Sie? Im Prinzip haben Sie natürlich recht, aber wie überprüfen Sie bitteschön, ob Ihr Programm bereits installiert ist (z.B. bei einem zweiten Aufruf)? Und wie desaktivieren Sie Ihren Spooler, wenn sich ein zweites Programm sich dazwischen geklinkt hat?

Um diesen Problemen zu entgehen, wurde das XBRA-Verfahren 'entwickelt'. Es wurde erstmals 1988 von Julian Reschke im 'ATARI ST Profibuch' vorgestellt und hat sich mittlerweile durchgesetzt. Es besteht im Grunde genommen 'nur' daraus, eine verkettete Liste aufzubauen, die man komplett durchlaufen kann. Im Bild ist die XBRA-Struktur zu sehen. Sie wird direkt vor dem Anfang der neuen Routine plaziert. Sie besteht aus der XBRA-Kennung 'XBRA', einer vier Buchstaben langen Routinen-Kennung, und dem ausgelesenen ursprünglichen Wert des Vektors.

Mit Hilfe dieser Struktur ist es möglich, eine Kette von Routinen zu verfolgen. Dazu untersucht man die Zeiger jedes Systemvektors. yZeigen diese auf eine Routine, der die XBRA-Struktur vorausgeht, nimmt man den Zeiger aus der XBRA-Struktur und hat somit die Adresse der nächsten Routine (siehe hierzu Programm im Utility-Teil).

Sollten Sie Ihren Speicher voller vektor'verbiegender' Programme haben, kann der Fall eintreffen, daß Sie nicht alle Routinen einer Kette finden. Dann hält sich mit Sicherheit eines der installierten Programme nicht an das XBRA-Verfahren und unterbricht damit die 'Durchlaufbarkeit' der Kette.

Terminierung

Was kann man unternehmen, wenn man sein Programm aus einer Kette herausnehmen möchte? Wenn die Routine an erster Stelle der Kette steht, langt es, die gerettete Adresse aus *xb_oldvec* in die Systemvariable zu schreiben, die auf die zu terminierende Routine zeigt. Danach sollte man noch den reservierten Speicher freigeben und das Programm beenden. Eine Routine mitten aus einer XBRA-Struktur zu nehmen, ist auch kein Problem: Sie tragen die in Ihrer *xb_oldvec*-Variable stehende Adresse in die *xb_oldvec*-Variable ihres Vorgängers ein; das war alles.

Bei einer unterbrochenen XBRA-Struktur wird die Sache etwas aufwendiger. Hier müssen Sie einen kleinen (modifizierten) Rest Ihrer XBRA-Routine stehenlassen. Zum einen natürlich die XBRA-Struktur. Sie sollten aber die Routinen-Kennung (*xb_id*) unkenntlich machen (schließlich ist das folgende ja nicht mehr Ihre Original-Routine!). Als Routine benötigen Sie nun einen Programmcode, der direkt die in *xb_oldvec* adressierte Routine anspringt.

Zum Schluß nach ein Wort zu der Routinen-Kennung. Es ist einleuchtend, daß ein und dieselbe Kennung für zwei verschiedene Routinen nicht sehr hilfreich ist [wie kann ein Programm dann (einfach) feststellen, ob es bereits installiert ist?]. Zu diesem Zweck wird von Julian Reschke eine XBRA-Liste geführt. Nähere Auskünfte erhalten Sie bei ihm (siehe ATARI Profibuch ST-STE-TT).

```
typdef struct
{
   char  xb_magic[4];
         /* muß XBRA-Kennung enthalten: 'XBRA' */
   char  xb_id[4];
         /* vier (ASCII-)Zeichen zur Kennung der
            Routine */
   long  xb_oldvec;
         /* vorheriger Wert des Vektors */
} XBRA;
```

XBRA-Struktur in C

Das Cookie-Jar-Prinzip

Neben dem mittlerweile etablierten XBRA-Protokoll, das wir Ihnen im vorherigen Artikel vorgestellt haben, existiert seit TOS 1.6 (STE) der Cookie-Jar (die Keksdose). Dabei handelt es sich um eine Informationsstruktur. Mit ihrer Hilfe kann man verschiedene rechnerabhängige Informationen erhalten bzw. Informationen mit anderen Programmen austauschen.

Rolf Kotzian / tw

Der Cookie-Jar befindet sich irgendwo im Speicher des Rechners. Auf seine Startadresse zeigt die Systemvariable *_p_cookies* (Adresse: $5A0). Die einzelnen Kekse (Cookies) bestehen aus zwei 32-Bit-Werten (Im ATARI-Slang ist Cookie ein gebräuchlicher Terminus für eine 32 Bits große Codenummer). Der erste Wert stellt die Cookie-Kennung (*cookie_id*; siehe Bild 1) dar. Der zweite Wert ist abhängig vom jeweiligen Cookie (siehe Tabelle 1).

Da der Cookie-Jar erst ab TOS 1.6 vom Betriebssystem automatisch installiert (und beim Reset wieder gelöscht) wird, muß er bei früheren Versionen von einem residenten Programm installiert werden.

Die Theorie

Wenn man mit Cookies arbeiten will, muß man als erstes einmal nachsehen, ob es sie überhaupt gibt. Steht in der Systemvariablen *_p_cookies* ein NULL-Zeiger, bedeutet dies, daß (noch) kein Cookie-Jar installiert ist. Hat man einen Zeiger bekommen, kann man die Tabelle gezielt nach einem Cookie durchsuchen, um entsprechende Informationen zu erhalten, einen neuen Cookie einzutragen oder einen Eintrag zu löschen.

Sie fragen sich, was für einen Sinn solch eine Tabelle hat? Nun, zum einen erhält man einige wichtige Informationen über die Rechnerkonfiguration (siehe nochmal Tabelle 1), und zum zweiten kann man anderen Programmen mitteilen, ob ein bestimmtes Programm installiert ist. Dazu sollte, wie beim XBRA-Verfahren, die Cookie-ID individuell sein. Bei der Namensgebung sind drei Punkte zu beachten:

1. Die Kennung darf nicht mit '_' beginnen; dies hat sich ATARI für eigene Cookies reserviert.

2. Die vier Zeichen müssen ASCII-Zeichen (Code zwischen 32 und 126; keine Sonderzeichen) sein.

3. Die Kennung sollte eine Abkürzung ergeben, mit deren Hilfe man auf das Programm schließen kann.

Wie bereits bei der XBRA-Bezeichnung, wird auch beim Cookie-Jar von Julian Reschke eine Liste der bereits verwendeten IDs geführt.

Im Langwort *cookie_value* kann eine beliebige Information abgelegt werden, z.B. ein Zeiger auf eine interne Programmstruktur [über die andere Programme beispielsweise bestimmte Betriebsparameter verstellen können (dies ist beim Mausbeschleuniger MACCEL3 der Fall)], auf einen Text oder sonst etwas; der Phantasie sind hier keine Grenzen gesetzt!

Es versteht sich natürlich von selbst, daß die Bedeutung von *cookie_value* irgendwo dokumentiert sein sollte! (Was

nützt schon ein Eintrag, wenn man nicht weiß, wozu er zu gebrauchen ist?)

Die Aufmerksamen unter Ihnen werden jetzt wahrscheinlich fragen, wieviele Cookies denn eigentlich in den Jar hineinpassen? Diese Frage kann nicht pauschal beantwortet werden, da der Speicherbereich, auf den *_p_cookies* verweist, ja an beliebiger Stelle im Speicher steht und somit irgendwann mittels *Malloc* (oder einer ähnlichen Funktion) angefordert wurde - die Größe des Jars ist also variabel!

Um aber trotzdem das Ende der Tabelle erkennen zu können, existiert ein sogenannter NULL-Cookie (ID: $00000000), der als Wert die maximale Anzahl der in den Jar heinpassenden Cookies enthält. Außerdem ist er der letzte der aktuellen Cookie-Liste.

Die Praxis

Soweit die Theorie. Schauen wir uns einige Beispiele an, die die Nützlichkeit dieses Verfahrens noch einmal unterstreichen sollen. Besonders wichtig (im Hinblick auf portable Programmierung) ist es z.B., festzustellen, ob ein bestimmtes Programm auf einem normalen ST, einem getuneten Rechner (etwa mit einem 68020-Prozessor) oder gar auf dem TT läuft. Dank der vom BIOS installierten System-Cookies [erst ab TOS 1.6; vorher muß selbst installiert werden (siehe Programm auf der Dis-

Aufruf:	create_cookie(cookie, id, value)
Funktion:	initialisiert einen Cookie
Parameter:	cookie: Zeiger auf COOKIE-Struktur id: (ASCII-)Kennzeichnung des Cookies (vier Zeichen) value: Wert des Cookies
Rückgabewert:	keiner
Aufruf:	new_cookie (entry)
Funktion:	trägt neuen Cookie in den Jar ein
Parameter:	entry: Zeiger auf einzutragenden Cookie
Rückgabewert:	TRUE, wenn ok; FALSE im Fehlerfall
Aufruf:	get_cookie(id, value)
Funktion:	liefert den Wert eines Cookies
Parameter:	id: (ASCII-)Kennzeichnung des Cookies (vier Zeichen) value: Zeiger für Wert des Cookies
Rückgabewert:	TRUE, wenn ok; FALSE, wenn Cookie nicht existiert
Aufruf:	remove_cookie(id)
Funktion:	entfernt einen Cookie
Parameter:	id: (ASCII-)Kennzeichnung des Cookies (vier Zeichen)
Rückgabewert:	TRUE, wenn ok; FALSE, wenn Cookie nicht existiert
Aufruf:	move_cookiejar(dest, size)
Funktion:	verschiebt kompletten Jar an neue Speicherstelle; evtl. Vergrößerung der Anzahl der möglichen Einträge
Parameter:	dest: neue Adresse für Jar size: neue Größe des Jars
Rückgabewert:	keiner
Aufruf:	cookie_size()
Funktion:	liefert die Anzahl der maximal möglichen Einträge in den installierten Jar
Parameter:	keine
Rückgabewert:	Anzahl der maximal möglichen Cookies
Aufruf:	print_size()
Funktion:	gibt Inhalt des Jars aus Standardausgabe aus
Parameter:	keine
Rückgabewert:	keiner

kette)] ist es nun kein Problem mehr, festzustellen, auf welchem Rechner-Typ und mit welchem Prozessor das Programm läuft.

Aufgrund der außerordentlichen Nützlichkeit dieses Verfahrens bot es sich an, einige Hilfsroutinen zu schreiben, die den Umgang mit Cookies erleichtern sollen. Diese finden Sie, neben einem Beispiel, auf den Disketten zum Heft. Das Beispiel demonstriert einen Anwendungsfall von Cookies: Wie kann ein Programm feststellen, ob es sich bereits installiert hat? Einige der im Listing verwendeten Funktionen stammen aus dem Cookie-Modul; in der Praxis dürfte es sich als nützlich erweisen, dieses Modul getrennt zu übersetzen und in einer 'LIB'-Datei festzuhalten. Bei einer Programmentwicklung wird dann diese Datei automatisch vom Projektmanager an den Linker weitergereicht ...

Die Implementierung

Kommen wir aber zu den eigentlichen Hilfroutinen des Moduls. Sie übernehmen die wichtigsten Operationen, die beim Umgang mit Cookies auftreten können (Erzeugen, Eintragen, Abfragen, Löschen etc.).

Ein neuer Cookie kann ganz einfach mit Hilfe der Funktion *create_cookie()* erstellt werden; man übergibt dieser dazu lediglich einen Zeiger auf eine COOKIE-Struktur, die Programmmkennung (*cookie_id*) sowie den Wert des Cookies (long!). Der so erzeugte Cookie kann mittels *new_cookie* in den hoffentlich vorhandenen Cookie-Jar eingetragen werden.

Mit Hilfe von *get_cookie()* kann überprüft werden, ob ein bestimmter Cookie breits im System vorhanden ist (Return-Wert = TRUE); in diesem Fall wird auch gleich der zugehörige Wert in der Variablen value mitgeliefert.

Das Löschen von Cookies (das kann beispielsweise dann notwendig sein, wenn das Programm, das diesen installiert hat, sich wieder aus dem Speicher entfernen will) übernimmt die Funktion *remove_cookie()*. Gelöscht wird übrigens einfach dadurch, daß - nachdem die Cookie-ID gefunden wurde, alle folgenden Cookies (einschließlich des abschließenden NULL-Cookies) eine Position nach oben verrückt werden.

move_cookiejar() ermöglicht es, den kompletten Jar an eine neue Speicherstelle zu kopieren (der zweite Parameter gibt dabei die evtl. neue Größe des Jars an), und *cookie_size()* liefert die Größe des Jars, d.h. die Anzahl der maximal in ihn hineinpassenden Cookies. Zu guter Letzt kann der Inhalt des gesamten Cookie Jars noch mittels *print_cookie()* auf die Standardausgabeeinheit ausgegeben werden.

```
typedef struct
{
   char  cookie_id[4];  /* Cookie-Kennung -
                           4 ASCII-Zeichen  */
   long  cookie_value;  /* Wert des Cookies */
} COOKIE;
```

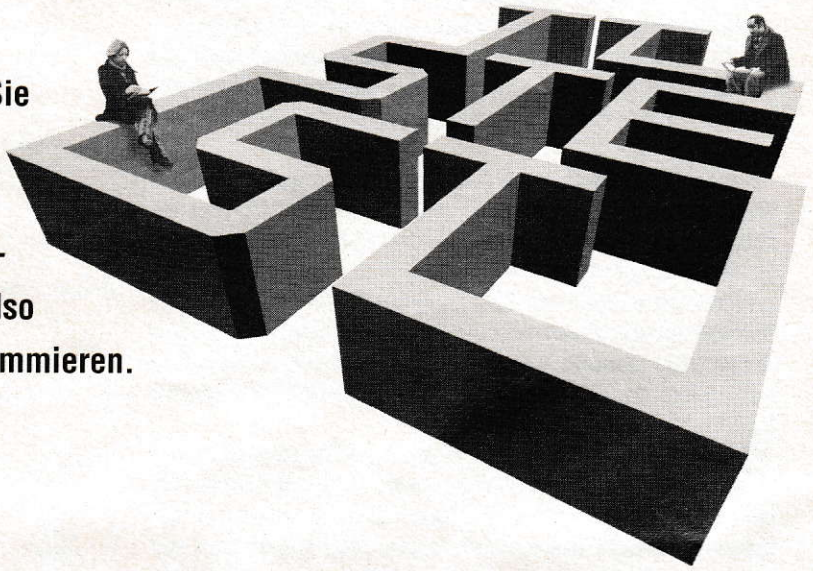
Bild 1: Die Cookie-Struktur in C

ID	Bedeutung
_CPU	Prozessortyp: Die Werte 0, 10, 20, 30 und 40 stehen für die Prozessoren 68000, 68010, 68020, 68030 und 68040.
_FPU	FPU-Typ: das obere Wort gibt Auskunft über das Vorhandensein eines Floating-Point-Koprozessors und über die Unterstützungsart des Betriebssystems bei Fließkommazahlenrechnung: Bit 0: gesetzt wenn FPU-Zusatzkarte (z.B. SFP 004 von ATARI) vorhanden ist; 68881 als Peripheriebaustein (ST und STE) Bit 1 + 2: Koprozessortyp (68881 oder 68882): 0: weder noch 1: 68881 oder 68882 (Typ unbekannt) 2: 68881 3: 68882 Bit 3: gesetzt wenn 68040
_FRB	Fast-RAM-Buffer (TT): Zeiger auf einen 64-KByte-Puffer im ST-RAM, der als Zwischenspeicher beim normalen ACSI-DMA-Transfer dient.
_MCH	Maschinentyp; das obere Wort bezeichnet die Rechnerfamilie 0: ST (520ST, 1040ST oder Mega ST) 1: STE (1040 STE und Mega STE) 2: TT
_SND	Soundhardware: Bit 0 gesetzt, wenn 'normaler' Soundchip (GI/Yamaha) vorhanden Bit 1 gesetzt, wenn Stereo-DMA-Chip vorhanden (STE und TT)
_SWI	DIP-Switch: Werte des internen DIP-Schalters, sofern vorhanden
_VDO	Video-Hardware: Das obere Wort beschreibt die Art der Video-Hardware: 0: ST 1: STE 2: TT
	Außer den bisher aufgeführten Cookies, die ab TOS1.6 automatisch installiert werden, verwendet ATARI folgende weitere Cookies:
_FDC	Informationen über Floppy-Controller. Oberstes Byte erteilt Auskunft über die maximale Schreibdichte: 0: normal (720 KB) 1: High-Density (1,44 MB) 2: Extra High Density (2,88 MB) Die unteren drei Bytes sind eine Herstellerkennung.
_FLK	GEMDOS verfügt über FILE-LOCKING-Erweiterungen; Wert ist Versionsnummer der Erweiterung
_INF	STEFIX (Patch-Programm für TOS 1.06) ist installiert
_NET	GEMDOS-Netzwerkerweiterung; Wert ist ein Zeiger auf die Herstellerkennung und -versionsnummer
_OOL	POOLFIX3 (Patch-Programm für GEMDOS 0.15) ist installiert
_SLM	Diablo-Treiber für SLM-Laserdrucker ist installiert

Tabelle 1: Die System-Cookies

Unterschiede in Rechner- & TOS-Versionen

In diesem Artikel erfahren Sie, was Sie beachten müssen, damit Ihre Programme auf allen Rechnertypen (ST, STE und TT) und unter allen Betriebssystemversionen lauffähig sind; Sie also 'sauber' auf ATARI-Rechnern programmieren.



Jürgen Haage

Selbst ein Anwender, der die ST-Entwicklung seit 1985 mitverfolgt hat, wird die Änderungen und Neuerungen, die das ATARI-Betriebssystem erfahren hat, bis auf die größte Neuerung, das Desktop (STE 2.05, TT 3.01), nicht sehr umfangreich nennen.Auch das Desktop des ST wurde bereits durch Zusatzprogramme wie Neodesk oder das Shareware-Produkt Gemini in den bebotenen Möglichkeiten erweitert, die auch deshalb von vielen weiterhin verwendet werden. Die Software-Entwickler brachten auf allen Gebieten sehr interessante und brauchbare Produkte zustande. Doch eines war fast bei jeder Entwicklung mehr oder weniger stark zu beobachten: Alles wurde weniger auf Einheitlichkeit ausgelegt als auf Ablaufgeschwindigkeit. Am deutlichsten ist dies auch heute noch bei Editor- und Textverarbeitungsprogrammen zu spüren. Wer erinnert sich nicht an den sagenhaft schnellen Tempus-Editor und kurz danach an sein um ein Vielfaches an Scroll-Geschwindigkeit gesteigertes Upgrade? Doch wie konnten solche Geschwindigkeiten erreicht werden? Waren doch die Urvorbilder wie 1ST_WORD wahre Krücken in puncto Text-Scrolling.

Warum die Betriebssystemfunktionen nutzen, wenn man es selbst viel besser nachprogrammieren kann? So dachten und handelten viele Programmentwickler damals und einige auch noch heute. Sehr zum Ärgernis der Hersteller von Hardware-Erweiterungen wie beispielsweise von Grafikkarten und Turboboards, die die Inkompatibilitäten, zum Teil auch des Betriebssystemes selbst, am deutlichsten zu spüren bekamen.

Das Zauberwort ist LINEA

Es handelt sich dabei um eine Sammlung von hardwarespezifischen Grafikfunktionen, auf die die VDI-Betriebssystemfunktionen zurückgreifen. Direkt nach der offiziellen Dokumentation von ATARI wurden sehr viele Programme dahingehend umgestellt oder neuprogrammiert. Heute weiß man, daß es besser gewesen wäre, wenn ATARI stattdessen offizielle Programmierrichtlinien veröffentlicht hätte, was übrigens bis heute nicht geschehen ist. Mit der Einführung der ATARI-TT-Geräte verschwanden auch die LINEA-Funktionen. Es handelte sich eben nicht, wie beim VDI-Gerätetreiber, um eine geräteunabhängige Funktionssammlung. Auch ATARI hat den Fehler eingestanden und rät nun jedem Software-Entwicklerm tunlichst die Finger von den hardwarespezifischen Betriebssystemroutinen zu lassen. Nachzulesen ist dies in der allerneusten Ausgabe des ATARI Profibuches für den ST-STE-TT vom Sybex Verlag. Dort sind nun auch zum ersten Mal die XBIOS-Funktionen, die die Video-Hardware betreffen, als nur auf die Rechnergrundkonfiguration (ohne Grafikzusatz-Hardware) bezogene Funktionen deklariert. Für die Programmierer bedeutet dies, daß sie zwar wissen dürfen, was die Funktionen bewirken, aber der Einsatz in eigenen Programmen eine Inkompatibilitätgarantie auf ATARI-Computern mit geänderten Grafiksystemen darstellt.

Weitere Schwierigkeiten bereitet der Motorola-68030-Prozessor im TT oder auf Turboboard für Programme, die einen eigenen Exceptionhandler installie-

ren und die Prozessoränderung nicht beachten. Die im folgenden aufgeführten Punkte sollten unbedingt vor der Entwicklung systemunabhängiger Software bedacht werden. Die danach beschriebenen Problemlösungen bieten wir Ihnen als Alternative an.

- Hat man vor, residente Programme zu schreiben, die einzelne Betriebssystemfunktionen abfangen oder die Funktionsliste erweitern, sollte unbedingt der eingesetzte Prozessortyp abgefragt werden. Bei Mißachtung erhält man sonst fehlerhafte Parameter vom Stack- und was viel schlimmer ist, durch fehlerhafte Stack-Korrektur einen Bombenhagel.
- Wird der Floating-Point-Coprozessor angesprochen, sollte von dessen Existenz natürlich nicht allgemein ausgegangen werden. Prüfen Sie sein Vorhandensein vor seinem Einsatz nach!
- Gehen Sie nie von einer festen Bildschirmauflösung und Farbanzahl aus. Bereits beim Programmstart erhält man alle nötigen Informationen, um sich eindeutig auf die vorgefundene Umgebung einzustellen.
- Wenn Sie in GFA-BASIC programmieren sollten keine Befehle zum Löschen oder zum Retten und Restaurieren des Bildschirms benutzt werden. Es handelt sich hierbei um starre Befehle, die sich nicht auf das jeweils eingesetzte Grafiksystem einstellen können.
- Sollten Sie in Ihren Programmen die Adresse des Bildschirmspeichers ermitteln, überdenken Sie bitte, wozu Sie diese benötigen. Auf erweiterten Grafiksystemen ist der Speicherbereich meist sehr eng begrenzt und kann nicht, wie auf der normalen Grundkonfiguration, irgendwo im Speicher liegen. Des weiteren läßt sich die Art der Bitmap-Aufteilung nicht genau bestimmen. Ein aktuelles und gutes Beispiel, wie man es nicht machen sollte, ist SIGNUM! 3. Das Programm funktioniert zwar in Farbe auf einem normalen TT, aber bei Grafikkarten mit linear angeordnetem Bitmap-Speicher versagt es seinen Dienst, da alle Bildschirmausgaben nicht über das Betriebssystem stattfinden.
- Benutzen Sie nie die Funktion *Setscreen (XBIOS 5)* zum Umschalten der Bildschirmanzeige. Farbgrafikkarten haben meist einen eigenen Grafikspeicher, so daß auf solchen Systemen kompliziert aus dem ST-Speicher in den Grafikspeicher kopiert werden muß. Oft als ist bei hochauflösender Farbdarstellung die Speicheranforderung so hoch, daß das zuvor angezeigte Bild dadurch verlorengeht.
- Beim Anzeigen von Rastergrafiken, egal ob in Farbe oder Schwarzweiß, sollte dies immer mit den VDI-Funktionen *vro_cpyfm (VDI 109)* oder *vrt_cpyfm (VDI 121)* geschehen. Des weiteren ist es wichtig, die darzustellende Rastergrafik durch die Funktion *vr_trnfm (VDI 110)* in das gerätespezifische Format zu transformieren, auch wenn es sich nur um eine Monochromgrafik handelt.
- Auch die LINEA-Grafikfunktionen sind verboten. Diese sind nur auf ST-Computern bis Betriebssystem 2.05 vorhanden und speziell auf die Video-Hardware ausgelegt.
- Rufen Sie bei Bildschirmausgaben nie die VDI-Alphatext-funktionen auf, da bei deren Initialisierung der Bildschirm komplett gelöscht wird.
- Vermeiden Sie die direkte Ausgabe, auch mit VDI-Betriebssystemfunktionen, in Bildschirmbereiche, die nicht durch ein AES-Fenster begrenzt sind.

```
...

Long stack,*cookie;
Long bigproz = 0L;

...

sack = Super(0L); /* I.d.Supervisormodus schalten */
cookie = *((Long **)0x5A0L); /*Cookie-Zeiger holen */
Super((void *)sack); /* Wieder in den User-Modus */

if(cookie != 0L) /* ist ein Cookie angelegt */
  {
  do
    {
    if((cookie[0] == '_CPU') && (cookie[1] > 0L))
      {
      bigproz = cookie[1]; /* Prozessor > 68000 */
      break;
      }
    cookie = &(cookie[2]); /* nächster Cookie */
    } while( cookie[ -2]); /* war Null-Cookie? */
  }

...
```

Listing 1: Prozessortypermittlung an Hand des Cookie-Jar-Eintrags

```
;—— Feststellen, ob 68010/20/30/40 installiert ——
;
  pea     illegaltst(pc)
  move.w  #4,-(sp) ; Illegal Vector
  move.w  #5,-(sp) ; Setexc
  trap    #BIOS
  addq.l  #8,sp
  moveq   #-1,bigproz ; Ist da
  move    CCR,d1      ; 68000 -> Illegal Instr. sonst
                        ohne Effekt
  move.l  d0,-(sp) ; alten Vektor wieder einsetzen
  move.w  #4,-(sp) ; Illegal Vector
  move.w  #5,-(sp) ; Setexc
  trap    #BIOS
  addq.l  #8,sp
  bra     weiter

illegaltst:
    clr.w   bigproz   ; Illegal -> 68000 im System
    addq.l  #2,2(sp)  ; PC Auf nächsten Befehl setzen
    rte
```

Listing 2: Prozessortypermittlung bei älteren TOS-Versionen

```
; Exception-Routine:

            movea.l a7,a0
            tst.w   bigproz
            beq.b   68000    ; normaler 68000

            addq.l  #8,a0    ; falls 68010 -> 68XXX
                               Stackframe korrigieren
            bra.b   weiter
    68000:  addq.l  #6,a0
            bra.b   weiter

weiter:                      ; jetzt können die
                               Parameter korrekt vom
                             ; Stapel (in A0) geholt
                               werden!

...
```

Listing 3: Beispiel zur Stackframe-Behandlung bei verschiedenen Prozessortypen

- Fenster sollten zwingend mit AES-Funktionen geöffnet werden und nicht starr auf dem Bildschirm liegen, sondern verschiebbar sein.
- Bei langeandauernden Ein-/Ausgaben darf deren Dauer nie mit einem normalen Dialogfenster [*form_alert()*] dokumentiert werden, da hierdurch die Bearbeitung anderer Programme oder Accessories unterbunden wird (siehe auch 'Multitasking' später in diesem Artikel)
- Eine Veränderung der Farbeinstellung sollte vor der Beendigung des Programmes wieder rückgängig gemacht werden.

Prozessortyp?

Seit TOS 1.06 existieren die Cookie-Jar-Einträge. Dabei handelt es sich um eine Systemvariable, die auf eine Tabelle mit Kennungen und Werten zeigt. Jeder Tabelleneintrag ist ein Paar von Langwörtern - ein Langwort für die eindeutige Kennung des Cookies *nd* eines für den zu übermittelnden Wert.

Will man den Prozessortyp ermitteln, muß man die Cookie-Jar-Systemvariable auslesen und die Tabelle durchsuchen, bis man auf das Ende trifft, das durch eine Null gekennzeichnet ist. Der Wert der Null-Kennzeichnung ist die in der Cookie-Tabelle erfaßbare Anzahl an Einträgen. Ein Bespiel dazu finden Sie in Listing 1. Zur Ermittlung des Prozessortyp bei einer älteren TOS-Version sehen Sie eine nicht ganz so einfache Methode in Listing 2.

Zur Ermittlung wird zuerst eine Exception-Routine *illegaltst* installiert, die bei einer illegalen Instruction aufgerufen werden soll. Danach wird der Befehl *CCR* ausgeführt, der erst ab dem 68010 im Befehlssatz enthalten ist. Startet man das Programm auf inem System mit einem normalen 68000-Prozessor, wird eine Illegal-Instruction-Exception (Ausnahmebehandlung mittels Interrupt; siehe Artikel 'Timer und Interrups' in diesem Sonderheft) ausgelöst, die zuvor installierte Routine angesprungen und die Variable *bigproz* gelöscht.

Was hat man nun davon, daß man den Prozessortyp kennt? Bei manchen Programmierproblemen ist es wichtig, den Prozessor genau zu kennen. Zum Beispiel, wenn ein eigener Exception-Handler installiert werden soll, da hierbei auf den um 2 Byte größeren Stackframe der Prozessoren ab 68010 reagiert werden muß. (siehe Listing 3)

Video-Hardware

Die allergrößte Schwierigkeit war und ist die Beachtung und entsprechende Reaktion auf verschiedene Video-Hardware. Mittlerweile gibt es ein sehr reichhaltiges Angebot an Grafikerweiterungen und Farbgrafikkarten, die alle über eigene Treiber-Software verfügen, die speziell an die Hardware-Eigenschaften angepaßt ist.

Was immer noch sehr oft falsch gemacht wird, ist die Ermittlung der Bildschirmauflösung. Zuverlässig kann dies nur mit den Funktionen *open_screenworkstation (VDI 1)* oder mit *open_virtualscreenworkstation (VDI 100)* gemacht werden. Diese Funktionen müssen llerdings immer aufgerufen werden, wenn Programme die Grafikfunktionen des VDI-Treibers nutzen wollen. Auch bei BASIC-Programmen muß dies geschehen. Die Rückgabewerte in GFA-BASIC können aus dem vorgegebenen Array *WORK_OUT* ausgelesen werden. Listing 4 zeigt dies in C am Beipiel einer *open_vwork()*-Routine. Die Rückgabewerte, die nach dem Aufruf im work_out-Array stehen, können Sie der Tabelle 1 entnehmen.

Da diese Informationen noch lange nicht ausreichend sind, beispielsweise fehlt die Information zur Farbtiefe (Bitplanes), existiert die Funktion *vq_extnd (VDI 102)*, mit der auch die erweiterte Parameterliste ermittelt werden kann (siehe Tabelle 2)

GFA-BASIC

In GFA-BASIC-Programmen lassen sich drei Fehler sehr häufig beobachten:

Der Befehl BITBLT ist in der GFA-BASIC-Dokumentation leider sehr dürftig erklärt. Will man ihn benutzen, müssen unzählige Parameter in verschiedene Arrays eingetragen werden. Einfacher hat man es mit den Befehlen SGET und SPUT. Mit diesen beiden Befehlen ist es sehr einfach möglich, den gesamten Bildschirm in einem String abzuspeichern und bei Bedarf wieder anzuzeigen. Doch leider steht nirgens im Handbuch, daß sich diese Befehle nur auf die normalen ST-Grafikmodi beschränken, da ein BASIC-String nur maximal 32768 Byte aufnehmen kann, was

der monochromen Maximal-Pixel-Auflösung von 640 * 400 entspricht.Wird ein Großbildschirm verwendet, bei dem der Bildschirmspeicher größer als 32 KByte angelegt ist, stürzt das Programm garantiert an dieser Stelle ab.

Ein weiteres Problem stellt die Verwendung des Befehls CLS dar, mit dem der gesamte Bildschirminhalt gelöscht werden kann. Sie sollten in Ihren eigenen Programmen immer damit rechnen, daß weitere Programme oder Accessories ebenfalls Bildschirmausgaben machen und diese dann durch das Fehlverhalten Ihres Programmes übermalt werden. Öffnen Sie deshalb für jede Ausgabe ein AES-Fenster. Nur so ist gewährleistet, daß andere Prorgamme „parallel" zu Ihrem auf dem Bildschirm ausgeben können.Auch der Aufruf der VDI-Funktion *v_enter_cur (VDI 5, ESC 3)* verursacht ein Löschen des gesamten Bildschirminhaltes und sollte deshalb nie benutzt werden.

Warnungen

Vermeiden Sie generell den Einsatz der XBIOS-Funktionen zur Ermittlung oder Beeinflussung der Video-Hardware! Im einzelnen sind dies die Funktionen:

EgetPalette	(XBIOS 85)
EgetShift	(XBIOS 81)
EsetBank	(XBIOS 82)
EsetColor	(XBIOS 83)
EsetGray	(XBIOS 86)
EsetPalette	(XBIOS 84)
EsetShift	(XBIOS 80)
Esetsmear	(XBIOS 87)
Getrez	(XBIOS 4)
Logbase	(XBIOS 3)
Physbase	(XBIOS 2)
Setcolor	(XBIOS 7)
Setpalette	(XBIOS 6)
Setscreen	(XBIOS 33)

Multitasking

Bereits seit einiger Zeit existiert MultiGEM, ein GEM-Multitasking-System, das es ermöglicht, mehrere Programme nicht nur gleichzeitig im Speicher zu halten, sondern auch „parallel" ablaufen zu lassen. Damit alles einwandfrei funktionieren kann, müssen sich die Programme strikt an die GEM-Konventionen halten. Einige Programmierer tun dies, aus Programmierzeitgründen nicht andere aus Geschwindigkeitsgründen. Das Betriebssystem ist nun mal nicht das schnellste, aber dafür geräteunabhängig (Grafiktreiber usw.). Damit mehrere Programme gleichzeitig ablaufen können, muß jedes Programm bei Text- oder Grafikausgaben ein AES-Fenster öffnen. Denn nur so ist gewährleistet, daß jedes Programm in einen speziell angeforderten Bildschirmbereich ausgibt. Geschieht dies nicht, wie z.B. bei vielen Zeichenprogrammen zu beobachten, ist es auch nicht möglich, diese Programme parallel ablaufen zu lassen.

Ein weiterer zu beachtender Punkt ist die Gewährleistung, daß mehrere Programme gleichzeitig ablaufen können. Dies ist dann nicht der Fall, wenn eine Dialogbox auf dem Bildschirm angezeigt wird, wenn man sich mit der Maus innerhalb der Menüleiste befinden, und wenn Programme (z.B. Mandelbrotprogramme) die ganze CPU-Zeit an sich ziehen. Ersteres ist sehr häufig bei Programmen zu beobachten, die eine Druckerausgabe erlauben und währendessen durch einen Dialog darauf hinweisen, daß jetzt nicht mehr weitergearbeitet werden kann.

Nach soviel Hinweisen, Geboten und Verboten sollten Sie jetzt einmal Ihre eigenen Programme durchsuchen. Sie werden sicher die ein oder andere fehlerhafte Stelle finden.

```
int   work_in[12],
      work_out[57];

int   handle,
      phys_handle;

int   gl_hchar,
      gl_wchar,
      gl_hbox,
      gl_wbox;

int   gl_apid;

int   xres,yres;

/************************************************/
/* boolean open_vwork(void);                    */
/*                                              */
/* Workstation öffnen ...                       */
/*                                              */
/* Eingabe:   Nichts                            */
/*                                              */
/* Rückgabe:  TRUE   falls das VDI initialisiert */
/*                   werden konnte              */
/*            FALSE  sonst                      */
/************************************************/

boolean open_vwork(void)
{
register int i;

if((gl_apid = appl_init( )) != -1)
      {
      for( i = 1; i < 10; work_in[i++] = 1);
      /* Koordinatenangaben in Rasterkoordinaten */
      work_in[10] = 2;
      /* das AES-Handle für VDI-Aufrufe ermitteln */
      phys_handle = graf_handle(&gl_wchar, &gl_hchar,
                                &gl_wbox, &gl_hbox);
      /* der Funktion muß ein bereits existierendes
          Handle übergeben werden */
      work_in[0]  = handle = phys_handle;
      /* Funktionsaufruf */
      v_opnvwk( work_in, &handle, work_out);
      /* Werte werden im work_out-Array abgelegt */

      xres = work_out[0]; /* Maximalen Pixelwerte */
      yres = work_out[1];

      if(handle)
            return (TRUE);
      }

return (FALSE);
}
```

Listing 4: Beispiel zum work_out- und work_in-Feld

Literatur:
ATARI Profibuch ST-STE-TT,
Jankowski/Rabich/Reschke, SYBEX-Verlag

WORK_OUT(0) -	horizontale Pixel-Anzahl
WORK_OUT(1) -	vertikale Pixel-Anzahl
WORK_OUT(2) -	Gerätekoordinaten-Flag
	0: Bild kann genau skaliert werden
	1: Bild kann nicht genau skaliert werden
WORK_OUT(3) -	Breite eines Pixels in µm (mm/1000)
WORK_OUT(4) -	Höhe eines Pixels in µm (mm/1000)
WORK_OUT(5) -	Anzahl der Schriftzeichenhöhen
	0: beliebig veränderbar
WORK_OUT(6) -	Anzahl der Linientypen
WORK_OUT(7) -	Anzahl der Linienbreiten
	0: beliebig veränderbar
WORK_OUT(8) -	Anzahl der Marker-Typen
WORK_OUT(9) -	Anzahl der Marker-Größen
	0: beliebig veränderbar
WORK_OUT(10) -	Anzahl der verfügbaren Zeichensätze
WORK_OUT(11) -	Anzahl der verfügbaren Füllmuster
WORK_OUT(12) -	Anzahl der Schraffuren
WORK_OUT(13) -	Anzahl der vordefinierten Farben
WORK_OUT(14) -	Anzahl der Grafikgrundfunktionen
WORK_OUT(15) bis	
WORK_OUT(24) -	Liste der unterstützten Grafikgrundfunktionen. Das Ende ist durch -1 gekennzeichnet. GEM-VDI unterstützt folgende 10 Funktionen:
	1: Balken
	2: Bogen
	3: Kreisausschnitt
	4: Kreis
	5: Ellipse
	6: elliptische Bogen
	7: Ellipsensegment
	8: Rechteck mit abgerund. Ecken
	9: ausgefülltes, abgerund. Rechteck
	10: justierter Grafiktext
WORK_OUT(25) bis	
WORK_OUT(34) -	Liste möglicher Attribute für die Grafikgrundfunktionen:
	0: Linie
	1: Marker
	2: Text
	3: ausgefüllter Bereich
	4: kein Attribut
WORK_OUT(35) -	Farbdarstellungs-Flag
	0: nicht verfügbar
	1: verfügbar
WORK_OUT(36) -	Textrotations-Flag
	0: nicht verfügbar
	1: verfügbar
WORK_OUT(37) -	Flächenfüllung
	0: nicht verfügbar
	1: verfügbar

→

Tab. 1: Inhalt des work_out-Arrays auf Aufruf der Funktion OPEN WORKSTATION (VDI 1) oder OPEN VIRTUAL SCREEN WORKSTATION (VDI 100)

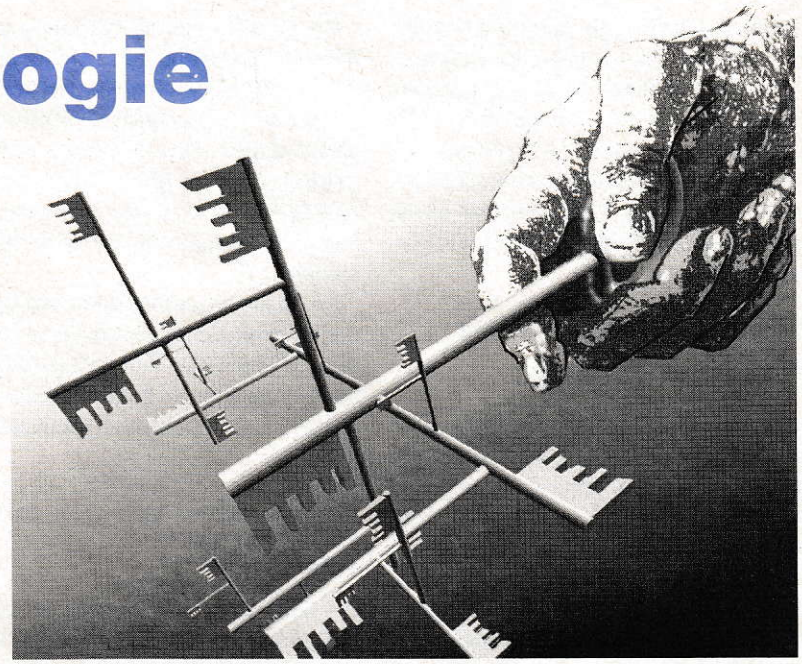
WORK_OUT(38) -	CELLARRAY-Flag
	0: nicht verfügbar
	1: verfügbar
WORK_OUT(39) -	Anzahl der verfügbaren Farben
	0: mehr als 32768
WORK_OUT(40) -	Kennzeichnung für Grafik-Cursor-Kontrolle
	0: keine
	1: nur Tastatur
	2: Tastatur und anderes Gerät (Maus)
WORK_OUT(41) -	Eingabegerät für variierende Eingaben
	0: keine
	1: Tastatur
	2: anderes Gerät
WORK_OUT(42) -	Auswahltasten
	0: keine
	1: Funktionstasten auf der Tastatur
	2: anderes Tastenfeld
WORK_OUT(43) -	alphanumerische Eingabe (String)
	0: keine
	1: Tastatur
WORK_OUT(44) -	Ein-/Ausgabegerät-Typ
	0: nur Ausgabe
	1: nur Eingabe
	2: Ein-/Ausgabe
	3: reserviert
	4: Metafile-Ausgabe
WORK_OUT(45) -	geringste Zeichenbreite
WORK_OUT(46) -	geringste Zeichenhöhe (Abstand zwischen Baseline und Topline)
WORK_OUT(47) -	größte Zeichenbreite
WORK_OUT(48) -	größte Zeichenhöhe
WORK_OUT(49) -	geringste Linienbreite
WORK_OUT(50) -	immer 0
WORK_OUT(51) -	größte Linienbreite
WORK_OUT(52) -	immer 0
WORK_OUT(53) -	geringste Marker-Breite
WORK_OUT(54) -	geringste Marker-Höhe
WORK_OUT(55) -	größte Marker-Breite
WORK_OUT(56) -	größte Marker-Höhe

Tab. 1: Inhalt des work_out-Arrays auf Aufruf der Funktion OPEN WORKSTATION (VDI 1) oder OPEN VIRTUAL SCREEN WORKSTATION (VDI 100)

WORK_OUT(0) -	Bildschirmtyp
	0: kein Bildschirm
	1: getrennter Alpha- und Grafik-Kontroller und getrennte Bildschirme
	2: getrennter Alpha- und Grafik-Kontroller mit gemeinsamem Bildschirm
	3: gemeinsamer Alpha- und Grafik-Kontroller mit getrenntem Bildschirm
	4: gemeinsamer Alpha- und Grafik-Kontroller mit gemeinsamem Bildschirm
WORK_OUT(1) -	Anzahl der verfügbaren Hintergrundfarben
WORK_OUT(2) -	Bit-Vektor d.verfügbaren Texteffekte
WORK_OUT(3) -	Flag für Vergrößerungsraster
	0: Vergrößern möglich
	1: Vergrößern nicht möglich
WORK_OUT(4) -	Anzahl der Farbebenen
WORK_OUT(5) -	„Lookup-table"-Unterstützung
	0: nicht verfügbar
	1: verfügbar
WORK_OUT(6) -	Anzahl der 16*16 Pixel-Raster-Operationen pro Sekunde
WORK_OUT(7) -	Contour-Fill-Verfügbarkeit
	0: nicht verfügbar
	1: verfügbar
WORK_OUT(8) -	Textrotation
	0: nicht möglich
	1: 90°-Drehung
	2: beliebig
WORK_OUT(9) -	Anzahl der Schreibmodi
WORK_OUT(10) -	höchster Grad der Eingabemodi
	0: keiner
	1: request
	2: sample
WORK_OUT(11) -	Textausrichtungsverfügbarkeit
	0: nicht verfügbar
	1: verfügbar
WORK_OUT(12) -	Farbstiftwechsel am Ausgabegerät
	0: nicht möglich
	1: möglich
WORK_OUT(13) -	Farbbandwechselmöglichkeit
	0: nicht verfügbar
	1: farbige Zeilen
	2: farbige Zeilen und Rechtecke
WORK_OUT(14) -	maximale Anzahl von Koordinatenpaaren für Polyline, Polymarker und Filled-Area
	-1: unbegrenzt
WORK_OUT(15) -	maximale Größe des INTIN-Arrays
	-1: unbegrenzt
WORK_OUT(16) -	Anzahl der Maustasten
WORK_OUT(17) -	Linientypen für breite Linien
	0: nicht möglich
	1: möglich
WORK_OUT(18) -	Schreibmodi für breite Linien
	0: nicht verfügbar
	1: verfügbar
WORK_OUT(19) bis WORK_OUT(44) -	reserviert, enthält 0 als Ausgabewert
WORK_OUT(45) -	obere linke x-Koordinate des Clipping Rechtecks
WORK_OUT(46) -	obere linke y-Koordinate des Clipping Rechtecks
WORK_OUT(47) -	untere rechte x-Koordinate des Clipping Rechtecks
WORK_OUT(48) -	untere rechte y-Koordinate des Clipping Rechtecks
WORK_OUT(49) bis WORK_OUT(56) -	reserviert, enthält 0 als Ausgabewert.

Tab. 2: Inhalt des work_out-Arrays nach Aufruf der Funktion EXTENDED INQUIRE FUNCTION (VDI 102)

Kryptologie



oder: Keiner versteht mich (hoffe ich)

Georg Scheibler

In diesem Artikel wollen wir uns mit dem Verschlüsseln von Daten beschäftigen. Das Interesse daran, ein bestimmtes Wissen vor seinen Mitmenschen geheimzuhalten, dürfte so alt sein wie die Menschheit. In der Frühgeschichte dürfte es sich dabei vor allem um das Wissen über gute Nahrungsplätze gehandelt haben.

Die Kniffe bei der Herstellung von Gebrauchsgegenständen und (Jagd-)Waffen wurden ebenfalls möglichst wenigen Leuten mitgeteilt. Solange das Wissen nur mündlich überliefert wurde, mußte man nur den Mund halten, um die Verbreitung zu verhindern (manchen Leuten soll dies ja sehr schwerfallen).

Wenn das Wissen schriftlich festgehalten wurde, mußte man dafür sorgen, daß die Schriftstücke nicht in die 'falschen Hände' fielen. Zusätzliche Sicherheit bietet ein Verschlüsseln des Textes, damit andere Leute den Inhalt auch dann nicht entschlüsseln können, wenn sie in den Besitz der Schriftstücke kommen.

Die Geheimhaltung interessiert vor allem die Militärs und die Geheimdienste, die eigentlich alles für geheim halten, was Ihnen in die Hände fällt, selbst wenn es jeder in irgendwelchen öffentlichen Bibliotheken nachlesen kann. Andererseits lassen sie sich einen Computer klauen, auf dessen Festplatte die Strategie für den Golfkrieg gespeichert war. Auf jeden Fall sind dies die beiden Gruppen, die besonders an guten Verschlüsselungsverfahren interessiert sind. Auf der anderen Seite haben Sie das Bedürfnis, die Chiffrierverfahren der Konkurrenz zu knacken.

In der Industrie wird die Bedeutung von Chiffrierverfahren mit der zunehmenden Verbreitung von Computern größer. Fast alle wichtigen Daten sind im Computer gespeichert. Da die Computer immer häufiger über öffentliche (Daten-)Netze erreichbar sind, müssen die Daten auch besser geschützt werden. Der erste Schritt ist sicher die Vergabe von Zugriffsrechten. Daß dieser Schutz nicht hundertprozentig ist, bekommt man gelegentlich mit, wenn mal wieder bekannt wird, das Hacker irgendwo in ein System eingedrungen sind. Die eigentliche Gefahr stellen dabei nicht so sehr die Leute dar, die nur zeigen wollen, daß sie in das System hineinkommen, sondern diejenigen, die an Betriebsgeheimnisse kommen wollen um diese an die Konkurrenz zu verkaufen bzw. selbst zu nutzen. Damit der Schaden möglichst gering bleibt, kann man wichtigere Daten zusätzlich verschlüsseln.

Manche versuchen auch an das Geld anderer Leute heranzukommen. Seit einigen Jahren gibt es Geldautomaten, bei denen man mit der EC-Karte Bargeld abheben kann. Zur Identifizierung des Eigentümers dient dabei die 'PIN'. Damit aber der Geldautomat die Richtigkeit der PIN überprüfen kann, muß diese auf der Karte vermerkt sein. Ein Dieb darf natürlich nicht mit jedem beliebigen Kartenleser die Karte auslesen können und so an die PIN herankommen, die Zahl muß also verschlüsselt werden. Ob das gewählte Chiffrierverfahren wirklich sicher genug ist (wie

Banken behaupten), oder ob die Skeptiker recht haben, kann ich nicht beurteilen, da ich nicht weiß, welches Verfahren eingesetzt wird. Auf jeden Fall sind die Daten nur so lange sicher, wie der Codierschlüssel unbekannt ist. Bei einer vierstelligen Zahl kann man sonst schnell alle 10000 Möglichkeiten in kurzer Zeit testen.

Was heißt eigentlich 'verschlüsseln'?

Unter Verschlüsseln versteht man eine eindeutige Substitution von Elementen durch andere Elemente. Dabei ist entscheidend, daß die Substitution in die andere Richtung ebenfalls eindeutig ist.

Diese Definition wirkt sehr abstrakt, aber ich habe sie so gewählt, damit sie alle Möglichkeiten der Verschlüsselung umfaßt. So kann man die Bezeichnung von Gegenständen mit Namen auch als eine Verschlüsselung interpretieren:

Wenn jemand sagt, er sehe einen Baum, dann stellt sich sein Gesprächspartner auch einen Baum vor. Es ist nur die Frage, ob es sich dabei um einen Laubbaum oder einen Nadelbaum handelt. Die Bezeichnung ist also nicht ganz eindeutig. Wir haben hier aber ein Bild (den Anblick des Baums) durch etwas anderes (das Wort 'Baum') ersetzt.

Einigen Lesern mag das Hinzurechnen von Sprache als Verschlüsselungsverfahren zu weit gehen. Sie sollten aber bedenken, wie wenig sie verstehen, wenn sie einen Text in einer anderen Sprache vorgelegt bekommen. Ein Beispiel, bei dem die Schrift regelrecht entschlüsselt werden mußte, sind die Hieroglyphen in den ägyptischen Pyramiden.

Hex	Prozent	Hex	Prozent
20 (Sp)	13,99	63 c	2,31
65 e	13,54	75 u	2,08
6E n	8,18	0D (CR)	1,95
69 i	6,26	0A (LF)	1,95
72 r	5,32	6D m	1,78
73 s	4,45	67 g	1,74
74 t	4,32	6F o	1,55
61 a	3,80	77 w	1,35
64 d	3,51	62 b	1,29
6C l	3,47	66 f	1,19
68 h	3,42	2E .	1,03

Bild 1: Häufigkeit der Buchstaben dieses Artikels

Wir wollen jetzt die Definition enger fassen. Wir betrachten nur noch Verfahren, die eingesetzt werden können, um Texte oder allgemein Dateien zu verschlüsseln. Das Ziel ist es, dafür zu sorgen, daß (möglichst) nur ausgewählte Personen in der Lage sind, den Sinn des Textes oder der Datei zu verstehen.

Die Auswahl des Verfahrens hängt von den Hilfsmitteln ab, die einem zur Verfügung stehen. Die Verschlüsselung eines Textes über ein Exklusiv-Oder-Verfahren ist ohne Computer recht zeitaufwendig.

Betrachten wir ein paar Verfahren

Wir wollen uns hier nur mit Verfahren beschäftigen, die für beliebige Texte geeignet sind und die von den Buchstaben bzw. deren ASCII-Wert ausgehen. Wir wollen nicht auf Verfahren eingehen, die auf Wörter zurückgehen. (z.B. durch Angabe von Seiten-, Zeilen- und Wort-Nummer in einem abgesprochenen Buch. Man muß dann nachsehen, welche Wörter an den entsprechenden Stellen stehen. Nur: Wie verschlüsselt man ein Wort, das in dem Buch nicht vorkommt?)

Wenn wir ein Verfahren verwenden oder ein neues Verfahren entwerfen wollen, müssen wir uns immer Gedanken machen, wie ein Außenstehender den verschlüsselten Text entschlüsseln kann. Wir müssen dabei voraussetzen, daß ihm das Prinzip des Verfahrens bekannt ist. Nach einigen falschen Versuchen wird er also irgendwann auf das richtige Verfahren tippen. Wenn das Verfahren gut ist, braucht er trotzdem nicht den richtigen Schlüssel zu finden bzw. er benötigt dafür - statistisch gesehen - viel zuviel Zeit. Es kann natürlich immer sein, daß jemand gleich beim ersten Versuch auf den richtigen Schlüssel tippt, selbst wenn die Wahrscheinlichkeit für sechs Richtige im Lotto viel größer sein sollte. Wir müssen uns daher überlegen, ob es wirklich keinen Ansatz gibt, mit dem der Schlüssel schnell ermittelt werden kann. Die Schwierigkeit besteht darin, daß wir für alle Ansätze, die uns eingefallen sind, zeigen können, daß der Aufwand zu groß wird. Wenn wir den entscheidenden Ansatz übersehen, wägen wir uns in der falschen Gewißheit, ein sicheres Verfahren zu haben. Man kann dann nur hoffen, daß uns jemand rechtzeitig auf den Denkfehler aufmerksam macht. Daraus können Artikelserien wie [1] bis [4] folgen.

Die vorgestellten Verfahren arbeiten mit den ASCII-Werten der Zeichen und liefern als Ergebnis wieder eine Folge von ASCII-Werten. Es dürfen dabei jeweils alle Werte von 0 bis 255 vorkommen. Wenn man nur normalen Text verschlüsseln will, kann man eventuell die erlaubten Zeichen einschränken. Besonders bei den Verfahren mit Exklusiv-Oder-Verknüpfung kann man dadurch die Sicherheit des Verfahrens erhöhen. Darauf werde ich später noch etwas ausführlicher eingehen.

Die Listings dienen nur der Illustration der Verfahren. Die Hauptprogramme enthalten daher keine Dateizugriffe. Die Verschlüsselungsroutinen können so verwendet werden, solange die Datei nicht länger als 32 KB ist (maximale String-Länge). Andernfalls müssen wir die Dateien in einem festen Speicherbereich, oder bei GFA-BASIC 3.x in einem Byte-Array, ablegen. Der Zugriff auf die Zeichen ist dann entsprechend anzupassen.

Die Listings 1 bis 4 sind sowohl für GFA-BASIC 2.x als auch für GFA-BASIC 3.x geeignet. Listing 5 ist nur für GFA-BASIC 3.x, da es in der Version 2.x noch keine Schiebebefehle gibt.

Monoalphabetische Verfahren

Betrachten wir zunächst einige Verfahren, die nur von dem aktuellen Zeichen abhängig sind. Ein bestimmtes Zeichen wird immer auf gleiche Zeichen abgebildet.

Ein recht einfaches Verfahren, das bereits Caesar verwendet hat, verschiebt die Buchstaben einfach um ein paar Stellen im Alphabet. Wir addieren zum ASCII-Wert also einfach eine konstante Zahl dazu. Wird die Zahl dabei größer als die größte erlaubte Zahl, wird die Anzahl der Zeichen subtrahiert. Wenn wir z.B. jeweils 3 addieren, wird aus dem 'a'($61) ein 'd'($64) und aus '³'($FE) wird [Pfeil hoch]($01).

Ein weiteres Verfahren, das mit Computern sehr einfach zu realisieren ist, ist die Exklusiv-Oder-Verknüpfung mit einer beliebigen Zahl, die kleiner als 256 ist. Wenn in Listing 1 die Zeile 22 ent-

fernt oder als Kommentar gekennzeichnet wird, haben wir ein Demoprogramm für dieses Verfahren. Wird bei diesem Programm als Schlüssel eine Zahl größer als 255 angegeben, werden nur die unteren 8 Bits der Zahl verwendet. Die Verfahren mit Exklusiv-Oder-Verknüpfung haben den Vorteil, daß die gleiche Prozedur zum Entschlüsseln verwendet werden kann. Man muß die Verschlüsselung nur ein zweites Mal durchführen.

Da es bei diesen beiden Verfahren nur 255 mögliche Schlüssel gibt, ist der Aufwand sehr klein, alle Kombinationen durchzuprobieren.

Bei dem letzten Verfahren werden die Buchstaben stärker durcheinandergewürfelt. Da ist es schon naheliegend, gleich eine beliebige zufällige Vertauschungstabelle zu erstellen. Beim Aufstellen dieser Tabelle muß man nur darauf achten, daß nicht zwei Zeichen dem selben Wert zugeordnet werden. Es gibt für die Vertauschungstabelle [A anzahl Elemente]! (! = Faktultät) Möglichkeiten. Bei 256 Zeichen sind dies $8{,}5*10^{508}$ Möglichkeiten. Selbst für einen schnellen Computer ist es unmöglich, sie alle durchzuprobieren. Haben wir jetzt also das sichere Verfahren?

Um es kurz zu machen: nein. Zunächst können wir meist eine Aussage darüber machen, um was es sich bei der verschlüsselten Datei handelt. In der Regel wird es sich um einen geschriebenen Text handeln. Bei einem normalen Text kommen vor allem die normalen kleinen Buchstaben sowie die Satzzeichen vor. Die Großbuchstaben sind relativ selten. Es bleiben dann nur noch ungefähr 10^{30} Möglichkeiten übrig.

Beim Entschlüsseln kann man viele dieser Möglichkeiten ausschließen, weil die Häufigkeit der einzelnen Buchstaben recht unterschiedlich ist. Wenn man die statische Häufigkeit der Buchstaben in einer bestimmten Sprache mit der Häufigkeit der verschiedenen Zeichen im Text vergleicht, kann man schon eine ganze Reihe Buchstaben richtig zuordnen. In Bild 1 ist der Anfang der Häufigkeitstabelle für diesen Text wiedergegeben. Die Tabelle zeigt, wie unterschiedlich oft die einzelnen Buchstaben vorkommen. Die Anzahl der Return-Zeichen ($0A und $0D) hängt natürlich von der Breite der Zeilen ab.

Weitere Einschränkungen sind möglich, wenn wir uns ansehen, welche Buchstaben häufig als Doppelbuchstaben vorkommen. Ein nächster Schritt betrachtet die Wahrscheinlichkeit für die Abfolge von verschieden Zeichen. z.B. ist die Folge 'sch' recht häufig, aber die Folge 'hcs' wird wohl kaum vorkommen.

Wenn man dann erst einige Buchstaben zugeordnet hat, kann man die weiteren Buchstaben nach und nach raten. Wie gut man die Buchstaben raten kann, sieht man an den Rätseln, bei denen die Buchstaben durch Zahlen ersetzt sind und als Einstieg nur ein kurzes Wort bekannt ist.

Wird die Vertauschungstabelle mit einem Pseudozufallszahlengenerator (PSZZG) erzeugt (wie z.B. im Listing 4), werden von den theoretisch möglichen Kombinationen nur einige genutzt, da die Zufallszahl von ihrem Vorgänger abhängig ist. Die Reihenfolge der Zahlen ist eindeutig festgelegt, mit dem Startwert legen wir nur fest, an welcher Stelle der Endlosschleife wir anfangen. Die Implementation im Listing 4 verwendet jeweils nur die unteren acht Bits der Zahl. Wir erhalten dadurch immerhin mehr als 256 verschiedene Kombinationen.

```
' Verschlüsseln mit xor
' von Georg Scheibler, 4920 Lemgo
' (c) 1992 MAXON Computer
a$="dies ist ein test"
TEXT 0,13,a$
s1%=&HE62FA952
@crypt(s1%,a$,*b$)
TEXT 0,30,b$
@crypt(s1%,b$,*c$)
TEXT 0,50,c$
PROCEDURE crypt(o1%,s$,d%)
  LOCAL d$,m%,h%,g%
  ' Konstanten für PSZZG
  ' zur schnelleren Berechnung
  m%=2^(23-1) !MSB
  h%=2^(9-1)  !zweites bit
  g%=m%+m%-1  !größte Zahl
  '
  l%=LEN(s$)
  d$=s$ !damit nicht zuviel Stringmüll entsteht
  FOR i%=1 TO l%
    ' "AND $FF" erfolgt automatisch
   MID$(d$,i%,1)=CHR$(ASC(MID$(s$,i%,1)) XOR o1%)
    o1%=@pszzg(o1%)
  NEXT i%
  *d%=d$
RETURN
' PSZZG mit schieben nach links
DEFFN pszzg(i%)=((i%+i%) AND g%)-((0<(i% AND m%))
XOR ((i% AND h%)>0))
```

Listing 1: Verschlüsselung mittels XOR

```
' Verschlüsseln durch Vertauschen der Buchstaben
' zusätzlich xor-Verschlüsselung in Kommentarzeilen
' von Georg Scheibler, 4920 Lemgo
' (c) 1992 MAXON Computer
a$="dies ist ein test"
DIM unused!(32000)
TEXT 0,13,a$
s1%=&HE62FA952
s2%=2345252
@crypt(s1%,s2%,TRUE,a$,*b$)
TEXT 0,30,b$
@crypt(s1%,s2%,FALSE,b$,*c$)
TEXT 0,50,c$
PROCEDURE crypt(key1%,key2%,crypt!,s$,d%)
  LOCAL d$,m%,h%,g%,z%
  ' Konstanten für PSZZG
  ' zur schnelleren Berechnung
  m%=2^(23-1) !MSB
  h%=2^(9-1)  !zweites Bit
  g%=m%+m%-1  !größte Zahl
  '
  l%=LEN(s$)
  d$=SPACE$(l%)
  i%=0
  max%=l%
  ARRAYFILL unused!(),TRUE
  WHILE i%<l%
    key1%=@pszzg(key1%,g%,m%,h%)
    ptr%=key1% MOD max%
    IF unused!(ptr%)
      unused!(ptr%)=FALSE
      ' key2%=@pszzg(key2%,g%,m%,h%)
      IF crypt!
         MID$(d$,ptr%+1,1)=MID$(s$,i%+1,1)
        ' z%=ASC(MID$(s$,i%+1,1)) XOR key2%
        ' MID$(d$,ptr%+1,1)=CHR$(z%)
      ELSE
        MID$(d$,i%+1,1)=MID$(s$,ptr%+1,1)
        ' z%=ASC(MID$(s$,ptr%+1,1)) XOR key2%
        ' MID$(d$,i%+1,1)=CHR$(z%)
      ENDIF
      INC i%
    ENDIF
  WEND
  *d%=d$
RETURN
DEFFN pszzg(i%,g%,m%,h%)=((i%+i%) AND g%)-((0<(i%
AND m%)) XOR ((i% AND h%)>0))
```

Listing 2: Verschlüsselung mittels Vertauschen der Buchstaben

Was ist ein Pseudozufallszahlen-generator (PSZZG) ?

Da zur Erzeugung von willkürlichen Zahlenfolgen häufig ein PSZZG verwendet wird, wollen wir etwas ausführlicher darauf eingehen. Da für einige Anwendungen (z.B. Monte-Carlo-Integration) Zufallszahlen benötigt werden, enthalten viele Hochsprachen eine Zufallsfunktion, mit der Zufallszahlen ermittelt werden können. Da die meisten Computer keinen echten Zufallszahlengenerator enthalten, werden die Zufallszahlen berechnet. Es sind also keine echten, sondern Pseudozufallszahlen. Charakteristisch für Pseudozufallszahlengeneratoren ist, daß das Ergebnis der letzten Berechnung als Argument für die nächste Zahl verwendet wird:

$x_{n+1} = f(x_n)$

In der Literatur findet man verschiedene Funktionsvorschriften für die Berechnung von Zufallszahlen. Wir wollen hier nur die Möglichkeit betrachten, die auf rückgekoppelten Schieberegistern beruht:

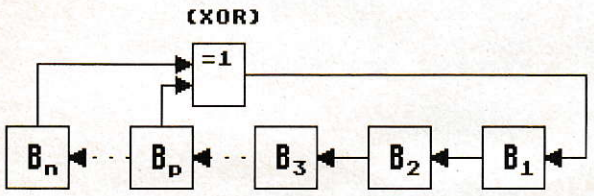
Wir verwenden (oder simulieren) ein Schieberegister mit n Bit. Der Inhalt kann als Integerzahl entsprechender Größe interpretiert werden. Um eine neue Zahl zu erzeugen, werden die Bits um eine Stelle verschoben. Mathematisch entspricht dies einer Multiplikation mit zwei oder einer Division durch zwei. Das Bit, welches am Ende herausfällt, wird mit ein oder zwei Bits aus dem Innern der Zahl exklusiv-oder verknüpft und in die freie Stelle am anderen Ende geschoben. Wenn die Größe des Schieberegisters und der (die) Abgriffpunkt(e) günstig gesetzt werden, werden alle Zahlen zwischen 0 und 2^n erzeugt. Bei (2^n-1) Aufrufen erhält man dann auch garantiert jeden Wert genau einmal geliefert. Bei der Simulation des Schieberegisters hat die Division den Vorteil, daß bei Integerzahlen das niedrigste Bit automatisch herausfällt. Bei der Multiplikation müssen wir das höchste Bit zusätzlich löschen. Dies kann man mit einem *AND* (2^n-1) durchführen. Die Multiplikation mit 2 können wir dafür aber durch die schnellere Addition ersetzen. Selbst in Assembler ist die Addition schneller als das Verschieben um ein Bit. Werden sowieso nur die unteren Bit der Zahl verwendet, müssen wir das höchste Bit nicht einmal löschen, da bei einem Überlauf nur das C-Flag gesetzt wird. Es wird keine Fehlermeldung ausgelöst, wie dies bei Hochsprachen der Fall ist. Ein inneres Bit wird durch ein *AND* 2^p geprüft. Ist das Ergebnis ungleich 0, war das Bit gesetzt. Die Zahl '2^p' sollte allerdings vorher einer Variablen zugewiesen werden, damit sie nicht jedesmal berechnet werden muß (Bei GFA-BASIC 3.x können wir auch den Befehl 'BTST' verwenden). Es ist zu beachten, ob die Bits ab 0 oder ab 1 gezählt werden. Im letzteren Fall muß p um Eins erniedrigt werden, es sei denn, die Zahl wird vor dem Test mit 2 multipliziert.

In der Tabelle in Bild 2 sind einige Werte für 'n' und 'p' aufgeführt. Den ersten Wert für 'p' habe ich von [3] übernommen, die weiteren durch Ausprobieren ermittelt (der Startwert des PSZZG muß das nächste Mal nach genau 2^n-1 Aufrufen als Ergebnis geliefert werden. Mir ist dabei aufgefallen, daß neben dem Wert p auch immer der Wert n-p möglich ist. Da dies bei allen geprüften Werten für n der Fall ist, dürfte es sich nicht mehr um einen Zufall handeln.

Die Bildungsvorschrift für die PSZZG auf der Basis von Schieberegistern bringt es mit sich, daß nach einer (beliebigen) Zahl nur zwei Zahlen möglich sind, weil n-1 Bits nur um eine Stelle verschoben wurden. Welche der beiden Zahlen gewählt wird, hängt von den inneren Abgriffen ab.

```
 1: ' Verschlüsseln durch Addieren der ASCII-Werte
 2: ' von Georg Scheibler, 4920 Lemgo
 3: ' (c) 1992 MAXON Computer
 4: a$="dies ist ein test"
 5: TEXT 0,13,a$
 6: s1%=45
 7: @crypt(s1%,a$,*b$)
 8: TEXT 0,30,b$
 9: @decrypt(s1%,b$,*c$)
10: TEXT 0,50,c$
11: PROCEDURE crypt(o1%,s$,d%)
12:   LOCAL d$
13:   l%=LEN(s$)
14:   d$=s$ !damit nicht zuviel Stringmüll entsteht
15:   o_1%=0
16:   FOR i%=1 TO l%
17:     o_0%=ASC(MID$(s$,i%,1))
18:     o_1%=(o_1%+o1%+o_0%) AND &HFF
19:     MID$(d$,i%,1)=CHR$(o_1%)
20:   NEXT i%
21:   *d%=d$
22: RETURN
23: PROCEDURE decrypt(o1%,s$,d%)
24:   LOCAL d$
25:   l%=LEN(s$)
26:   d$=s$ !damit nicht zuviel Stringmüll entsteht
27:   h1%=0
28:   FOR i%=1 TO l%
29:     o_1%=ASC(MID$(s$,i%,1))
30:     o_0%=(o_1%-h1%-o1%) AND &HFF
31:     MID$(d$,i%,1)=CHR$(o_0%)
32:     h1%=o_1%
33:   NEXT i%
34:   *d%=d$
35: RETURN
```

Listing 3: Verschlüsselung mittels Addieren der ASCII-Werte



Einige Werte für 'n' und 'p'
für ein Schieberegister als PSZZG

n	p	n	p
7	4,3,6	33	13
9	4,5	119	8
10	3,7	127	15
11	9,2	151	15
15	8,4,7,11	175	16
17	3,5,6,11,12,14	380	47
23	9,5,14,18	396	25
25	3,7,18,22	476	15
28	25,19,15,13,9,3	521	48
31	25,24,18,13,7,6	924	31

Bild 2: Werte für den PSZZG

Wird beim Verschlüsseln eine stärkere Unterscheidung der Zahlen gewünscht, kann der PSZZG mehrfach hintereinander augefrufen werden. Die Anzahl der Aufrufe wird als Dezimation bezeichnet. Wird jede Zahl verwendet, haben wir eine Dezimation von 1. Wird jede zweite Zahl genommen, haben wir eine Dezimation von 2. Wir müssen allerdings den PSZZG nicht x-mal aufrufen, um eine Dezimation von x zu bekommen. Wir können auch den PSZZG so schreiben, daß direkt die Dezimation x berechnet wird, sofern x kleiner als p ist. Die passenden Funktionen für GFA-BASIC 3.x sind in Listing 5 wiedergegeben. Durch die Schiebebefehle bleibt die Berechnung schnell. Stehen keine Schiebebefehle zur Verfügung, müssen wir stattdessen auf die Multiplikation mit bzw. die Division durch 2^x zurückgreifen. Die Masken, die im Listing aus -1 erzeugt werden, erzeugt man über (2^x-1). Wenn wir die Funktion nur für eine bestimmte Dezimation schreiben, können wir die Masken für die AND-Befehle vorgeben. Dies ist natürlich schneller als deren Erzeugung.

Das Verfahren mit dem (simulierten) Schieberegister hat den Vorteil, daß keine Rechenungenauigkeiten auftreten können. Benötigen wir für die Zufallszahl mehr Bits, als die Integervariablen zur Verfügung stellen, verwenden wir am besten ein Array für die Zahl. Die Zahl wird dann auf die Array-Elemete aufgeteilt und der Aufwand für die Rechnung natürlich größer. Eventuell kann man die Funktion in Assembler schreiben.

Andere PSZZG arbeiten über die Nachkommastellen von Fließkommazahlen. Eine mögliche Funktion ist *FRAC (FRAC(100000*X)+3*X)* [1]. Die Eigenschaft der so erzeugten Zufallszahlen hängt vom Startwert ab. Würden wir z.B. mit X = 0.5 anfangen, wird als Folgezahl wieder 0.5 geliefert. Dies ist der ungünstigste Fall. Wir sehen daran aber, daß die Startzahl viele Stellen hinter dem Komma haben sollte.

Bei der Verwendung von Fließkommazahlen müssen wir jedoch darauf achten, welches Zahlenformat die jeweilige Sprache bietet. So kann die gleiche Formel bei GFA-BASIC 3.x eine andere Zahlenfolge liefern als bei GFA-BASIC 2.x, weil bei GFA-BASIC 3.x mit einer größeren Mantisse gerechnet wird. Werden dann noch Funktionen dazu genommen, die sowieso nur näherungsweise berechnet werden können (z.B. trigometrische Funktionen), ist auch noch entscheidend, wie diese Funktion implementiert ist.

Diese Überlegungen gelten auch für die von der jeweiligen Sprache zur Verfügung gestellte Zufallsfunktion. Diese Funktionen sind sowieso nur brauchbar, wenn man einen definierten Startwert einstellen kann. Wechseln wir die Programmiersprache, müssen wir zudem noch damit rechnen, daß der PSZZG auf einer anderen Formel basiert.

Hauptsache gut gewickelt

Die Spartaner im 5 Jh. v. Chr. verwendeten ein Verfahren, bei dem die Buchstaben nicht geändert wurden, sondern deren Reihenfolge.

Sie wickelten einen Papierstreifen um einen Rundstab und schrieben den Text senkrecht auf den Stab. Wenn der Papierstreifen wieder abgewickelt wird, ist der Text nicht mehr zu lesen, weil die Buchstaben auseinandergerissen sind. Zum Entschlüsseln muß man den Papierstreifen wieder um einen Stab mit dem gleichen Durchmesser wickeln. Eine Umsetzung für den Computer sollte Ihnen nicht schwerfal-

```
' Verschlüsseln durch Substitution und Addieren
' von Georg Scheibler, 4920 Lemgo
' (c) 1992 MAXON Computer
DIM c%(256),d%(256)
a$="dies ist ein test"
TEXT 0,13,a$
z%=451
s1%=23
s2%=65
@crypt(z%,s1%,s2%,a$,*b$)
TEXT 0,30,b$
@decrypt(z%,s1%,s2%,b$,*c$)
TEXT 0,50,c$
PROCEDURE crypt(z%,key1%,key2%,s$,d%)
  LOCAL key_1%,key_2%,l%,d$
  IF z%<>c%(256)
    @zufall(z%)
  ENDIF
  l%=LEN(s$)
  d$=s$ !damit nicht zuviel Stringmüll entsteht
  key_1%=0
  key_2%=0
  FOR i%=1 TO l%
    key_0%=c%(ASC(MID$(s$,i%,1)))
    key_1%=c%((key_1%+key1%+key_0%) AND &HFF)
    key_2%=c%((key_2%+key2%+key_1%) AND &HFF)
    MID$(d$,i%,1)=CHR$(key_2%)
  NEXT i%
  *d%=d$
RETURN
PROCEDURE decrypt(z%,key1%,key2%,s$,d%)
  LOCAL key_1%,key_2%,h1%,h2%,l%,d$
  IF z%<>c%(256)
    @zufall(z%)
  ENDIF
  l%=LEN(s$)
  d$=s$ !damit nicht zuviel Stringmüll entsteht
  h1%=0
  h2%=0
  FOR i%=1 TO l%
    key_2%=(ASC(MID$(s$,i%,1)))
    key_1%=(d%(key_2%)-h2%-key2%) AND &HFF
    key_0%=(d%(key_1%)-h1%-key1%) AND &HFF
    MID$(d$,i%,1)=CHR$(d%(key_0%))
    h1%=key_1%
    h2%=key_2%
  NEXT i%
  *d%=d$
RETURN
PROCEDURE zufall(x%)
  LOCAL h1%,h2%,xx%
  ' Erzeugen einer "zufälligen"
  ' Verschlüsselungstabelle
  ' mit Pseudozufallzahlen
  '
  h1%=2^11  !maske ffr bit 1 (+ obergrenze)
  h2%=2^2  !maske ffr bit 2
  x%=x% MOD h1%
  IF x%=0
    INC x%
  ENDIF
  ARRAYFILL c%(),-1
  c%(256)=x% !Schlüssel merken
  cnt%=0
  WHILE cnt%<256 !Codiertabelle aufbauen
    xx%=x% AND &HFF
    IF c%(xx%)<0 !Wert noch nicht dagewesen
      c%(xx%)=cnt% !Codiertabelle
      d%(cnt%)=xx% !Decodiertabelle
      INC cnt%
    ENDIF
    ADD x%,x%
    IF (x% AND h1%)<>0 XOR (x% AND h2%)<>0
      INC x%
    ENDIF
    IF x%=>h1%
      SUB x%,h1%
    ENDIF
  WEND
RETURN
```

Listing 4: Verschlüsselung mittels Substitution und Addition

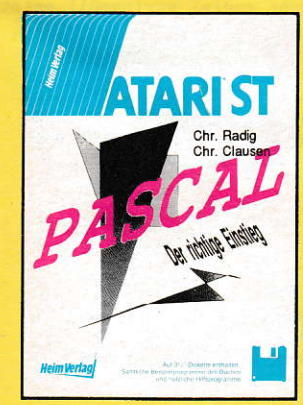
Die Buchpalette für

B-411
HARDCOVER
inklusive
Programmdiskette
ISBN 3-923250-55-X
DM 59,–

B-440
HARDCOVER
inklusive
Programmdiskette
ISBN 3-923250-82-7
DM 59,–

B-413
HARDCOVER
über 500 Seiten
inkl. Programmdisk.
ISBN 3-923250-60-6
DM 59,–

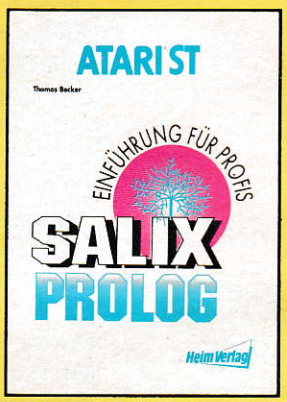
B-448
HARDCOVER
über 400 Seiten
ISBN 3-923250-xx-x
DM 49,–



B-439
HARDCOVER
inklusive
Programmdiskette
ISBN 3-923250-81-9
DM 59,–

B-444
HARDCOVER
inklusive
Programmdiskette
ISBN 3-923250-96-7
DM 59,–

B-447
HARDCOVER
über 390 Seiten
ISBN 3-923250-89-4
DM 49,–

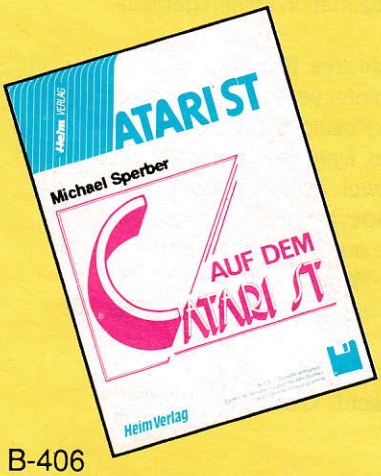


B-457
HARDCOVER
über 300 Seiten
ISBN 3-923250-04-9
DM 49,–

B-433
HARDCOVER
200 Seiten
ISBN 3-923250-75-4
DM 29,–

B-436
HARDCOVER
inklusive
Programmdiskette
ISBN 3-923250-77-0
DM 59,–

den Programmierer

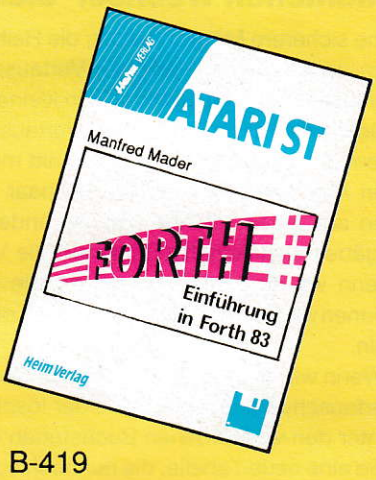


B-406
HARDCOVER
über 500 Seiten
inkl. Programmdisk.
ISBN 3-923250-45-2
DM 59,–

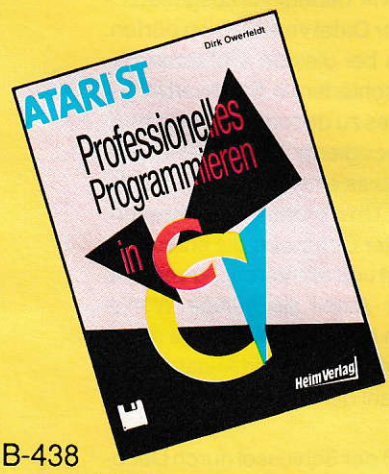
MODULA-2 ist die konsequente Weiterentwicklung von Pascal und eine der modernsten Programmiersprachen überhaupt. Bei Ihren ersten Schritten in MODULA-2 nimmt Sie dieser Kurs an die sichere Hand. Begriffe wie Datentypen (Byte, Integer, Pointer), Datenstrukturen (Verbunde, Felder, Listen), wiederholte Anweisungen – sprich Schleifen (For, Repeat, While, Loop-nein keine Endlosschleifen), Prozeduren, Prozedurvariablen, Module (lokale, Definitions-, Implementations-, Programm-Module) und Co-Routinen (für parallele Prozesse) werden Ihnen schon bald vertraut sein, wie Ihr tägliches Frühstück. Sie steigen gleich voll ins Programmieren ein. Die einzelnen Elemente von Modula-2 werden vor Ort jeweils an einem konkreten Beispiel erklärt – so, wie sie benötigt und verwendet werden.

Über den Inhalt:
- komplette Adreßverwaltung
- ein UPN-Rechner (nicht 2.3 sondern 23.)
- eine Grafikbibliothek
- dynamische Strings
- eigener Editor
- ein Infix-Posfix-Konverter
- im Finale ein UPN-Interpreter mit Schleifen, Variablen, Prozeduren und allen Funktionen der Grafikbibliothek.

B-446
HARDCOVER
inklusive
Programmdiskette
ISBN 3-923250-85-1
DM 59,–



B-419
HARDCOVER
über 530 Seiten
inkl. Programmdisk.
ISBN 3-923250-69-x
DM 54,–



B-438
HARDCOVER
über 400 Seiten
ISBN 3-923250-78-9

DM 59,–

B-435
HARDCOVER
inklusive
Programmdiskette
ISBN 3-923250-79-7
DM 59,–

Alle Preise
sind unverbindlich empfohlene Verkaufspreise

BESTELL - COUPON

Bitte senden Sie mir: _____

_____

zuzüglich Versandkosten DM 6,- (Ausland DM 10,-) unabhängig von der bestellten Stückzahl

❑ per Nachnahme ❑ Verrechnungsscheck liegt bei

Name, Vorname _____

Straße, Hausnr. _____

PLZ, Ort _____

Oder benutzen Sie die eingeheftete Bestellkarte

In Österreich:
Dipl.-Ing. Reinhart Temmel
Ges.m.b.H. & Co.KG.
St. Julienstraße 4a
A-5020 Salzburg

In der Schweiz:
DTZ Data Trade AG
Landstraße 1
CH-5415 Rieden-Baden

Heim Verlag
Heidelberger Landstraße 194
6100 Darmstadt-Eberstadt
Telefon (0 61 51) 5 60 57
Telefax (0 61 51) 5 60 59

len. Da das Verfahren nur wenige Schlüssel erlaubt, wollen wir hier auf ein Listing verzichten.

Bäumchen wechsel' dich

Eine sicherere Möglichkeit ist, die Reihenfolge der Buchstaben nach einer willkürlichen Vertauschungstabelle neu zu sortieren. Der Text wird dabei in kleine Abschnitte mit soviel Buchstaben unterteilt, wie die Vertauschungstabelle an Elementen hat. Da der letzte Abschnitt meist weniger Buchstaben hat, müssen wir noch ein paar willkürliche Buchstaben anhängen. Würden wir die anderen Plätze freilassen, ergäben sich Anhaltspunkte auf die Vertauschungstabelle. Wenn wir die Vertauschungstabelle groß genug machen, können wir den gesamten Text als einen Abschnitt verschlüsseln.

Wenn wir den Text nicht auffüllen wollen, können wir in der Vertauschungstabelle die Felder löschen, die eine Position hinter den vorhandenen Buchstaben darstellt. Wir erstellen also eine neue Tabelle, die nur noch soviel Elemente enthält, wie auch Buchstaben vorhanden sind. Bei der Realisierung würde man allerdings nicht die Tabelle verändern, sondern einfach den Pointer in die Tabelle erhöhen, ohne einen Buchstaben zu kopieren.

Damit die Ver- bzw. Entschlüsselung in einer akzeptablen Zeit erfolgt, müssen wir allerdings den gesamten verschlüsselten Text (bzw. einen Abschnitt) auf einmal im Speicher halten. Im anderen Fall müssen wir viel zu viele Dateizugriffe vornehmen. Im Klartext greifen wir auf die Buchstaben der Reihe nach zu, es reicht daher, wenn wir jeweils kurze Abschnitte im Computer halten. Beim Atari ST sollte der Speicher normalerweise groß genug sein, um den gesamten Text auf einmal zu laden. Ist die Datei jedoch größer, oder wollen wir die verschlüsselte Datei jemandem zusenden, der nur einen Computer mit wenig Speicher hat, müssen wir die Verschlüsselung ebenfalls in kleinen Abschnitten durchführen. Wir sollten in diesem Fall beim zweiten Abschnitt den PSZZG einfach weiterlaufen lassen. Würden wir wieder mit dem ersten Startwert anfangen, wäre der Text leichter zu entschlüsseln.

Wir können die Positionen der Buchstaben mit jedem PSZZG berechnen, der garantiert jeden möglichen Wert mindestens einmal pro Zyklus annimmt (z.B. den PSZZG mit Schieberegister). In diesem Fall ist es möglich, einen PSZZG zu verwenden, der mehr Zahlen liefert, als der Text lang ist. Man kann dann z.B. die letzten n Bit der Zahl (d.h. *modulo* 2^n) als neue Position für den Buchstaben verwenden, wobei n so gewählt wird, das 2^n gerade größer ist als die Länge des Textes. Diese Modulodivision kann man schnell mit *AND* (2^n-*1*) durchführen. Eine andere Möglichkeit ist, den Wert [Zahl des PSZZG] modulo [Länge des Textes] zu verwenden. In diesem Fall müssen wir zwar direkt eine Division durchführen, aber dafür erscheint mir dieses Verfahren sicherer, da jede Zahl mehr als zwei Nachfolger haben kann (Listing 2).

Der Text sollte dabei um einiges kleiner sein als die größte Zahl, die der PSZZG liefert. Von diesem Verhältnis hängt die Anzahl der möglichen Nachfolger ab. Bei sehr langen Dateien sollte man entweder einen größeren PSZZG verwenden oder den Text abschnittweise verschlüsseln. Für die Länge der Abschnitte sollte man auf keinen Fall Zweierpotenzen verwenden.

Wenn wir zur Ermittlung der Zufallszahlen eine Dezimation größer Eins verwenden, müssen wir darauf achten, daß trotzdem alle Zahlen möglich bleiben. Dies ist gegeben, wenn 2^r-1 (r Bit PSZZG) und die Dezimation keinen gemeinsamen Teiler haben.

Wenn wir nur einen Teil der Zahl des PSZZG bzw. das Ergebnis der Modulodivision als Pointer verwenden, müssen wir überprüfen, ob eine bestimmte Position schon einmal (in diesem Abschnitt) errechnet wurde. Dafür benötigen wir ein zusätzliches Array. Da wir nur zwei Zustände benötigen (schon dagewesen, noch nicht dagewesen) verwenden wir ein Boolsches Array. Wollen wir nur Text verschlüsseln, kommt in der Datei also der ASCII-Wert 0 nicht vor, können wir auch den Speicherbereich für den verschlüsselten Text dafür verwenden, indem wir vor dem Verschlüsseln den Bereich auf 0 setzen. Beim Entschlüsseln werden die Zeichen nach der Bearbeitung gelöscht. Diese Methode hat zwei Vorteile:

1) Wir sparen Platz: Das Boolsche Array benötigt 1/8 des Speichers des verschlüsselten Textes.
2) Der Zugriff auf ein Byte-Array ist schneller als der Zugriff auf ein Boolsches Array (Bit-Array).

Dem stehen die Nachteile gegenüber, daß der codierte Text dabei gelöscht wird und daß - wie bereits vorausgesetzt - nicht alle 256 ASCII-Zeichen in der Datei vorkommen dürfen.

Eine Möglichkeit zum Knacken bei diesem Verfahren besteht darin, ausgehend von verschiedenen Startwerten für den PSZZG z.B. die ersten 80 Bytes zu decodieren. Man muß sich dann ansehen, ob die decodierte Buchstabenfolge einen Sinn ergibt. Die Frage ist, ob es eine schnelle Möglichkeit gibt, den Computer mit der Überprüfung zu betrauen. Eine Möglichkeit wäre z.B., daß der Computer die erzeugten Wörter in einem Wörterbuch sucht und alle Versuche verwirft, bei denen nicht ein bestimmter Anteil als gültige Wörter erkannt wurde. Wenn wir uns alle decodierten Strings ansehen müssen, um zu entscheiden, ob der richtige Ansatz dabei ist, haben wir viel zu tun. Die Anzahl der Möglichkeiten steigt dabei mit der Textlänge.

Wenn wir verhindern wollen, daß der Schlüssel durch Decodieren der ersten Bytes erkannt wird, können wir zusätzlich

```
' Funktionen zur Berechnung von Zufallszahlen
' mit n Bit, zweiter Abgriff bei Bit p
' Zählung in gleiche Richtung wie Schieben
' mit Dezimation x ; x<p
' von Georg Scheibler, 4920 Lemgo
' (c) 1992 MAXON Computer
FUNCTION pszzg_l(z%,n%,p%,x%)
  ' Schieben nach links
  neu%=SHR(z%,n%-x%) XOR SHR(z%,p%-x%)
  neu%=neu% AND SHR(-1,32-x%)
  z%=SHL(z%,x%) AND SHR(-1,32-n%)
  RETURN z% OR neu%
ENDFUNC
FUNCTION pszzg_r(z%,n%,p%,x%)
  ' Schieben nach rechts
  neu%=z% XOR SHR(z%,n%-p%)
  neu%=SHL(neu% AND SHR(-1,32-x%),n%-x%)
  z%=SHR(z%,x%)
  RETURN z% OR neu%
ENDFUNC
```

Listing 5: Funktionen zur Berechnung von Zufallszahlen

einen Vorspann aus einer zufälligen Zeichenfolge vor den eigentlichen Text setzen (z.B. einen bereits verschlüsselten Text von 1KB Länge).

Wissen (bzw. vermuten) wir, daß die Positionen mit einem PSZZG auf der Basis eines Schieberegisters ermittelt wurden, bei dem dann die unteren i Bit für die Zahl verwendet wurden, können wir die Tatsache ausnutzen, daß jede Zahl nur zwei mögliche Nachfolger hat. Nach der Zahl x können nur die Zahlen 2*x und 2*x+1 folgen (wenn mit zwei multipliziert wird). Nach dem ersten Zeichen kann nur das zweite oder das dritte Zeichen folgen. Dem zweiten Zeichen folgt das 4. oder 5. Zeichen. Wir haben dann für das zweite Zeichen 4, für das dritte Zeichen acht mögliche Positionen. Nach n Zeichen sind 2^n Positionen möglich. Wir sollten aber nach einigen Zeichen in der Lage sein, uns auf wenige Folgen zu beschränken, weil die anderen keinen Sinn ergeben. Aus dem Weg, den wir jeweils eingeschlagen haben, können wir die weiteren Bits des PSZZG ermitteln, sobald wir soviel Buchstaben gefunden haben, wie Bits am oberen Ende der Zufallszahl abgeschnitten wurden. Je größer der PSZZG ist, desto mehr Buchstaben müssen wir raten. Die weiteren Buchstaben können dann direkt berechnet werden. Wir müssen nur noch herausfinden, an welchen Stellen im decodierten Text wir uns befinden. Zwischendurch kann es dabei zu Störungen kommen, wenn die Positionen schon vorher vergeben wurden. An einigen dieser Stellen müssen wir erneut anfangen, die Bit-Folge aufzubauen.

Man könnte eine Dezimation von x verwenden. Für den ersten Nachfolger ergeben sich dann zwar 2^x Möglichkeiten, aber dafür haben wir dann bereits nach sehr wenigen Buchstaben soviele Bits festgelegt, daß die weiteren Zeichen viel schneller eindeutig festgelegt sind. Wir müssen allerdings häufiger neu anfangen. Der Aufwand wird also schon größer.

Berechnen wir die Position der Zeichen über Modulo [Länge des Textes], gibt es für jede Position viele mögliche Nachfolger, sofern der Text nicht gerade eine Länge von 2^n hat. Die Anzahl hängt davon ab, um welchen Faktor die Zufallszahl größer ist als die Länge des Textes. Man sollte den PSZZG allerdings mit einer großen Zahl starten, damit die Zahl möglichst schnell einmal größer als die Textlänge wird. Starten wir den PSZZG mit 1, wirkt die Modulodivision erst ab dem 10. oder 15. Zeichen, je nach Länge des Textes.

Zumindest, wenn man kein zusätzliches Verfahren verwendet, sollte man die zweite Variante wählen.

Kombinieren wir dieses Verfahren mit einem anderen, das die Buchstaben verändert, wird es viel schwerer, den Schlüssel zu finden. In diesem Fall dürfte die Entscheidung für den richtigen Weg schwerfallen, da dieser noch keinen sinnvollen Text ergibt.

Polyalphabetische Verfahren

Bei Polyalphabetischen Verfahren wird der gleiche Buchstabe an verschiedenen Stellen im Text durch verschiedene Buchstaben ersetzt. Die Entschlüsselung wird dadurch schwieriger.

Wir haben bereits gesehen, daß die Verschlüsselung über Exklusiv-Oder recht einfach, aber bei Verschlüsselung mit einem Byte das Verfahren nicht sehr sicher ist. Vorhin wurde gesagt, daß im Listing 1 eine Zeile gestrichen werden sollte. Dies zeigt bereits, daß im Listing zumindest eine Variante des Verfahrens verwendet wird.

Vernam-Verfahren

Das Verfahren wird sicherer, wenn wir nicht nur ein Byte, sondern eine größere Anzahl Bytes zum Verschlüsseln verwenden. Der erste Buchstabe wird mit dem ersten Byte verschlüsselt, das zweite Zeichen mit dem zweiten Byte usw. bis zum letzten Byte des Schlüssels. Danach beginnen wir wieder mit dem ersten Byte.

Zum Entschlüsseln muß man zunächst herausfinden, wieviel Bytes der Schlüssel hat, um dann aus den Zeichen, die mit dem gleichen Byte verschlüsselt wurden, den Wert der Schlüssel-Bytes zu ermitteln. Im günstigsten Fall haben wir mehr Bytes im Schlüssel, als der Text Zeichen hat. Es gibt dann keine Wiederholung, die es ermöglicht, den Schlüssel zu finden. Dieses Verfahren wurde 1917 von G. S. Vernam erfunden. Laut [1] soll dieses Verfahren für das 'rote Telefon' (tatsächlich ein Fernschreiber) zwischen Washington und Moskau verwendet werden.

Eine Alternative zu einem langen Schlüssel sind zwei relativ kurze Schlüssel mit unterschiedlicher Länge. Wenn wir den Text zunächst mit dem einen und dann mit dem anderen Schlüssel codieren, entspricht dies einem Schlüssel mit einer Länge, die gleich dem kleinsten gemeinsamen Vielfachen der Längen der beiden Schlüssel ist. Diesen Schlüssel kann man ermitteln, indem man die beiden Schlüssel untereinander exklusiv-Oder-verknüpft. Jedesmal, wenn nur ein Schlüssel zu Ende ist, beginnen wird bei diesem String wieder am Anfang. Wenn wir bei beiden Schlüsseln gleichzeitig wieder am Anfang anfangen müssen, beginnt der Schlüssel sich zu wiederholen.

Wenn wir den Schlüssel bei der Nachrichtenübermittlung verwenden wollen, muß auch der Empfänger in dessen Besitz sein. Verwenden wir immer den gleichen Schlüssel, so kann jemand, der mehrere Nachrichten abfängt, ihn daraus irgendwann ermitteln. Man sollte jeden Schlüssel also möglichst selten (am besten nur einmal) verwenden.

Da der Aufwand, einen langen Schlüssel zu übermitteln, recht groß wird, müssen wir uns ein anderes Verfahren überlegen:

Wir können als Schlüssel im Prinzip jede Datei verwenden, die auf beiden Seiten (also beim Sender und Empfänger) vorhanden ist. In der Datei sollten

P,Q Zwei beliebige Primzahlen
N = P * Q F = (P - 1) * (Q - 1)
S beliebige Zahl mit S<F und ggt(F,S) = 1
T beliebige Zahl mit (S * T) mod F = 1

nur N und S dürfen bekanntgegeben werden!!

Verschlüsselung von Zahlen X_i:
Verschlüsseln: $Y = X^S \bmod N$
Entschlüsseln: $X = Y^T \bmod N$

Bild 3: Kurzfassung des RSA-Algorithmus'

allerdings keine größeren Abschnitte vorkommen, die nur ein Zeichen enthalten. Man sollte es sich also gut überlegen, ob ein Bild als Schlüssel geeignet ist.

Sehr große Schlüssel stellen z.B. CD-ROM-Disks dar. Wenn beide Seiten ein entsprechendes Laufwerk haben, kann man verabreden, daß ein bestimmtes CD-ROM als Schlüssel dient. Der Inhalt spielt in diesem Zusammenhang keine Rolle. Man muß nur festlegen, an welcher Stelle der Schlüssel anfängt.

Wir können uns den Schlüssel auch besorgen, indem wir ab einem beliebigen, definierten Zeitpunkt einen Sender empfangen, der digitale Daten sendet. Es spielt dabei keine Rolle, welche Daten dort übertragen werden. Wir müssen die Daten nur als n-Bit-Code interpretieren können.

Eine Möglichkeit wären z.B. die Daten von Wettersatelliten, da das Wetter jeden Tag anders ist. Natürlich müssen Sender und Empfänger die gleichen Daten fehlerfrei empfangen können. Für den normalen Anwender wird der Aufwand für eine Empfangsanlage zu groß. Er muß daher auf andere Verfahren zurückgreifen.

Wir haben bereits einen PSZZG verwendet, um eine große Anzahl Zahlen zu ermitteln. Wir können diese Zahlen auch für die Exklusiv-Oder-Verschlüsselung verwenden. Wenn wir beliebige ASCII-Dateien verschlüsseln wollen, verwenden wir die unteren 8 Bits der Zahl. Ein entsprechendes Programm zeigt Listing 1.

Werden auf diese Weise normale Texte verschlüsselt, so ist dieses Verfahren nicht mehr sicher. Wie in [4] gezeigt wurde, ist es sehr einfach, den Startwert des PSZZG zu ermitteln. In [4] wird zwar nur der PSZZG aus [3] betrachtet (9 Bit, Dezimation 5), aber das Verfahren ist auch bei Verschlüsselung mit einem anderen PSZZG auf der Basis von Schieberegistern anwendbar. Das Knacken wird allerdings schwieriger, wenn der PSZZG mehr Bits hat.

Den Ansatz zum Entschlüsseln liefern die oberen Bits der verschlüsselten Zeichen. Ein normaler Text besteht überwiegend aus Kleinbuchstaben. Die ASCII-Werte dafür sind $6x und $7x. In Binärform ist das %011xxxxx. Wenn wir eine Stelle finden, an der mehrere kleine Buchstaben hintereinander stehen, können wir bei jedem Zeichen drei

$b = 11 = \%1011$

$a^{11} = a^8 * a^2 * a^1$

$a^{2x} = a^x * a^x$

$a^1 \bmod N = 23 \bmod 50 = 23$

$a^2 \bmod N = (23 * 23) \bmod 50 = 29$

$a^4 \bmod N = (29 * 29) \bmod 50 = 41$

$a^8 \bmod N = (41 * 41) \bmod 50 = 31$

$23^{11} \bmod 50 = (23 * 29 * 31) \bmod N = 27$

Bild 4: Berechnung von ab mod N mit a= 23, b = 11, N = 50

Bits der Zufallszahl ermitteln. Bei bekannter (vermuteter) Dezimation kann man die ganze Zufallszahl zusammensetzen. Die Anzahl der kleinen Buchstaben, die man dafür benötigt, hängt von der Größe des Schieberegisters ab. Wenn in dem untersuchten Abschnitt andere Zeichen vorkommen, d.h. die ermittelte Zahl entschlüsselt die nachfolgenden Zeichen nicht richtig, nehmen wir den Abschnitt, der ein Zeichen später anfängt.

Eine Möglichkeit, die verräterischen Bits zu vermeiden, besteht darin, daß wir nicht die unteren 8 Bits der Zufallszahl als Schlüssel verwenden, sondern nur die unteren 5 Bits. In diesem Fall gibt es keine Anhaltspunkte für die Zufallszahl, aber dafür kann man erkennen, zu welcher Gruppe von 32 Zeichen das jeweilige Zeichen gehört. Man erkennt also, ob es sich um einen Klein-, einen Großbuchstaben oder ein Satzzeichen handelt. Die deutschen Umlaute fallen dabei besonders auf, sofern der Atari-(IBM-)ASCII-Code verwendet wird.

Die schwächste Stelle bei dieser Variante sind die Leerzeichen. Zusammen mit den Satzzeichen sind hier die oberen drei Bits gelöscht. Die Satzzeichen unterscheiden sich aber von den Leerzeichen dadurch, daß ihnen (meist) noch ein Leerzeichen folgt. Die Leerzeichen kommen zwar nur in größerem Abstand vor, aber sie geben dann jeweils einen Hinweis auf fünf Bits der Pseudozufallszahl. Diese Methode erscheint mir sicherer als die Verschlüsselung mit den unteren 8 Bits.

5*2oo	10oo+	*2 =	20oo
3o*2oo	6ooo+	*2 =	12ooo
2oo*2oo	4ooo		4oooo
5* 3o	15o+	*2 =	3oo
3o* 3o	9oo		9oo
2oo* 3o	6ooo+		
5* 5	25		25
3o* 5	15o+		
2oo* 5	10oo+		
	55225		55225

mit + gekennzeichnete Produkte sind doppelt

Bild 5: Schnelle Berechnung von Quadraten am Beispiel 235

Da die Leerzeichen bei Verwendung von nur 5 Bits leicht zu finden sind, sollten wir diese am besten gar nicht verschlüsseln. Sie können dann keinen Hinweis auf den Schlüssel geben.

Je nach Anwendung können wir die erlaubten Zeichen einschränken. Wir können z.B. die oberen drei Bits generell löschen oder setzen. Dabei wird jedoch der Zeichensatz auf 32 Zeichen eingeschränkt. Damit die Satzzeichen noch erkannt werden können, müssen sie umcodiert werden (ab besten auf die Zahlen zwischen 26 und 32).

Wenn weiterhin alle Zeichen möglich sein sollen, können wir die Zeichen auf mehrere Zeichensätze verteilen. In diesem Fall müssen wir noch Umschaltzeichen definieren, die dem Decodierprogramm signalisieren, daß das nächste Zeichen aus einem anderen Zeichensatz kommt. Da einige Zeichen durch zwei Zeichen ersetzt werden, wird der Text dadurch etwas länger. Die Zeichen, die am häufigsten vorkommen, sollten im Standardzeichensatz stehen.

Bei normalen Texten empfiehlt es sich, als Standardzeichensatz die Kleinbuchstaben und das Leerzeichen zu definieren. Der größte Wert signalisiert dann, daß es sich bei dem nächsten Zeichen um einen Großbuchstaben handelt. Werden mehr als zwei Zeichensätze verwendet, stellt sich die Frage, ob wir im Standardzeichensatz für jeden weiteren Zeichensatz ein Umschaltzeichen definieren (also z.B. 31 für Alternativzeichensatz 1, 30 für Alternativzeichensatz 2), oder ob wir in jedem - außer dem letzten - Zeichensatz den gleichen Wert zum Weiterschalten auf den nächsten Zeichensatz verwenden. Die zweite Methode hat den Vorteil, daß wir mehr Zeichen im Standardzeichensatz unterbringen können. Die Zeichen in den anderen Zeichensätzen sollten dafür möglichst selten benötigt werden.

In [2] wird gezeigt, wie man den Startwert des PSZZG ermitteln kann, wenn dieser mit der Formel

```
X = FRAC(997*X)
```

arbeitet. Diese Funktion wurde in [1] verwendet. Die Unsicherheit liegt in der begrenzten Mantisse der Fließkommazahlen.

Ein Wechsel der Funktionsvorschrift für den PSZZG hilft also nicht unbedingt, das Verfahren sicherer zu machen.

Damit der Startwert des PSZZG nicht so leicht ermittelt werden kann, sollte der PSZZG möglichst viele Werte erzeugen können. Je mehr Bits für die Zahl verwendet werden, desto größer wird der Aufwand, um die Bits zu ermitteln. Ein Verfahren ist dann sicher, wenn die Computer lange (bis zu mehreren Jahren) benötigen würden, um den Schlüssel zu finden.

Es gibt allerdings auch beim besten PSZZG Bereiche, die nicht so sicher sind. Wird z.B. ein PSZZG auf der Basis von Schieberegistern mit sehr vielen Bits verwendet und als Startwert 256 genommen (oder bald erreicht), haben die nächsten (p-8) Zahlen in den unteren 8 Bits nur Nullen stehen. Der entsprechende Textbereich wird dann nicht verschlüsselt. Es hilft natürlich nichts, stattdessen z.B. andere 8 Bits zu verwenden. Wir haben das gleiche Problem, nur bei einer anderen Startzahl. Man sollte am besten einen Startwert verwenden, bei dem die 1-Bits gut über die ganze Zahl verteilt sind. Bei Verwendung einer größeren Dezimation werden diese Bereiche schneller verlassen.

Wir können bei der Exklusiv-Oder-Verknüpfung auch die Zahlen überspringen, die in den unteren 8 Bits nur Nullen haben. Solange nur wenige Buchstaben uncodiert bleiben, lohnt sich der Aufwand für die Überprüfung nicht.

Nachwirkung

Betrachten wir nun ein paar Verfahren, die beim Verschlüsseln der Zeichen auch die vorhergehenden Zeichen berücksichtigen.

Eine recht einfache Methode besteht darin, die (ASCII-)Werte der Zeichen zu addieren. Wenn das Ergebnis größer als 255 (der größte erlaubte Wert) ist, subtrahieren wir 256 von dem Ergebnis. Für das erste Zeichen können wir einen Wert als Schlüssel vorgeben. Dieser Schlüssel wirkt aber nur auf das erste Zeichen.

Eine leichte Abwandlung des Verfahrens addiert zusätzlich noch eine Konstante (Listing 3). Dadurch ergeben sich 255 verschiedene Codiermöglichkeiten. Für einen Computer also noch eine Kleinigkeit.

Da wir bereits Verfahren mit Exklusiv-Oder-Verknüpfung kennengelernt haben, liegt es nahe, hier ebenfalls dieses Verfahren zu verwenden. Wir können den ASCII-Wert des Zeichens mit dem Wert des vorhergehenden Zeichens verknüpfen. Dies führt allerdings dazu, daß bei Doppelbuchstaben jeweils der zweite Buchstabe den Wert 0 erhält. Dies können wir vermeiden, wenn wir nicht den ASCII-Wert des Zeichens im Klartext verwenden, sondern den Wert des verschlüsselten Zeichens. In diesem Fall ergibt sich der Schlüssel durch Verknüpfung sämtlicher vorhergehender Zeichen.

Die bisher beschriebenen Verfahren, die vom vorhergehenden Text abhängen, haben gemeinsam, daß man den Text leicht entschlüsseln kann, sofern das Verfahren bekannt ist.

Wenn das Verfahren sicherer werden soll, müssen wir andere Verfahren mit integrieren. In Listing 4 wird nicht nur der ASCII-Wert des Zeichens zum letzten Ergebnis sowie ein zusätzlicher Offset addiert, sondern zusätzlich eine Umcodierung über eine Vertauschungstabelle durchgeführt.

Im Listing wird die Umcodierung zweimal durchgeführt. Man kann das Verfahren sicherer machen, indem man noch ein paar Stufen einführt. Durch die Verwendung von Variablen, deren letztes Zeichen die Stufe angibt, zu der es gehört, sollte es für jeden möglich sein, weitere Stufen einzubauen. Jede zusätzliche Stufe ergibt 256mal soviel mögliche Verschlüsselungsarten.

Da der Startwert für den PSZZG ebenfalls ein Schlüssel ist, erhöht sich die Anzahl der Verschlüsselungsmöglichkeiten weiter. Ich habe mir jetzt keine Gedanken gemacht, ob der Faktor für die zusätzlichen Möglichkeiten so groß ist wie die Anzahl der Zahlen, die der gewählte PSZZG liefern kann. Auf jeden Fall ist der Faktor größer als 256.

Welchen Schlüssel nehmen wir jetzt?

Nachdem wir uns eine ganze Anzahl an Verfahren angesehen haben, wollen wir uns mal überlegen, ob ein brauchbares dabei ist. Von den hier vorgestellten Verfahren erscheint mir eine Kombination der Verschlüsselung mit Exklusiv-Oder mit Pseudozufallszahlen und der Vertauschung der Buchstaben nach Pseudozufallszahlen recht sicher zu sein. Die Verschlüsselung mit Exklusiv-Oder verhindert, daß man die Reihenfolge der Buchstaben herausbekommt. Die Vertauschung der Buchstaben verhindert, daß man die Bit-Kombination für den PSZZG für die Exklusiv-Oder-Verschlüsselung ermitteln kann.

	Bereich	Anzahl	Prozent
	bis 100	22	22.0
	bis 10^3	134	13.4
	bis 10^5	7493	7.5
	bis 10^6	63733	6.4
	bis 10^7	558037	5.6
10^{10}	bis $10^{10}+10^4$	487	4.9
10^{12}	bis $10^{12}+2000$	90	4.5

Bild 6: Häufigkeit von Primzahlen

Wir müssen allerdings zwei unabhängige PSZZG zur Berechnung der Zufallszahlen verwenden. Würden wir nur eine Folge von Zufallszahlen verwenden, könnte es passieren, daß wir nicht alle Positionen errechnen können. Wenn bei der Exklusiv-Oder-Verschlüsselung nicht alle Möglichkeiten genutzt werden, so verringert sich nur die Sicherheit des Verfahrens. Wir können durchaus in beiden Fällen die gleiche Funktionsvorschrift verwenden. Wir müssen nur getrennte Reihen von Zufallszahlen benutzen.

Im Listing 2 stehen schon die benötigten Zeilen. Wir müssen nur die als Kommentar gekennzeichneten Zeilen in der 'WHILE' - Schleife freigeben (Zeilen 32, 35,36,39 und 40). Die beiden Zeilen (34 und 38), die bisher das Ver- bzw. Entschlüsseln erledigten, müssen gesperrt werden. Allerdings passiert nichts, wenn

sie nicht gesperrt werden, da das Zeichen, das durch die Zeile verschoben wurde, gleich ersetzt wird. Diese Zeilen kosten also in diesem Fall 'nur' Zeit.

Man könnte die Konstanten für den PSZZG (also Anzahl Bits und mittlerer Abgriff, eventuell auch Dezimation) als zusätzliche Schlüssel definieren. Da die Sicherheit des Verfahrens aber mit der Anzahl der Bits des PSZZG zunimmt, halte ich das nicht für sinnvoll. Wenn man einen PSZZG mit Dezimation verwendet, kann man sie eventuell als zusätzlichen Schlüssel verwenden. Man muß jedoch beachten, daß nur sehr wenige Dezimationen möglich sind.

Wird die Verschlüsselung bei der Datenübertragung verwendet, müssen sich Sender und Empfänger auf ein bestimmtes Format für die Verschlüsselung einigen. Damit sage ich Ihnen wohl nichts Neues. Aber stellen Sie sich mal vor, in einer Gruppe von einigen hundert Personen will jeder mit jedem verschlüsselte Daten austauschen. Dabei sollen die anderen die Daten nicht entschlüsseln können. Man kann dabei natürlich nicht hunderte von Verschlüsselungsprogrammen verwenden. Man muß sich schon auf ein Verfahren einigen.

Haben wir uns auf ein Verfahren geeignet, können wir mit den anderen jeweils die geheimen Schlüssel verabreden. Wir legen uns also eine Liste der Schlüssel an, die dann sicher verwahrt werden muß (man kann wohl kaum alle Schlüssel im Kopf behalten). Jedesmal, wenn wir Daten übertragen wollen, müssen wir nachsehen, welcher Schlüssel zu verwenden ist. Dabei darf natürlich kein anderer den Schlüssel einsehen können.

Stellen wir uns vor, wir haben von jemandem eine Mitteilung im elektronischen Briefkasten. Wir wissen nicht, wer sie uns geschickt hat. Wir müssen dann alle möglichen Schlüssel testen, bis wir den richtigen gefunden haben.

Es wäre doch viel besser, wenn wir einen einzigen Schlüssel wählen könnten, den dann alle verwenden, die uns Nachrichten übermitteln wollen. Wenn alle den gleichen Schlüssel nehmen, müssen wir ihn auch nicht mehr geheimhalten. Er ist ja sowieso schon bekannt. Man könnte die Schlüssel veröffentlichen wie ein Telefonbuch. Natürlich sollen Nachrichten trotzdem sicher verschlüsselt sein.

Die hier vorgestellten Verfahren sind dafür nicht geeignet. Wenn wir eine Datei entschlüsseln wollen, geben wir den gleichen Schlüssel ein, mit dem sie verschlüsselt wurde. Man kann sich also fragen, ob es solche Verfahren überhaupt gibt. Die Frage kann inzwischen mit ja beantwortet werden. Diese Verfahren werden als 'PUBLIC KEY'-Verfahren bezeichnet.

Schlüssel bitte weitergeben

Bei einem 'Public Key'-Verfahren gibt es einen Schlüssel zum Ve- und einen anderen zum Entschlüsseln. Entscheidend ist, daß der Schlüssel für die Entschlüsselung nicht aus dem für die Verschlüsselung ermittelt werden kann. Da es einen eindeutigen Zusammenhang zwischen den beiden Schlüsseln geben muß, basiert die Sicherheit der 'Public Key'-Verfahren darauf, daß der Aufwand zur Ermittlung des Decodierschlüssels so groß wird, daß selbst schnelle Computer dafür Jahrzehnte oder sogar Jahrtausende benötigen würden..

Angesichts der schnellen Entwicklung bei Computern heißt dies, daß ein bestimmter Schlüssel nur für eine begrenzte Zeit sicher ist. Wenn die heutigen Computer allerdings einige Jahrtausende benötigen, um den Schlüssel zu ermitteln, dürfte das Verfahren noch einige Zeit sicher sein.

Das erste 'Public Key'-Verfahren wurde 1976 von den drei Mathematikern Ronald Rivest, Adi Shamir und Leonard Adleman entwickelt und 1978 in [5] veröffentlicht. Nach den Anfangsbuchstaben ihrer Namen wird dieses Verfahren meist als 'RSA'-Verfahren bezeichnet.

Wir wollen uns heute mit diesem Verfahren beschäftigen. Zunächst wollen wir uns ansehen, wie es funktioniert. Danach sehen wir uns an, wie wir uns einen Schlüssel dafür beschaffen. Es folgen einige Bemerkungen zu den Programmen. Zum Schluß wollen wir noch ein paar Überlegungen zur Sicherheit des Verfahrens anstellen.

Das RSA-Verfahren

Beim RSA-Verfahren gibt es drei Schlüsselzahlen. Wir bezeichnen die drei Schlüssel mit S,T und N. Die Schlüsselzahl 'S' wird nur beim Ver-, 'T' nur beim Entschlüsseln, und 'N' beide Male benötigt.

Bei diesem Verfahren werden Zahlen ver- bzw. entschlüsselt. Eine Zahl 'X' wird beim Verschlüsseln in eine Zahl 'Y' überführt. Die Funktionsvorschriften sind:

Verschlüsseln: $Y = X^S \bmod N$
Entschlüsseln: $X = Y^T \bmod N$

Der Operator 'mod' dürfte den meisten bekannt sein. Es handelt sich dabei um die Modulo-Division. Das Ergebnis ist also der Rest, der bei der Division übrigbleibt.

Wenn die drei Schlüsselzahlen richtig gewählt werden, so wird jede Zahl 'X', die kleiner ist als 'N', eindeutig auf eine andere Zahl (Y) im Bereich zwischen 0 und N abgebildet.

Auf den ersten Blick handelt es sich bei diesem nur um eine Variante des Verfahrens, bei dem die Buchstaben (bzw. deren ASCII-Werte) willkürlich gegen andere Buchstaben vertauscht werden.

Wenn wir für 'X' jeweils den ASCII-Wert eines Zeichens verwenden, handelt es sich hier wirklich nur um dieses Verfahren. Es hat dann nur den Nachteil, daß die Zahlen 'Y' größer als 255 werden können (max. n-1) und die Verschlüsselung länger dauert.

Damit das Verfahren sicher wird, müssen wir mehrere Zeichen zusammenfassen. Selbst in längeren Texten werden dann nur ganz selten mehrfach die gleichen Zeichensequenzen zusammengefaßt. Da die ASCII-Zeichen Zahlen bis 255 entsprechen, können wir mehrere Zeichen eindeutig zusammenfassen, indem wir die weiteren Zeichen mit 256^P multiplizieren [P=1,2..]. Wir ersparen uns natürlich die Multiplikation und betrachten p aufeinanderfolgende Zeichen einfach als eine p*8-Bit-Integerzahl. Die Zahl muß natürlich kleiner als 'N' sein. Wir wählen daher p so groß, daß 'X' gerade noch kleiner als 'N' ist.

Es ist sinnvoll, für 'X' soviel Zeichen wie möglich zusammenzufassen, weil die Anzahl der Berechnungen damit möglichst klein gehalten wird. Außerdem ist der Längenunterschied zwischen dem Klartext und dem verschlüsselten Text dabei am geringsten.

Die Schlüssel

Sehen wir uns die Bedingungen für die Schlüsselzahlen an:
Für die Zahl 'N' gilt, daß es sich dabei

um das Produkt von zwei beliebigen (großen) Primzahlen 'P' und 'Q' handeln muß. Diese beiden Primzahlen sind der entscheidende Ausgangspunkt für die Schlüssel. Sie müssen daher geheimgehaltenwerden. Zur Ermittlung der anderen beiden Schlüssel benötigen wir noch eine weitere geheimzuhaltende Zahl 'F'. Sie wird nach der Formel $F = (P - 1) * (Q - 1) = N - P - Q + 1$ berechnet. Für die Zahl 'S' gelten die Bedingungen, daß 'S'kleiner als 'F' sein muß und mit 'F' keinen ge-meinsamen Teiler haben darf. Für die Schlüsselzahl 'T' gilt: $(S * T) \bmod F = 1$. Diese Aussage ist äquivalent mit der Aussage $(S * T) = a * F + 1$. Dabei ist 'a' eine ganze Zahl. Die Zahl 'T' muß ebenfalls geheim bleiben. In Bild 3 sind die Bedingungen noch einmal zusammengefaßt. Mit den so ermittelten Schlüsselzahlen wird jede Zahl 'X', die kleiner ist als 'N', beim Verschlüsseln eindeutig in eine Zahl 'Y' überführt, die ebenfalls kleiner als 'N' ist. Den Beweis dafür wollen wir hier nicht antreten.

Ich möchte aber noch darauf hinweisen, daß die Zahlen 'S' und 'T' ausgetauscht werden können. Wir können also auch die Verschlüsselung mit der Vorschrift $Y = X^T \bmod N$ vornehmen und dann mit $X = Y^S \bmod N$ wieder entschlüsseln. Es reicht also, wenn einer der beiden Schlüssel keinen gemeinsamen Teiler mit 'F' hat. Sie werden sich jetzt sicher fragen, was Sie davon haben, wenn Sie die beiden Schlüsselzahlen austauschen können: Die Zeit für die Ver- bzw. Entschlüsselung hängt von der Größe der Schlüsselzahl 'S' bzw. 'T' ab. Wird ein verschlüsselter Text häufiger entschlüsselt, kann man Zeit sparen, wenn 'T' möglichst klein ist. Wird der Text auf verschiedenen Computern ver- und entschlüsselt, kann man dem schnelleren Computer den größeren Schlüssel zuordnen.

Da es sich bei diesem Verfahren um ein 'Public Key'-Verfahren handelt, können wir die beiden Zahlen, die zum Verschlüsseln benötigt werden ('N' und 'S'), veröffentlichen. Außenstehende sind nicht in der Lage, die Schlüsselzahl 'T' in kurzer Zeit zu ermitteln, sofern wir für 'N' eine sehr große Zahl verwenden. Die erforderliche Größe von 'N' hängt dabei von der Rechenleistung der Computer ab. Je schneller die Computer werden, desto größer muß 'N' werden. Wir werden nachher noch die Zeiten abschätzen, die dafür nötig sind.

Die Programme

Nachdem wir wissen, wie der Algorithmus aussieht, können wir uns dem Programm zuwenden (Die beiden Assembler- und das GFA-Programme befinden sich auf den Disketten zum Heft).

Im ersten Assemblerprogramm ist die Funktion zum Ver- und Entschlüsseln von Speicherbereichen realisiert. Die Schlüsselzahlen werden als 127-Bit-Integerzahlen übergeben. Damit können für die Schlüsselzahlen bis zu 38stellige Zahlen verwendet werden [max. $127*\log_{10}(2)$]. Die Begrenzung habe ich so gewählt, weil

1) der Aufwand zur Ermittlung der Schlüsselzahlen sonst (auf dem Atari ST) zu groß wird,
2) wir bei größeren Zahlen die Modulodivision nicht mehr in den Registern des Prozessors durchführen können. Die benötigte Rechenzeit würde damit drastisch ansteigen.

Die Anzahl der Zeichen, die jeweils zu einer Zahl zusammengefaßt werden, richtet sich nach der Größe von 'N'. Es wird ein Byte weniger verwendet, als zum Speichern von 'N' benötigt werden. Ist 'N' z.B. eine Zahl mit 100 Bits, werden zur Darstellung der Zahl 13 Bytes benötigt. Für die Zahl 'X' werden dann jeweils 12 Bytes zusammengefaßt. Die Zahl 'Y' hat 13 Byte, weil das Ergebnis der Berechnung max. N-1 ist. Wir benötigen für das Ergebnis genauso viel Bits wie für 'N'. Man könnte die Anzahl Bytes, die jeweils zu einer Zahl 'X' zusammengefaßt werden, variabel machen: Ist das erste Zeichen für 'X' kleiner als das höchste Byte (MSB) von 'N' [aber ungleich 0], könnten wir ein Zeichen mehr für die Zahl 'X' verwenden. Insbesondere, wenn das MSB von 'N' größer als 128 ist, würde der verschlüsselte Text kaum länger sein als der Klartext. Ich habe auf diese Möglichkeit verzichtet, weil man dann nicht mehr ausrechnen kann, an welcher Stelle im verschlüsselten Text eine bestimmte Textpassage steht. Man kann jetzt durchaus einzelne Abschnitte des verschlüsselten Textes durch einen anderen Text ersetzen.

Entsprechend dem üblichen Zahlenformat bei Motorola ist das erste Zeichen der jeweiligen Zeichenkette das MSB. Beim Datenaustausch mit Computern, die Intel-Prozessoren verwenden, muß man darauf achten, daß sie das gleiche Format verwenden.

Rechnen mit Resten

Den größte Teil des Listings nimmt die Funktion zum Berechnen von $a^b \bmod N$ ein. Wir wollen uns daher etwas ausführlicher darauf eingehen.

Wir benötigen das genaue Ergebnis der Berechnung und können daher nicht auf die Berechnung mit Fließkommazahlen zurückgreifen. Wir können aber auch nicht zunächst a^b berechnen. Wenn die Zahl a 100 Bits hat, benötigen wir für a^2 bereits 200 Bits. Für ab benötigen b*100 Bits. Jeder noch so große Speicher ist dafür zu klein. Die Berechnung würde auch zu lange dauern.

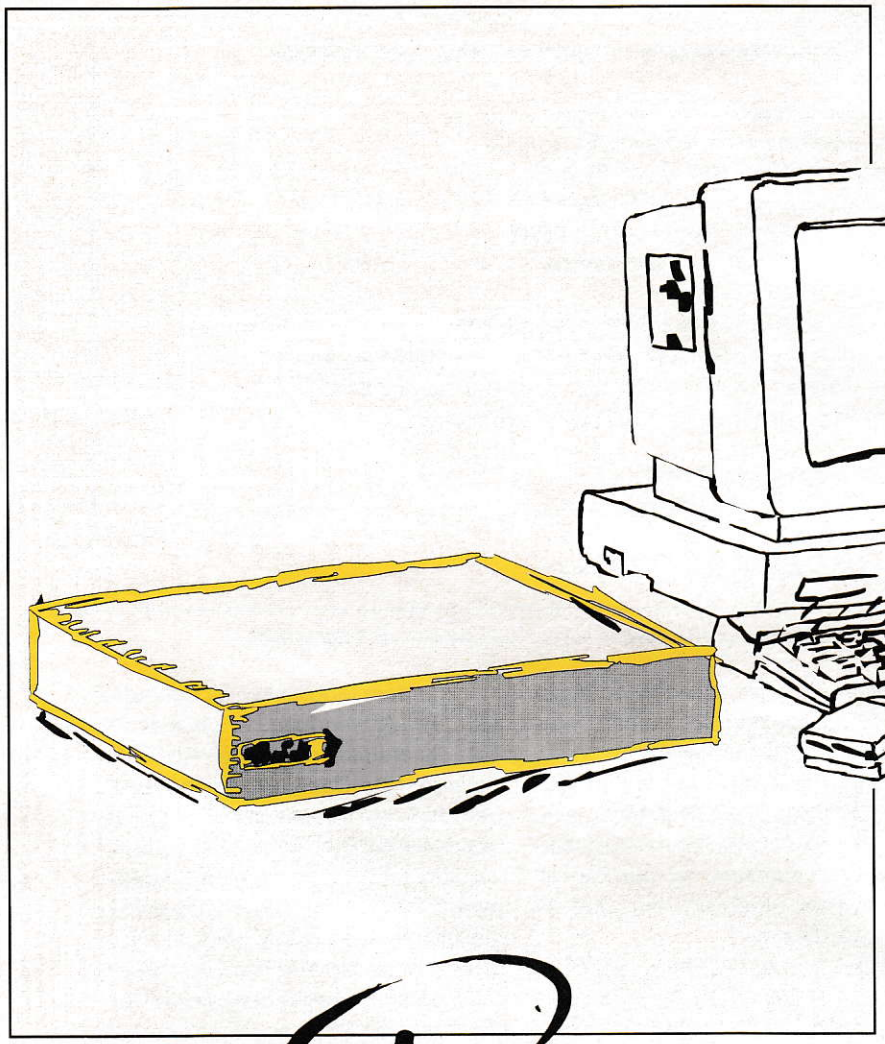
Wir sind allerdings auch nicht an dem Ergebnis von ab interessiert, sondern an dem Ergebnis von $a^b \bmod N$. Wir können ausnutzen, daß $(c * d) \bmod e$ das gleiche Ergebnis liefert wie $((c \bmod e) * (d \bmod e)) \bmod e$. Die Zahl a^b könnten wir auch als Produkt von b mal den Faktor a schreiben. Wir müßten dann b-1 mal mit a multiplizieren und die Modulodivision durchführen. Wir können aber auch die Zahl b in ihre Zweierpotenzen zerlegen und dann die entsprechenden Potenzen von a modulo N berechnen. Die Zweierpotenzen werden als Produkt der jeweils nächstkleineren Zweierpotenz berechnet ($a^2x = a^x * a^x$). Es reichen dann weniger als $2*\log_2(b)$ anstatt b Multiplikationen. Dabei werden $\log_2(b)$ Multiplikationen zur Ermittlung der Zweierpotenzen benötigt. Für jedes gesetzte Bit in b ist dann noch die Multiplikation des Teilergebnisses mit der entsprechenden Zweierpotenz durchzuführen. In Bild 2 ist dies an einem Beispiel dargestellt.

Die Berechnung von $(c * d) \bmod e$ könnte man analog durch Additionen und Subtraktionen zurückführen. Die Berechnung geht allerdings schneller, wenn wir bei der Multiplikation und Division bleiben. In Assembler können wir ja entsprechend mehr Platz für das Zwischenergebnis reservieren.

Wer sich das Listing ansieht, wird wahrscheinlich erstaunt feststellen, daß neben der Funktion zur Berechnung von a*b (abmodn) ein zweiter Programmteil zur Berechnung von a^2 (ah2modn) vorhanden ist. Der Grund liegt darin, daß ich a^2 schneller berechnen kann als a*b. Um dies zu verstehen, müssen wir

Die hohe Kunst

ein neues Festplatten-System zu konzipieren



1. ProFile II, für alle Atari ST/STE/TT; externe Festplatten mit ACSI- und SCSI-Anschluß, Hardware-Schreibschutz, thermisch geregeltem Lüfter und komfortabler Software mit vielen Extras.

2. ProFile SCSI, externe Festplatten in kleinem Gehäuse. Sie arbeiten an allen Computern mit externem SCSI-Anschluß (z.B. Atari TT, Macintosh, MS-DOS-PCs u.a.). Mit Kapazitäten von 20-500MB. Selbstverständlich auch als Wechsel-Festplatte mit 42 oder 84 MB erhältlich.

3. ProFile intern, Festplatten zum Selbsteinbau in den TT und Mega STE. Mit Speicherkapazitäten von 100 bis 500 MB.

4. ProFile II Tape Streamer T60/T150. 60 bzw. 150 MB; mit ACSI- und SCSI-Anschluß, komfortable Backup-Software mit Desktop-Oberfläche und Batch-Sprache.

PROFILE II

- Komfortable Verwaltungssoftware
- Herausgeführter SCSI-Bus
- Leise, durch thermisch geregelten Lüfter
- Per Software aktivierbarer Hardware-Schreibschutz
- Treibersoftware mit vielen Extras (z.B. grafische Zugriffsanzeige, Lesen Schreiben von MS-DOS-Wechselplatten)

protar Elektronik GmbH
Alt-Moabit 91D • 1000 Berlin 21
Tel 030 391 20 02 • Fax 030 391 73 32
Vertretung in Österreich:
Dipl. Ing. R. Temmel GmbH & Co KG
St. Julienstraße 4a • 5020 Salzburg
Tel 0662 71 81 64 • Fax 06244 71 88 3
Vertretung in der Schweiz:
DTZ Data Trade AG
Landstraße 1 • CH 5415 Rieden/Baden
Telefon 056 82 18 80 • Fax 056 82 18 84

protar

uns mal ansehen, wie wir mehrstellige Zahlen schriftlich multiplizieren:

Wir multiplizieren jede Ziffer des ersten Faktors mit jeder Ziffer des zweiten Faktors. Üblicherweise werden die Produkte, die zu einer Ziffer des zweiten Produkts gehören, sofort addiert. Würden wir dies nicht machen, würden wir bei der Berechnung von a*a sehen, daß wir die meisten Produkte zweimal berechnen. Es reicht natürlich, wenn wir das Produkt der Ziffern einmal berechnen und dann das Ergebnis zweimal addieren. Wir können also fast die Hälfte der Multiplikationen sparen. In Bild 5 habe ich die Berechnung von 235^2 mal entsprechend aufgeschlüsselt.

Im Assemblerprogramm befindet sicht nur direkt die Verschlüsselungsroutine. Der Funktion werden die Parameter 'C'-kompatibel übergeben. Für die anderen Aufgaben reicht die Geschwindigkeit einer Hochsprache. Diese Aufgaben werden von einem GFA-BASIC-Programm erledigt. Das BASIC-Programm lädt die Assembler-Routine nach. Dazu wird das Listing 6 wie ein normales, eigenständiges Programm assembliert und in 'CRYPTRSA.ASM' umbenannt. Dadurch soll verdeutlicht werden, daß es sich nicht um ein eigenständig lauffähiges Programm handelt. Beim 'DEVPAC'-Assembler, mit dem ich das Programm geschrieben habe, hätte man das Programm auch als Binär-File speichern können. In diesem Fall würde der 'SEEK'-Befehl entfallen. Ich weiß aber nicht, ob alle Assembler diese Möglichkeit bieten. Ein Programm kann man jedoch mit jedem Assembler erzeugen. Damit das Listing möglichst unverändert auch von anderen Assemblern verarbeitet wird, habe ich die Offset-Tabelle über 'EQU' definiert.

Da die Schlüsselzahlen sehr groß sind, werden sie nicht im Dialog erfragt, sondern sind in den DATA-Statements abgelegt. Wenn man die Schlüsselzahl 'T' mit einträgt, muß man natürlich dafür sorgen, daß diese Programmversion nicht von anderen Personen genutzt werden kann. Man sollte das Programm dann genauso sicher aufbewahren wie seine Tresorschlüssel. Man sollte sich vielleicht zwei Versionen des Programms machen. Die eine enthält nur die Schlüssel zum Verschlüsseln von Daten. Diese Version kann man bei seinen normalen Disketten lagern oder auf der Festplatte ablegen.

```
Es müssen zwei (große) Zahlen vorgegeben werden (max. 19 Ziffern)

bitte 1. Zahl vorgeben:  999999999.___.___|
bitte 2. Zahl vorgeben:  888888888.___.___|
P = 1.000.000.007  00:00:00,39  21:25:36
888.888.893 3.491  00:00:00,44  21:25:38
888.888.899 19.141  00:00:00,68  21:25:38
Q = 888.888.901  00:00:01,05  21:25:38

Kombinationen für N = 888.888.907.222.222.307  00:00:01,06  21:25:38
S = 271 T  = 9.840.098.583.025.831       a = 3  00:03:41,66  21:29:18
S = 77.837.087  T = 45.679.453.823       a = 4  00:03:48,69  21:29:26
S = 17  T  = 261.437.913.333.333.353     a = 5  00:08:02,43  21:33:38

N = 888.888.907.222.222.307
S = 271 T  = 9.840.098.583.025.831
S = 77.837.087  T = 45.679.453.823
S = 17  T  = 261.437.913.333.333.353
```

Bild 7: Ein verkürzter Lauf des Programms zur Berechnung von Schlüsselzahlen

Die andere Version die den persönlichen Decodierschlüssel enthält, wird gut verschlossen aufbewahrt.

Die Schlüsselzahlen dürfen (Tausender-)Punkte enthalten. Wenn man eine so große Zahl eingibt, kann man schon leicht Fehler machen. Ist die Zahl in Dreiergruppen unterteilt, kann man sie leichter vergleichen.

Im Listing sind ein paar Beispielschlüssel eingetragen. Die ersten beiden Schlüssel ergänzen sich. Im ersten Schlüssel fehlt die Schlüsselzahl 'S', im Zweiten die Zahl 'T'. Zum letzten Schlüssel kennt nur mein Programm die Schlüsselzahl 'T'.

Das Unterprogramm 's_to_i' wandelt den String mit der Zahl in eine 128-Bit-Zahl um. Man sieht daran, daß man auch in BASIC mit sehr großen Zahlen umgehen kann. Der Aufwand wird dabei nur größer und das Programm langsamer.

Das Programm kann im Dialog genutzt werden. Dabei werden die Quell- und Zieldatei über die bekannte Dialogbox erfragt. Danach wird gefragt, welcher Schlüssel verwendet werden soll. Als letztes gibt man noch an, ob man die Quelldatei ver- oder entschlüsseln will. Da man sich die Nummer für den Schlüssel nicht so gut merken kann, kann man das Programm eventuell ändern, so daß jedem Schlüssel ein Name zugeordnet wird. Der Schlüssel wird dann nach dem Namen des Adressaten gesucht. Die bequemste Wahlmöglichkeit bietet wohl eine Dialogbox, in der die Namen eingetragen werden.

Eine andere Möglichkeit ist der Aufruf als 'TTP'-Anwendung. In diesem Fall wird als erstes ein 'C' oder ein 'D' für Codieren bzw. Decodieren eingegeben. Dahinter folgen die Nummer des Schlüssels sowie die Dateinamen für die Quell- und Zieldatei. Die einzelnen Eingabeparameter werden durch Leerzeichen getrennt. Das erste Zeichen des Commandstrings darf (wie beim Aufruf vom Desktop aus) dessen Länge enthalten. Der String wird aber auch korrekt bearbeitet, wenn diese Angabe nicht vorhanden ist. Die Parameterübergabe über den Commandstring ist allerdings nur beim compilierten Programm verwendbar. Wird das Programm über 'Anwendung anmelden' vom Desktop aus gestartet, enthält der Commandstring den Namen der BASIC-Datei. Das Programm meldet dann 'Befehl nicht erkannt'. Damit das Programm gestartet werden kann, muß man an den Anfang der Commandline ein 0-Byte schreiben:

```
POKE BASEPAGE+128,0
```

Damit jede Datei eindeutig wieder entschlüsselt werden kann, wird vor dem Verschlüsseln hinter dem Klartext ein $FF-Byte geschrieben. Nach dem Decodieren wird die zurückgegebene Länge nach diesem $FF-Byte korrigiert. Hinter dem $FF-Byte folgen nur 0-Bytes.

Um das Programm zu testen, nehmen wir einen kurzen Text und verschlüsseln ihn. Anschließend wird der Text wieder decodiert und unter einem anderen

Namen gespeichert. Jetzt können wir prüfen, ob die Dateien identisch sind.

Berechnung der Schlüssel

Wir sind jetzt in der Lage, Daten zu ver- und zu entschlüsseln, sofern uns jemand die benötigten Schlüsselzahlen gibt. Damit unser persönlicher Schlüssel wirklich geheim ist, wollen wir ihn natürlich selbst ermitteln. Wenden wir uns daher jetzt der Berechnung der Schlüsselzahlen zu (Zweites (Assembler-)Program auf den Disketten.)

Wir haben bereits erfahren, daß die Zahl 'N' das Produkt von zwei Primzahlen ist. Wir müssen also zunächst zwei Primzahlen ermitteln.

Wir erinnern uns an die Definition von Primzahlen:

Eine Zahl ist eine Primzahl, wenn sie nur durch Eins und durch sich selbst ohne Rest teilbar ist.

Wir nehmen uns also eine beliebige Zahl und teilen sie durch alle Zahlen, die kleiner sind als diese Zahl. Bleibt dabei einmal kein Rest, können wir aufhören, da wir uns keine Primzahl ausgesucht haben. Wir müssen dann bei der nächstgrößeren Zahl wieder von vorn anfangen. Wenn man sich die Anzahl der erforderlichen Divisionen vorstellt, überlegt man sich, ob man nicht mit weniger Divisionen auskommen kann:

Es reicht, wenn man durch die Zahlen teilt, die kleiner sind als die Wurzel der Zahl. Ist der Divisor größer als die Wurzel, wird das Ergebnis kleiner. Bei der Division durch das Ergebnis hätten wir bereits gemerkt, daß es sich nicht um eine Primzahl handelt.

Da die Primzahl ungerade ist (zwei ist die einzige gerade Primzahl, weil alle geraden Zahlen durch zwei teilbar sind), müssen die möglichen Teiler ebenfalls ungerade sein.

Wir haben jetzt die Anzahl der erforderlichen Divisionen schon erheblich reduziert, sind aber noch nicht zufrieden. Es würde doch reichen, wenn wir die Zahl nur durch die Primzahlen teilen. Wenn wir durch eine andere Zahl teilen, hätten wir bereits bei deren Primfaktoren festgestellt, daß die Zahl keine Primzahl ist. Wenn wir die Primzahlen erst ermitteln müssen, wird der Aufwand dafür größer als die gesparte Zeit.

Als Kompromiß können wir einen Teil der Nichtprimzahlen weglassen. Wir haben bereits alle Zahlen weggelassen, die durch zwei teilbar sind. Als nächstes lassen wir auch alle Zahlen weg, die durch drei teilbar sind. In diesem Fall müssen wir zum Divisor abwechselnd einmal Zwei und einmal Vier addieren (außer am Anfang). Je mehr Primzahlen wir berücksichtigen, desto komplizierter wird die Abstandsliste.

Sieb des Eratosthenes

Im 2. Jh. v. Chr. hat Eratosthenes ein Verfahren erdacht, mit dem alle Primzahlen bis zu einer bestimmten Zahl ermittelt werden können: Wir schreiben zunächst alle Zahlen auf. Nun beginnen wir bei der Zahl Zwei und streichen dann jede zweite Zahl aus der Liste. Nun sehen wir in der Liste nach, welches die kleinste Zahl ist, die noch nicht durchgestrichen ist. Wir finden dabei die Drei. Wir streichen daher jede dritte Zahl durch. Die Zahlen, die bereits durchgestrichen sind, müssen dabei mitgezählt werden. Dies wiederholen wir solange, bis wir alle Zahlen durchgetestet haben.

Wenn wir nur die ersten Primzahlen sieben, haben wir eine Liste, in der ein Teil der Zahlen gestrichen ist, die keine Primzahlen sind. Da sich die Abstände ab dem Produkt der gesiebten Primzahlen wiederholen, können wir wieder am Anfang der Liste anfangen. Das heißt, nicht ganz am Anfang, da die Primzahlen selbst nicht durchgestrichen sind. Wir nehmen daher den Abschnitt hinter der größten gesiebten Primzahl. Im Programm werden noch die Abstände bis zur nächsten nicht gestrichenen Zahl ermittelt, damit wir die Zahlen nicht testen müssen.

Da wir die geraden Zahlen gar nicht beachten müssen, wird in der Tabelle nur jeder zweite Platz verwendet. Der Bereich dazwischen findet für ein zweites Teilsieb Verwendung.

Wir wollen uns jetzt mal überlegen, wieviel Divisionen wir mit der aufwendigen Tabelle sparen: Wenn wir die Primzahl p zusätzlich sieben, wird jede p-te Zahl gestrichen. Ein Teil der Zahlen war zwar bereits gestrichen, aber der Anteil ist bei den nicht gestrichenen Zahlen genauso groß. Die Ersparnis liegt daher bei 1/p der bisherigen Zeit. Anders ausgedrückt, wir können die Zeit mit (p-1)/p multiplizieren. Werden alle Primzahlen von 3 bis 31 verwendet, ergibt sich als Faktor 0,306. Die benötigte Zeit wird

also auf ca. 30 Prozent reduziert. Da die Berücksichtigung der Zahl 2 bereits eine Halbierung der zu prüfenden Zahlen brachte, müssen wir also ca. 15 Prozent der Zahlen überprüfen. In Bild 6 ist die Häufigkeit der Primzahlen für einige Bereiche angegeben. Wir sehen daran, daß wir 'nur' noch dreimal soviel Divisionen durchführen, wie mindestens erforderlich wären.

In Bild 7 ist ein verkürzter Lauf des Programms wiedergegeben. Die beiden Zeilen zwischen 'P=' und 'Q=' zeigen, bei welchen Zahlen zwischen der vorgegebenen Zahl und der Primzahl der kleinste Primfaktor schon etwas größer ist. Bei jeder Ausgabe werden die Rechenzeit und die Uhrzeit angegeben. Bei dem Beispiel ist die Zeit noch nicht so interessant. Werden die Primzahlen aber größer, kann man besser abschätzen, wann die nächste Zahl ausgegeben wird. Ich wollte die Ausgabe der Zeit schon weglassen, aber so könnt Ihr Euch selbst von dem Anstieg der Rechenzeit überzeugen, ohne gleich große Zahlen vorgeben zu müssen.

Wenn die vorgegebene Zahl eine Dezimalstelle mehr hat, dauert die Überprüfung ca. dreimal so lange. Wird die größte 18stellige Zahl vorgegeben, dauert es ca. 12,5 Std. bis die erste 19stellige Zahl gefunden wird. Geben wir die größte 19stellige Zahl vor, müssen wir ca. 60 Std. warten bevor feststeht, daß die Zahl (10^{20}+51) die erste 20stellige Primzahl ist. In diesem Fall dauert es ca. 17 Std., bis das Programm den kleinsten Divisor von (10^{20}+49) gefunden hat. Da ich nicht so lange auf den Computer verzichten will (die weiteren Zahlen müssen ja auch noch berechnet werden), kann die Berechnung alle paar Sekunden mit einem beliebigen Tastendruck unterbrochen werden. Man kann dann den aktuellen Stand speichern und die Berechnung irgendwann (z.B. in der nächsten Nacht) fortsetzen, indem man anstelle der ersten Primzahl ein 'l' für Laden eingibt. Wird bei der Eingabe nur die [RETURN]-Taste gedrückt, wird das Programm beendet.

Die Divisionen bei der Primzahlsuche werden von dem Unterprogramm 'Rdivi' durchgeführt. Dieses Unterprogramm ist für eine 64-Bit-Division relativ lang. Man könnte die Division auch mit weniger als 20 Zeilen durchführen. Durch die vielen Fallunterscheidungen benötigt die Funktion allerdings weniger als die Hälfte der Zeit.

Wenn wir uns zwei Primzahlen ausgesucht haben, können wir 'N' und 'F' ausrechnen. Wie aber kommen wir an 'S' und 'T'?

In [6] wird vorgeschlagen, sich eine beliebige Zahl 'a' auszusuchen. Mit dieser Zahl berechnen wir nun *AF1 = a * F + 1*. Das Ergebnis ist ein mögliches Produkt von 'S' und 'T'. Nun zerlegen wir die Zahlen 'AF1' und 'F' in ihre Primfaktoren. Alle Primfaktoren von 'AF1', die auch Primfaktoren von 'F' sind, werden für die Zahl 'T' verwendet. Die verbleibenden Zahlen werden so auf 'S' und 'T' verteilt, daß beide Zahlen in etwa gleich groß sind.

Dieses Vorgehen ist zwar im Prinzip durchführbar, aber der Zeitaufwand kann sehr groß werden. Es besteht die Möglichkeit, daß die Zahl 'AF1' eine Primzahl ist. Die Zahl 'F' kann keine Primzahl sein, da sie das Produkt von zwei geraden Zahlen ist. Sie enthält zumindest zweimal den Primfaktor 2. Die Zahl F/4 kann aber schon eine Primzahl sein.

Wir können uns die Zerlegung von 'F' in Primfaktoren sparen. Um zu prüfen, ob ein Primfaktor von 'AF1' auch ein Primfaktor von 'F' ist, müssen wir nur 'F' durch diese Zahl teilen. Da die Zahl 'AF1' maximal ca. 80 Primfaktoren haben kann (wenn es sich um 3^x handelt), benötigen wir dafür erheblich weniger Zeit als für die Primfaktorzerlegung. Der Aufwand für eine vollständige Zerlegung von 'AF1' wird meist ebenfalls zu groß sein. Wir begnügen uns daher damit, die Primfaktoren zu suchen, die kleiner als eine bestimmte Zahl sind. Der Rest von 'AF1' wird einfach als Faktor von 'T' verwendet. Von den gefundenen Faktoren werden die Teiler von 'F' ebenfalls als Faktor von 'T' verwendet. Die verbleibenden Faktoren werden so aufgeteilt, daß 'S' und 'T' möglichst gleich groß werden. Bei großen Zahlen heißt dies in der Praxis meist, daß sie alle Faktoren von 'S' werden.

Das Programm nimmt für die Zahl 'a' alle Werte von 3 bis 13. Man kann sich dann die Kombination für 'S' und 'T' aussuchen, bei der beide Zahlen in etwa gleich groß sind. Nachdem alle Zahlen getestet wurden, werden nochmal alle gefundenen Kombinationen ausgegeben. Dies gilt auch für die Kombinationen, die vor eventuellen Unterbrechungen ermittelt wurden. Man muß diese Zahlen also nicht notieren.

Ist das Verfahren wirklich sicher ?

Um das Verfahren zu knacken, müssen wir die beiden Primzahlen finden, die für die Berechnung von 'N' verwendet wurden. Die zweite Primzahl können wir natürlich sofort berechnen, wenn wir die erste haben. Sobald wir diese Primzahlen haben, können wir 'F' berechnen. Um den fehlenden Schlüssel 'T' zu ermitteln, ist nur noch wenig Aufwand nötig. Aus dem Verhältnis von 'S' zu 'F mod S' kann man mögliche Werte für 'a' abschätzen.

Es liegt nahe zu versuchen, ob wir 'F' nicht auch so raten können. Da $F = (Q-1)*(P-1) = N-Q-P+1$ ist, können wir zwar eine obere Grenzen für 'F' angeben: $F < N-2*sqr(N)$. Ist aber z.B. $Q \approx P*10$, ergibt sich $F \approx N-3,5*sqr(N)$. Es müssen mehr Zahlen getestet werden als bei der Ermittlung der Faktoren von 'N'. Zudem kommen für 'a' jeweils sehr viele Zahlen in Frage.

Die entscheidende Frage ist also, wie lange es - statistisch gesehen - dauert, bis wir einen Primfaktor von 'N' finden. Wie wir bereits bei der Ermittlung der Primzahlen gesehen haben, steigt die Anzahl der möglichen Teiler für eine Zahl mit deren Größe. Wird die Zahl um zwei Dezimalstellen größer (also hundertmal so groß), steigt die Anzahl der möglichen Teiler um den Faktor 10. Es spielt dabei kaum eine Rolle, ob wir nur die Primzahlen überprüfen oder - wie bei unserem Programm - die dreifache Menge.

Setzen wir einen etwas schnelleren Computer voraus (z.B. einen Atari ST mit einem 16-MHz-Prozessor). Nehmen wir an, daß dieser Computer die erste 20stellige Zahl in 24 Std. überprüfen kann. Für die Zahl 'N' nehmen wir an, daß Sie 38 Dezimalstellen hat. Da die Zahl 18 Dezimalstellen mehr hat, benötigt das Programm 109 Tage dafür. Dies entspricht 2,7 Millionen Jahre. Dabei wurde noch nicht berücksichtigt, daß die Division länger dauert, wenn die Zahlen größer sind. Wenn wir annehmen, daß wir nur die Primzahlen prüfen, können wir den Zeitaufwand auf ca. 1 Million Jahre reduzieren.

Die geschätzten Zeiten beziehen sich auf den ungünstigsten Fall, daß wir bei

den kleinsten Primzahlen anfangen und die beiden Primzahlen ungefähr gleich groß sind. Wenn beide Primzahlen fast gleich groß sind, finden wir sie schneller, wenn wir bei Wurzel N anfangen und dann abwärts die Zahlen testen.

Nehmen wir an, daß die größere Primzahl (Q) zehnmal so groß ist wie die kleinere (P), so ist $P = sqr(N)/sqr(10) \approx sqr(N)/3$. Es hilft also nicht unbedingt, bei sqr(N) anzufangen.

Nehmen wir an, daß wir die Primfaktoren nach einem Zehntel der möglichen Divisionen finden, so bleiben immer noch 100 000 Jahre. Die Zeitersparnis ist dabei schon recht großzügig gerechnet.

Eine weitere Beschleunigung kann nur noch der Einsatz von Technik bringen. Die Rechengeschwindigkeit des ST ist zwar für viele Zwecke ausreichend, aber es gibt Computer, die erheblich schneller sind. Der Inbegriff für schnelle Computer ist sicherlich die 'Cray'. Ich weiß nicht, wie lange dieser Computer für die Berechnung benötigen wird. Es ist auf jeden Fall mehr, als das Verhältnis der Taktfrequenzen ergibt. Die 'Cray' arbeitet mit 250 MHz, sie ist von daher also mindestens 30mal so schnell.

Zusätzlich hat die Cray einen breiteren Datenbus und eine schnellere Architektur. Dadurch wird der Computer noch einmal schneller.

Billigen wir einem Supercomputer zu, daß er 10000mal so schnell rechnet wie der ST, so benötigt er immer noch 10 Jahre, um den Schlüssel zu ermitteln. Man könnte die Zeit natürlich verkürzen, wenn man die Arbeit auf mehrere Computer verteilen würden, aber wer hat schon mehrere Supercomputer zur Verfügung?

Solange also nicht jemand den Schlüssel knacken will, für den Geld keine Rolle spielt, dürften die ermittelten Schlüssel schon sehr sicher sein. Das Risiko, daß der Schlüssel gestohlen wird, dürfte größer sein, als das, daß der Decodier- aus dem öffentlichen Codierschlüssel rekonstruiert wird.

All jenen, denen die mit diesem Programm ermittelten Schlüssel noch nicht sicher genug sind, kann ich nur empfehlen, mit schnelleren Computern größere Schlüssel zu berechnen. Haben die Primzahlen zwei Dezimalstellen mehr, wird für die Überprüfung bereits die zehnfache Zeit benötigt. Um den Schlüssel zu knacken, ist die hundertfache Zeit notwendig.

Literatur

[1] Karl Schlessmann, Ein unknackbares Chiffrierverfahren, MC 1/89, S. 86 - 89

[2] Gunter Laßmann, Der Schlüssel zum Schlüssel, MC 7/89, S. 50 - 51

[3] Heinz-Georg Nacke, Eine harte Nuß für „Knack", MC 9/90, S. 84 - 92

[4] Gunter Laßmann, „Knack" schlägt zurück,MC 2/91, S. 120 - 121

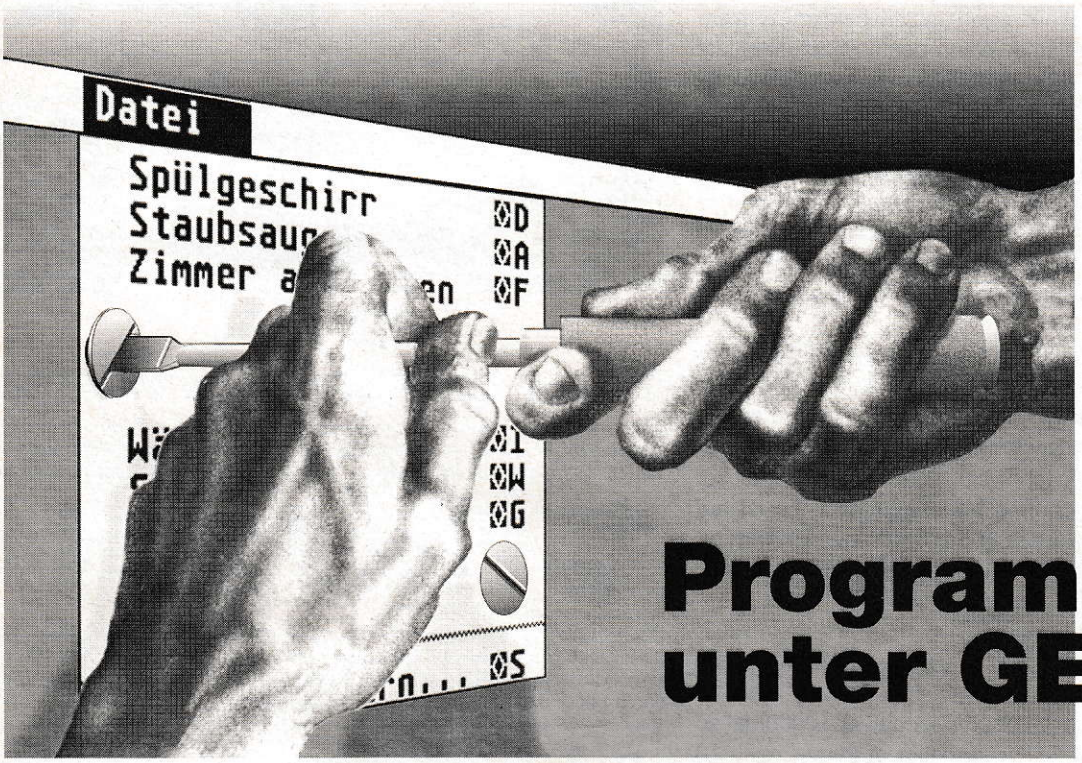
[5] R. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public key cryptosystems, Comunication of the AMC 21/1978, S. 120-126

[6] Eckart Winkler, Dreifach genäht, MC 3/88, S. 76-79

Vertiefende Literatur nach [4]:

[7] Dennig, Cryptography an data security, Addison-Wesley (1982)

[8] Heider, Kraus, Welschenbach, Mathematische Methoden der Kryptoanalyse, DuD Fachbeiträge 8, Vieweg (1985)



Programme unter GEM

Dietmar Rabich

Ein Programm unter der Benutzung von GEM zu schreiben, ist nicht allzu schwer. Der höhere Aufwand wird durch eine leicht zu bedienende Benutzeroberfläche entlohnt. Einige Grundlagen, die die Programmierung erleichtern, sollen hier erörtert werden. Ein wenig Kenntnis der GEM-Routinen und von deren Namensgebung sollte vorhanden sein, da nicht alle Routinen erklärt werden können. Trotzdem dürfte es nicht schwerfallen, den Erläuterungen zu folgen.

GEM steht - wie sicher die meisten wissen - für **G**raphics **E**nvironment **M**anager. Es handelt sich um eine Bedieneroberfläche, die geschaffen wurde, um den Umgang mit Computern zu erleichtern. GEM beinhaltet die beiden Funktionsblöcke AES - **A**pplications **E**nvironment **S**ervices - und VDI - **V**irtual **D**evice **I**nterface -. Auf Wissen um das VDI soll hier nicht vertiefend eingegangen werden, uns liegt hier vielmehr die Gestaltung der Programme am Herzen.

Das AES - eigentlich "die" AES - stellt die für den Bediener wichtigen Elemente wie Fenster, Menüs und Dialoge zur Verfügung. Da das AES Graphik-Routinen benötigt, steht es hierarchisch über dem VDI, d.h. es benötigt das VDI.

Grob verkürzt kann man sagen, daß das AES für das Programmhandling und das VDI für die Ein- und Ausgabe (von Graphik) zuständig ist.

GEM-Programme zeichnen sich dadurch aus, daß sie unabhängig vom Rechner sind. Sie sind also so zu halten, daß sie ohne weiteres auch auf anderen Rechnern lauffähig sind. Lediglich eine Übersetzung (Compilier-Vorgang) sollte erforderlich sein, um das Programm an neue Rechner anzupassen. Eine GEM-Programm-Portierung dürfte also in wenigen Minuten geschehen sein.

Hat man lange auf einem speziellen Rechner nicht-portabel programmiert, könnten zu Beginn einige Schwierigkeiten auftauchen. Aber auf Rechnern wie dem COMMODORE 64 ist eine vom Rechner unabhängige Programmierung nicht vorgesehen, und daher stört es auch nicht sehr, wenn im Speicher herumgewühlt wird! Die Umgewöhnung lohnt sich aber auf jeden Fall! Fänden Sie es denn nicht auch schön, wenn sich alle Programme mit einer gewissen Einheitlichkeit (nicht Eintönigkeit!!) prä-

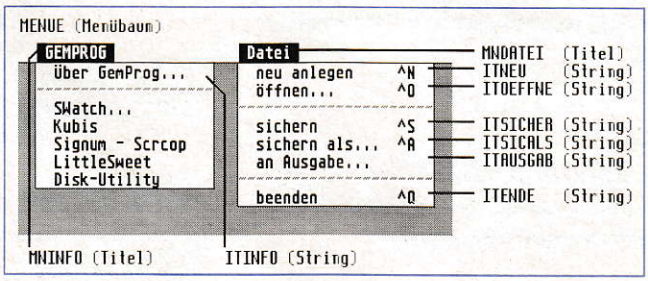


Bild 1: Die zu erstellende Menüleiste

sentierten und sofort bedienen ließen? Ärgert es Sie nicht auch, wenn Sie beispielsweise lange nach einer Möglichkeit suchen müssen, ein Programm beenden zu können?

Zur unabhängigen Programmierung gehört auf dem ST übrigens auch, daß die Line A-Routinen und Funktionen wie *Getrez* nicht benutzt werden. *Getrez* beispielsweise kann bei einem Ganzseiten-Monitor gar keine korrekten Werte liefern! GEM hingegen stellt Funktionen zur Verfügung, die es erlauben, den Arbeitsbereich korrekt zu ermitteln. Generell müssen rechnerspezifische Werte abgefragt und nicht angenommen werden. Es gibt nicht nur Monitore mit einer Breite von 640 und einer Höhe von 400 Pixeln. Auch liegt der Ursprung des Arbeitsbereichs nicht immer bei (0,0). Selbst die Buchstaben haben nicht immer eine Höhe von 16 Pixeln! Das AES und das VDI bieten reichlich Möglichkeiten, benötigte Werte abzufragen.

Aber welche Programmiersprache findet Verwendung? Eine passende Programmiersprache sollte einigen Kriterien genügen. Auf der einen Seite muß sie auch auf anderen Rechnern verfügbar sein, und auf der anderen Seite sollte man modularisieren können. Strukturierte Datentypen sind sehr hilfreich, da diese bei der Programmierung unter GEM immer wieder auftauchen. Ein Compiler darf auch nicht fehlen, da sonst einige Routinen nicht anwendbar sind. Man denke dabei nur an die für Accessories notwendige Funktion *menu_register* oder an die Funktion *appl_find*, die auf den korrekten Namen des Programms angewiesen ist!

Es bieten sich also geradezu C, Modula-2 und Pascal an. Die Priorität liegt jedoch eindeutig bei C, denn diese Programmiersprache findet man am ehesten auch auf anderen Rechnern. Es kommen jedoch nur die Entwicklungssysteme in Frage, die sich an den gegebenen Standard halten. Wenn auf einem anderen Rechner unter einem anderen Betriebssystem erst alle Funktionen umbenannt werden müßten, so wäre die Programmierung nicht mehr rechnerunabhängig. Abstriche sind jedoch bei allen Routinen zu machen, die Betriebssystemfunktionen benötigen, GEM-Routinen gehören allerdings nicht dazu.

Ziel dieses Artikels ist die Entwicklung eines Rahmenprogramms, welches

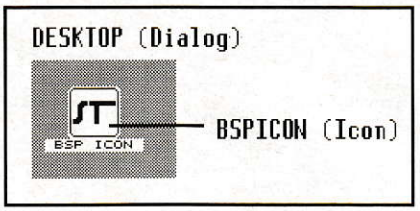


Bild 2: Die Dialogbox DESKTOP mit Icon

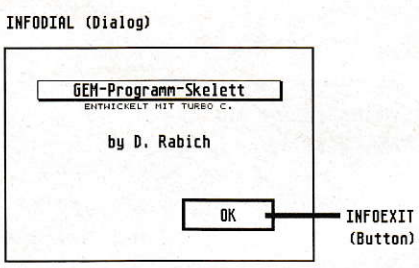


Bild 3: Die Info-Dialogbox

zeigt, wie man ein Menü auf den Bildschirm bringt und verwaltet. Parallel dazu werden Tastendrücke ausgewertet. Auch das Desktop wird gegen ein neues ausgetauscht und das darauf befindliche Icon verwaltet. Da das Programm in C geschrieben und sicher nicht jede Leserin/ jeder Leser dieser Programmiersprache mächtig ist, soll zunächst eine Kurzfassung C-spezifischer Elemente gegeben werden.

C erfüllt alle oben gegebenen Anforderungen. Es handelt sich um ein Entwicklungssystem mit Compiler, die Modularisierung ist erlaubt, auch strukturierte Datentypen existieren.

Zur C-Syntax: Eine Voranstellung von ++ bzw. -- steht für die Inkrementierung bzw. Dekrementierung vor der Verwendung eines Variableninhalts. Folglich bedeutet *++x* nichts anderes als *x=x+1*. Genauer: Steht in C *Proc(++x);*, so bedeutet das beispielsweise in Pascal: *x:=x+1; Proc(x);*. Steht das ++ bzw. -- nach der Variablen, wird diese erst nach der Benutzung verändert, also würde statt *Proc(x); x:=x+1;* in C nur *Proc(x++);* geschrieben.

Die Symbole +, -, * und / stehen wie in anderen Programmiersprachen für die vier Grundrechenarten.

Steht das * direkt vor einem Variablennamen - etwa **a* -, bedeutet das, daß *a* auf einen Wert zeigt. Dies entspricht dem ^ in Pascal. Die Adresse einer Variablen kann man über den Adreß-Operator & ermitteln. Wird der Adreß-Operator zur Übergabe von Parametern an die Routinen benutzt, kann der Inhalt der Variablen von der entsprechenden Routine verändert werden.

Die Operatoren >> und << bedeuten, daß um eine bestimmte Anzahl Bits nach rechts oder links verschoben wird. Eine bitweise Verknüpfung *und* wird mit *&*, eine bitweise Verknüpfung *oder* mit | abgekürzt. Werden die letzten beiden Symbole doppelt geschrieben, wird logisch verknüpft. Ein Vergleich auf Gleichheit erfolgt mit ==.

Die Berechnung *a=a+b* kann mit *a+=b* abgekürzt werden. Analog klappt das natürlich auch mit den anderen Operatoren.

Und schon wär's geschafft. Besonderheiten von C, die benutzt werden, sind aufgezählt. Auch Nicht-C-Programmierer dürften nun mit den Listings keine Schwierigkeiten mehr haben.

Zu einem GEM-Entwicklungssystem gehört für viele auch ein Resource-Construction-Programm, mit dem man Menüs, Dialoge und andere GEM-Elemente kreieren kann. Natürlich ist dieses Programm nicht zwingend notwendig, es erleichtert die Arbeit jedoch ungemein. Außerdem sieht man gleich, wie Dialoge und die anderen Elemente später aussehen werden.

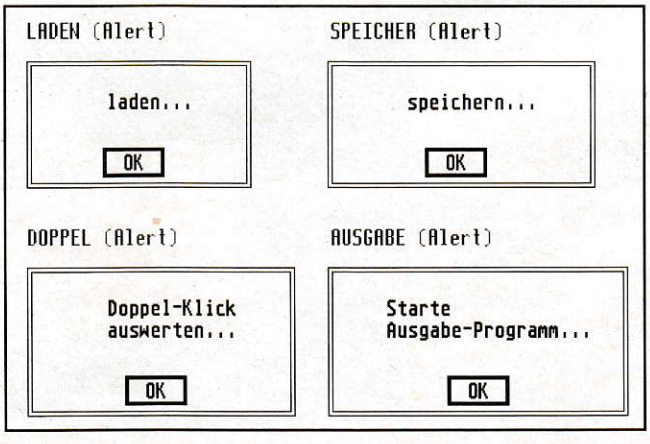


Bild 4: Die Alertboxen des Programms

Bevor nun die Programmierung beginnt, legt man noch einen Plan fest, der beinhaltet, was das Programm können soll und was auf welche Weise realisiert werden kann. Bei der eigentlichen Programmierung ist es sehr empfehlenswert, rechnerspezifische Routinen in einem eigenen Modul unterzubringen. Bei einer Portierung reicht es dann, dieses Modul den neuen Gegebenheiten anzupassen. Auch die GEM-Umgebung trennt man besser von den diversen anderen Routinen. Damit bleibt der Überblick gewahrt!

In einem kurzen Abriß sollen nun noch einige AES-Routinen vorgestellt werden. Logischerweise kann hier nicht alles erklärt werden, da eine ausführliche Erklärung viele, viele Seiten füllen würde. Eine vollständige Auflistung bleibt der Literatur überlassen.

Das AES besteht aus 11 Bibliotheken. Die APPL-Bibliothek enthält die Funktion *appl_init*, die zur Anmeldung eines GEM-Programms benötigt wird. Der zurückgegebene Wert gibt Aufschluß darüber, ob eine Anmeldung erfolgreich war oder nicht (Rückgabe: -1). War die Anmeldung nicht erfolgreich, darf der Rest des Programms nicht ausgeführt werden. Eine Auswertung des Wertes ist demzufolge unabdingbar. (Es gibt übrigens viele Programme, die keine Auswertung vornehmen. Man verbaut sich damit aber die Möglichkeit, daß das Programm unter jeder Bedingung lauffähig ist.) Ferner gehört in diese Bibliothek auch die Funktion *appl_exit*, die zum Abmelden des Programms benötigt wird. Sie darf nur dann ausgeführt werden, wenn *appl_init* erfolgreich war. Die Auswertung des Rückgabewertes scheint zur Zeit auf dem ST wenig Sinn zu haben, aber dies scheint eben nur so. Sobald ein zukünftiges Multitasking-GEM mehrere Programme abarbeiten kann, erhält der Rückgabewert seine Berechtigung.

Zu der zweiten Bibliothek - der EVNT-Bibliothek - gehören Routinen zur Überwachung von Ereignissen. Für jeden Ereignistyp gibt es eine eigene Funktion. Die wichtigste Funktion ist jedoch *evnt_multi*. Sie ermöglicht den Empfang von verschiedenen Ereignissen. Das Beispielprogramm verwendet sie, um auf einen Mausklick, einen Tastendruck bzw. eine Menüauswahl zu warten. Zur genaueren Beschreibung des Mausklicks sind die Bits für die betrachteten Maustasten sowie die Anzahl der abzuwartenden Mausklicks zu setzen. Die Ermittlung der gedrückten Taste erfolgt über einen 16 Bit-Wert. Die oberen 8 Bits enthalten den Scancode - erlauben also die Auswertung der Funktionstasten - und die unteren 8 Bits den ASCII-Code. Wird ein Menüpunkt ausgewählt, erhält das Programm eine Nachricht. Diese beinhaltet Menütitel und -eintrag.

Bild 5: Prinzipieller Ablauf eines GEM-Programms

Die dritte, die sogenannte MENU-Bibliothek enthält Routinen zur Menüverwaltung. Häufig gebraucht wird *menu_bar*, um eine Menüleiste auf den Bildschirm zu bringen und auch wieder zu entfernen, und auch *menu_tnormal*, um einen Menütitel zu invertieren bzw. die Invertierung aufzuheben. Nach jeder Auswahl einer der Optionen in einem Drop-Down-Menü bleibt nämlich der Titel invertiert. Die normale Darstellung wird erst durch Aufruf von *menu_tnormal* wieder erreicht. Benutzt man statt des Menüs die Tastatur, wird der entsprechende Menütitel invertiert, womit wieder die Funktion *menu_tnormal* Verwendung findet. Die Invertierung sollte unbedingt vorgenommen werden, damit der Programmbediener merkt, daß es auch ein Menü gibt, in dem das zum Tastendruck gehörige Kommando zu finden ist. Umgekehrt gehört zu dem passenden Menüeintrag auch ein Tastenkürzel, welches wieder auf die Tastenkombination hinweist.

Die vierte Bibliothek heißt OBJC-Bibliothek und umfaßt Routinen, die die Manipulation von Objekten erlauben. Wichtig sind insbesondere die Routinen *objc_draw*, *objc_find* und *objc_offset*. *Objc_draw* erlaubt die Ausgabe von Objekten, also auch von Dialogboxen, *objc_find* ermöglicht es, an einer bestimmten Position - etwa der Mausposition - ein dazugehöriges Objekt - beispielsweise ein Icon auf dem Desktop - zu ermitteln. *Objc_offset* berechnet die Position eines Objektes. Wer einen Objektbaum mit *objc_add* erweitern möchte, wird auf einige Probleme stoßen, denn der Speicherplatz für die Erweiterung muß vorhanden sein! Also ist das neue Objekt schon bei der Definition des Baums mitaufzunehmen oder aber der Baum an einen neuen Ort mit ausreichendem Speicherplatz zu kopieren.

Bibliothek fünf trägt den Namen FORM-Bibliothek und enthält die zur Verwaltung von Dialogen notwendigen Routinen *form_do*, *form_dial*, *form_center* und *form_alert*. *Form_alert* ist die einfachste darunter. Sie ist dafür veranwortlich, daß sogenannte Alertboxen ausgegeben und verwaltet werden. *form_center* zentriert ein Objekt oder in unserem Fall eine Dialogbox, *form_dial* reserviert den Hintergrund

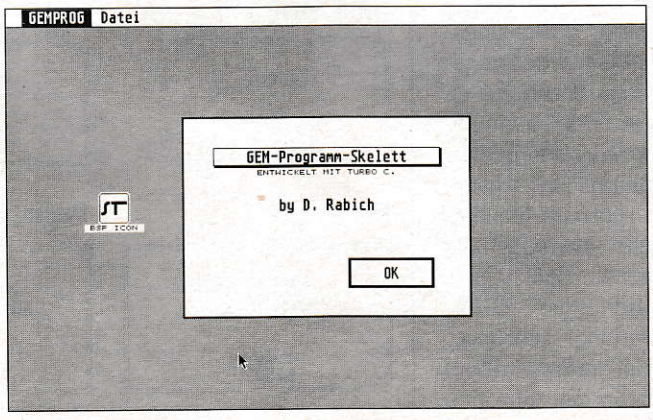


Bild 6: Das Beispielprogramm in Aktion

Wollen Sie Meer ?

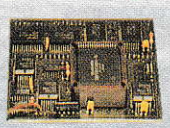
Meer RAM

MICRO RAM

So klein wie eine Streichholzschachtel. Mit Ihr erweitern Sie den RAM jedes ST's auf bis zu 4 MB.

ab DM 249,-

Meer Speed

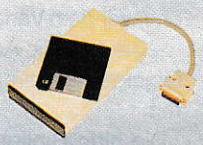


HYPERCACHE

Der neu überarbeitete Cache - Beschleuniger bringt 16 MHz Power für alle ST's. Auch mit NVDI 2.0 erhältlich.

nur DM 298,-

Meer Platz

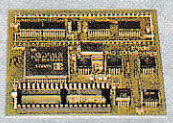


MICRO DRIVE

Festplatten in Floppydrive-Größe. Mit 52, 105 oder 245 MB, 15/17 ms und Cache. Platz auf der Platte und auf dem Tisch.

ab DM 798,-

Meer DOS



PC/AT-SPEED

Emulatoren von Sack für höchste DOS Ansprüche auf dem ST. PC-Speed, AT-Speed, AT-Speed C 16 und

ab DM 149,-

Wir sind Spezialist für Hardwareerweiterungen. Wir beraten Sie, wie Sie Ihren ST erweitern können, auch wenn es **Meer** Erweiterungen gleichzeitig sein sollen. Können Sie den Einbau nicht selbst vornehmen ? Wir bieten **Meer**: Unsere Techniker können Ihren Rechner bei Terminabsprache in wenigen Stunden aufrüsten. Suchen Sie andere Hard- oder Software ? Unser Gesamtangebot bietet natürlich noch **Meer**. Neugierig ? Dann lassen Sie sich eine kostenlose Gesamtpreisliste schicken oder bestellen Sie gleich unseren Katalog 92. Produkte Infos Tips, für 3,80 DM in Briefmarken. **Wollen Sie Meer, dann wollen Sie:**

Meyer & Jacob

Münsterstraße 141 • 4600 Dortmund 1 • Tel.: 0231/ 83 32 05

WOBO-Service

Willi B. Werk

MegaPlot 189,--
Der Werteplotter
****
Signum! 3 ➡ *Superpreis!!!*
Signum! 2 *die A. lohnt!* a.A.
Script I + II *A. lohnt!* a.A.
STAD 1.3+ *A. lohnt!* a.A.
Piccolo *die A. lohnt!* a.A.
SDO PreView, etc. 50,--
alle APi-Soft Prod. lieferb.
QUERDRÜCK2 ➡ 71,--
That's Write 2.x 299,--
CyPress 248,--
CALAMUS 1.09N 398,--
CALAMUS SL 1299,--
Type Art ➡ 545,--
Publ.Part.Mast.V.2. 699,--
TeX 13 Disketten 48,--
(S389-399,432,433)
ClipArt Paket 56,--
(16 PD-Pool Disk.)
GFA-BASIC 3.5 216,--
GFA-BASIC 3.6 TT 259,--
GFA-ASSEMBLER 119,--
ST Pascal Plus 199,--
MAXON PASCAL 216,--
Pure C *die A. lohnt!* a.A.
ACS *neu* ➡ 169,--
K-SPREAD 4 *A. lohnt!* a.A.
TEMPUS V.2.xx 99,--
Anti Viren Kit 3 79,--
Quick ST II 56,--
NVDI 2 83,--
NVDI 2 + Kobold 150,--
XBoot *Vers. 2.5x* 68,--
1st Lock ➡ 179,--
Adimens ST pl. 3.1 ➡ 239,--
1ST Base 219,--
EasyBase *A. lohnt!* a.A.
Phoenix *die A. lohnt!* a.A.
1st Card 278,--
1ST fibuMan 136,--
fibuMAN e/f 309,--/609,--
ARGON *neu* ➡ 89,--
CRYPTON 83,--
Diskus V. 2.x 136,--
MultiGEM 136,--
EASE; MultiDesk je **83,--**
NeoDesk 3 109,--
Harlekin II 136,--
Mortimer/Mort. plus a.A.
CodeKeys 89,--
Kobold 75,--
F-Copy Pro 76,--
BigScreen 2 + SPEX 83,--
MegaPaint II prof. 249,--
Arabesque Pro 298,--
DATA light ➡ 83,--
HASCS II prof. 139,--
TKR-Produkte ➡ a.A.

Btx/Vtx-Manager 4.x 129,--
ReproSt.J.+Scanman 509,--
TOS Extension Card incl. TOS 2.06 198,--
Pixel Wonder 135,--
Channel Videodat De. 369,--
Hardwareprodukte von Hard&Soft; FSE; protar; vortex; etc. **a.A.**
Speichererweit. 4MB 389,--
A.-Switch OverScan 110,--
That's a Mouse ➡ 69,--
Emulatoren:
ATonce-386SX 698,--
AT-Speed C16 445,--
IIT CoProc. für C16 178,--
Supercharger V.1.5 569,--
Unser PD-Angebot:
Wir bieten Ihnen die PD-Disketten aus dem Atari (PD) Journal **(J)**, PD-Pool (2000/5000) **(P)**, ST- Computer **(S)**, ST-Vision **(V)**, die TT-Serie **(T)** und die Demo-Serie **(De)** an. Die Preise (pro Diskette):

1 - 4 DM 5,--
5 - 9 DM 4,50
ab 9 DM 4,--

Bitte beachten Sie, daß wir nur original **Fuji MF2DD** Disketten (keine Bulkware) verwenden. Das alle Kopien nur mit "VERIFY" durchgeführt werden und die PD's auf Viren überprüft sind, ist für uns selbstverständlich.
Ab PD-Pool Disk. **2331** noch höhere Programmqualität; neuer Preis für PD-Pool **(P)** Disk. **(ab 2331) DM 10,--** pro Diskette. Auch für PD-Pool-Disk. **(ab 2331)** wird weiterhin hochwertiges Disketten-Material verwendet. Reine PD-Bestellungen werden bei Vorkasse versandkostenfrei und bei Zahlung per Nachnahme gegen DM 6,-- Nachnahmegebühren verschickt. Für alle anderen Bestellungen gelten die unten aufgeführten Bedingungen.
Zu PD-Versandbedingungen die drei folgenden Artikel:
OXYD Buch + Disk. 50,--
OXYD2 Buch + Disk. 60,--
Spacola Buch + Disk. 55,--

Preise in DM; vorbehaltlich Irrtümer und Preisänderungen. Bei Vorkasse 2% Skonto, zuzügl. DM 5,50 Versandkostenanteil; bei Nachnahme kein Skonto, zuzügl. DM 9,50 Versandkostenanteil. Kein Ladenverkauf!
Selbstabholung nach tel. Absprache möglich!
Dies ist nur ein kleiner Ausschnitt aus unserem Angebot.

Sielwall 87, D-2800 Bremen 1
Tel. 0421/75116; Fax 0421/701285; BTX 042175116

ARTWORKS

Gestaltungen für Ihr CALAMUS®
Bereits fertig gestaltete Geschäftspapierfamilien, Aufkleber, Prospekte und vieles andere mehr vom Profi.

Outline Fonts für Ihr CALAMUS®
in Schneideplotterqualität

CALAMUS® ist eingetragenes Warenzeichen der Firma DMC.

Microfile

Wechselplatte
44 MB, 20 ms Zugriffszeit, 2 Jahre Garantie — 998 DM
44 MB Medium — 178 DM

Festplatten
45 MB, 24 ms Zugriffszeit, 1 Jahr Garantie — 798 DM
105 MB, 17 ms Zugriffszeit, 64 KB Cache — 1248 DM
2 Jahre Garantie

Computertechnik Rosenplänter GmbH
Wagenstieg 5, 3400 Göttingen
Tel: 0551-377021, Fax: 377242

Solange Vorrat. Angebot freibleibend, technische Änderungen vorbehalten

---

Adventures & Simulations

Infocom ab 29.-DM

Beyond Zork	39,-	Seastalker	29,-
Zork I	39,-	Zork II	49,-
Trinity	59,-	Hitchhikers G.t.G	49,-
Stationfall	49,-	Infidel	49,-
Hollywood Hijinx	39,-	Witness	49,-
Ballyhoo	49,-	Spellbreaker	39,-
Moonmist	49,-	Leather G. o. Ph.	49,-
Lurking Horror	49,-	InvisiClues	je 19,-

Infocom ist auch für andere Systeme lieferbar, z.B. MS-DOS, ATARI XL, C64/128.
Fordern Sie unsere speziellen Infocom Infos an.

Level 9: Silicon Dreams (3 adv.) 29,-
Time and Magic 19,- Lancelot 9,-

diverse: Deja Vu 19,- Deja Vu 2 29,-
Chronoquest 2 29,- The Kristal 39,-
Corruption 29,- Monkey Island (engl) 59,-

Sierra: Space Quest 2 49,- Gold Rush 49,-
Conquest of Camelot 59,- Colonels Bequest 59,-
Mixed Up Mother G. 49,- Manhunter S F 59,-
Alle anderen lieferbaren Sierra adventures 99,-

Simulations: Flight Of The Intruder 89,-
Utopia 89,- Lemmings 69,- Powermonger 29,-
R Type2 79,- Logical 59,- Lotus Turbo 2 59,-
Airbus 119,- Wolfpack 59,- Monkey Island 59,-
Elvira 59,- F-15 S.E.2 99,- Railroad Tyc. 89,-
Microprose Golf 79,- Silent Service 2 79,-
Starglider 2 (monochrom lauffähig) 29,-

Liste gratis. Versand Vorkasse 5,- Nachnahme 7,-

A+S, U. Wandrer, Postf. 4
W-3061 Lindhorst ☎ 05725/5426

(zum Beispiel die sogenannte Rechteckliste für die Fenster benötigt diese Reservierung) und gibt ihn wieder frei. Restauriert wird der Hintergrund jedoch nicht automatisch! Für das Hintergrundfenster mit der Kennung 0 wird dem Programm die Arbeit abgenommen, aber nicht für die anderen Fenster. Aber keine Sorge: Das Programm erhält rechtzeitig eine Meldung über die EVNT-Funktion *evnt_mesag* bzw. *evnt_multi*. *Form_dial* mit dem Parameter FMD_FINISH kann übrigens auch dazu benutzt werden, um anderen Programmen eine "Nachricht" zu schicken, daß der Bildschirm zu erneuern ist.

Die GRAF-Bibliothek - die sechste - umfaßt Routinen, die etwa die bewegenden Rechtecke liefern. In dem Beispielprogramm findet graf_dragbox Verwendung, um dem Benutzer die Verschiebung von Icons zu ermöglichen und vor allem optisch nahezulegen. Außerdem sind hier auch die Routinen enthalten, mit denen man die Mausform ändern und den Status und die Position der Maustasten abfragen kann.

Die SCRP-Bibliothek trägt die Nummer sieben und beinhaltet die beiden Routinen *scrp_read* und *scrp_write*, die den Pfadnamen für das Scrap-Directory lesen bzw. setzen.

Mit *fsel_input* aus der Bibliothek FSEL - Nummer 8 - wird der bekannte File-Selector aufgerufen. Übergeben werden ein Pfad-beispielsweise "E:\ORDNER*.TXT" und ein Dateiname "DATEI.TXT". Zurückgegeben werden der neue Pfad und der neue Dateiname.

Die umfangreiche neunte Bibliothek WIND umschließt alle Routinen zur Verwaltung von Fenstern. Die Routine *wind_get* wird zusammen mit dem Parameter *WF_WORKYXWH* benutzt, um die Ausmaße des Hintergrundfensters zu ermitteln, und *wind_set* in Verbindung mit *WF_NEWDESK*, um den neuen Desktop zu setzen. Ferner sperrt *wind_update* den Bildschirmaufbau störende Aktionen.

Die Abfrage der Rechteckliste eines Fenster - die Teile eines Fensters, die sichtbar sind - geschieht ebenso mit wind_get. Übergibt man den Parameter *WF_FIRSTXYWH*, erhält man das erste Rechteck dieser Liste und mit *WF_NEXTXYWH* die weiteren. Das Ende der Liste ist durch ein 0 Pixel breites und hohes Rechteck gekennzeichnet.

Mit *rsrc_load* aus der RSRC-Bibliothek, sie ist die zehnte, wird ein mit einem Resource-Construction-Programm angefertigtes Resource-File geladen. *rsrc_free* gibt den dafür benutzten Speicher vor dem Verlassen des Programms wieder frei. Die Adressen der einzelnen Objekte werden mit *rsrc_gaddr* ermittelt.

Die elfte Bibliothek - SHEL - enthält eine Routine, mit der man andere Applikationen starten kann, sowie weitere passende Funktionen. Die GEMDOS-Funktion *Pexec* wird dabei nicht benutzt. *Shel_write* teilt dem AES mit, daß nach Beendigung des laufenden Programms ein anderes zu starten ist. Damit ist es möglich, zwischen diversen Programmen zu wechseln, ohne daß dabei mehrere Programme im Speicher gehalten werden müssen.

Bild 7: Eine der Alertboxen

Einen Überblick über die AES-Routinen hätten wir uns jetzt verschafft. Nun kann das Programmieren endlich losgehen. Oder doch nicht?

Erst müssen wir uns noch ein Resource-File verschaffen, welches wenigstens einige notwendig Dinge enthält. Bild 1 zeigt das Menü, Bild 2 das Desktop, Bild 3 die Dialogbox und Bild 4 die Alertboxen unseres Beispielprogramms. Richten Sie sich bei dem Entwurf bitte nach Ihrem persönlichen Resource-Construction-Programm. Die Namen der Elemente entnehmen Sie bitte den Bildern. Achten Sie aber bei der Gestaltung immer auf die bestehenden Konventionen. Zu den Konventionen aber gleich mehr.

Ein Resource-Construction-Programm ermöglicht die Gestaltung von Menüs, Dialogboxen, Alertboxen, freien Strings und Icons. Freie Strings und Icons sind für das Beispielprogramm unerheblich und sollen daher nicht weiter betrachtet werden. Ebenso sind Alertboxen sehr einfach zu erstellen und damit recht unwichtig. Menüs und Dialogboxen haben eines gemeinsam: Es sind einfache Objektbäume. Ist Ihnen etwas aufgefallen? Das Desktop für das Beispielprogramm ist nicht aufgeführt! Es wird jedoch auch mit dem Resource-Construction-Programm entworfen. Das Desktop ist "nur" eine Dialogbox. Exit-Buttons existieren natürlich nicht. Sie nehmen eine Dialogbox ohne Rahmen und plazieren darauf die zukünftigen Objekte. Die Größe muß nicht mit der Bildschirmgröße übereinstimmen. Es würde auch keinen Sinn machen, die Bildschirmgröße zu berücksichtigen, da diese nicht bekannt sein kann. Das Desktop wird nach dem Laden der Resource-Datei den örtlichen Verhältnissen angepaßt.

Wenn Sie das Resource-File haben, kann die Arbeit beginnen. Zuerst erstellen wir ein Modul mit den systemspezifischen Routinen. In unserem Beispiel handelt es sich lediglich um eine Routine, die das Bootlaufwerk ermittelt. Die Kenntnis über das Bootlaufwerk wird für den Scrap-Pfad benötigt.

Nun noch ein wenig zu den Konventionen. Im Menü gehören hinter jeden Eintrag, der eine Dialogbox oder ein Fenster zur Folge hat, drei Punkte, das Tastenkürzel gehört an den rechten Rand des Eintrags. Im sogenannten Desk-Menü findet man einen Eintrag, mit dem man eine Programminformation abrufen kann. GEM 2.0 setzt statt des Menütitels "Desk" automatisch den Programmnamen ein. Es bietet sich also geradezu an, dies unter dem aktuellen ST-GEM nachzuahmen und den Programmnamen anstelle von "Desk" einzusetzen. Allerdings muß dies schon bei dem Entwurf des Menüs geschehen. Unter der aktuellen GEM-Version auf dem ST findet man im zweiten Menü von links die Dateioperationen und auch den Eintrag, mit dem das Programm beendet werden kann.

Menüeinträge, die nicht selektiert werden können oder dürfen, stellt man gesperrt dar. Beispielsweise hat das Speichern eines Textes wenig Sinn, wenn gar kein Textfenster geöffnet ist. Auch

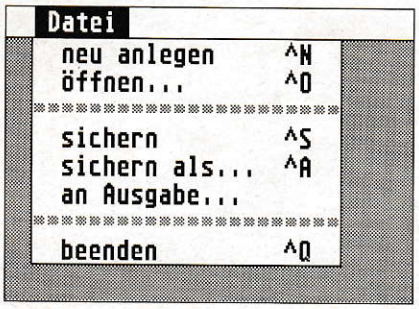


Bild 8: Ein heruntergeklapptes Menü

hat es wenig Sinn, extra für Accessories ein anderes Menü auf den Bildschirm zu bringen, da Accessories ein Zubehör zum laufenden Programm und nicht getrennt laufende Applikationen sind. Achten Sie auch immer darauf, daß Accessories wie zum Beispiel das Kontrollfeld Fenster öffnen können und daher verschiebbar sein müssen. Nehmen wir hierzu ein (Negativ-)Beispiel. Einige Programme lassen zu, daß Accessories aufgerufen werden. Für GEM-Programme ist das natürlich selbstverständlich. Ruft man nun das Kontrollfeld auf und verschiebt es, wird von dem Programm der komplette Bildschirm neu gezeichnet. Das Kontrollfeld ist zwar nach wie vor geöffnet, aber zu sehen ist es nicht mehr.

Vom Benutzer einstellbare Werte wie Icon-Positionen werden in einer Datei mit der Extension INF abgespeichert. Der Name dieser Datei ist mit dem des Programms identisch. In unserem Fall heißt das Programm GEMPROG und damit die Programmdatei GEMPROG.PRG, die Parameterdatei hieße dann GEMPROG.INF. Die Resourcedatei trägt übrigens immer die Extension RSC, also heißt sie hier GEMPROG.RSC.

In den Dialogboxen sollten die Buttons oder Felder zum Verlassen der Dialogbox immer an ähnlichen Stellen - beispielsweise nur am unteren Rand - zu finden sein und sich klar vom restlichen Inhalt abheben. Es ist sehr lästig, wenn man erst verschiedene Elemente anklicken muß, um einen Dialog zu beenden.

Die Mausform ist in der Regel ein Zeiger. Für den Fall, daß das Programm etwas mehr Zeit benötigt, ist die Form einer Biene zu wählen. Wird ein Objekt bewegt, erscheint die Maus (nicht der kleine graue Kasten!) in Form einer flachen Hand. Um eine Sache zu dimensionieren - beispielsweise ein Fenster - , wird die Hand mit dem Zeigefinger benutzt, und bei Texteingaben findet der Balken Verwendung.

Generell ist innerhalb des Programms auf innere Konsistenz zu achten. Es dürfen nicht ähnliche Dinge auf verschiedene Arten bedient werden können. Damit ist natürlich nicht gemeint, daß nicht parallel zum Menü auch die Tastatur bedient werden darf, sondern vielmehr, daß beispielsweise Buttons in Dialogboxen mal oben und mal unten plaziert werden.

Bei der Programmierung sollte ferner noch auf ein paar andere Dinge geachtet werden. Ein gelegentlicher *evnt_timer*-Aufruf bei längeren Operationen gewährleistet, daß auch andere Applikationen weiterarbeiten können. Der Dreifachklick für diverse Aktionen ist ein Kunststück für den Zeigefinger, also bleibt man beim Einfach- oder Doppelklick. Ein Icon für die Scrap-Ablage sieht gut aus und hat seine Berechtigung. Es kann Information darüber geben, ob Dateien in der Scrap-Ablage bereitliegen. Bei der Textausgabe über die VDI-Funktionen erreicht man einen schnelleren Bildschirmaufbau, wenn der Text auf einer Byte- oder noch besser einer Word-Grenze beginnt. Der Wiederaufbau eines Textfensters läuft schneller ab, wenn kein Clipping durchgeführt werden muß. Stattdessen berechnet man den Platz für den Text und gibt nur entsprechend viel aus. Icons lassen sich frei auf dem Desktop plazieren. Für den Benutzer ist es jedoch praktischer, wenn sie sich nur innerhalb eines Raster positionieren lassen. Beispielsweise könnten die Icon-Positionen auf Byte-Grenze festgelegt werden.

Ist ein Programm fertig, beginnt die Testphase. Zu einem umfangreicheren Test gehört auch ein Probelauf auf einem Farb- und einem Monochrommonitor. Auch die Darstellung auf einem Ganzseitenmonitor durch Grafikkarten sollte dazugehören.

Stellen Sie sich einen Fragenkatalog zum Testen: Laufen Accessories - auch mit Fenstern - einwandfrei? Kommt es nicht zu Abstürzen unter anderen Hardware-Bedingungen? Und so fort. Testen Sie Programme nicht alleine, lassen Sie auch Laien an das Programm! Soviel zum Testen.

Das Beispielprogramm zeigt uns, wie man sich ein Grundgerüst für ein GEM-Programm verschaffen kann. Das Rahmenprogramm ist in der Lage, einen eigenen Desktop (mit einem Icon) anzumelden, ein Menü darzustellen und natürlich auch, den Desktop und das Menü zu verwalten. Unser Desktop beinhaltet lediglich ein Icon, damit das Prinzip der Programmierung klar wird. Möchte man jedoch mehrere Icons haben, so müssen diese entweder vorher (beim Entwurf mit dem Resource-Construction-Programm) deklariert werden oder der Objekt-Baum (ein Array von Objekten, die miteinander verknüpft sind) muß erweitert werden. Die Erweiterung scheitert jedoch daran, daß ein Feld nicht so einfach vergrößert werden kann. Also muß der gesamte Objekt-Baum (das Array, nicht jedoch Texte etc.) an eine andere Stelle kopiert werden, die genügend Platz bietet.

Sollen eigene Dinge auf das Desktop gebracht werden, so darf dies niemals direkt geschehen. Werden Ausgaben direkt auf das Desktop gemacht und wird danach darüber ein Fenster gebracht, so ist die Ausgabe verschwunden, wenn das Fenster geschlossen wird. Für die eigenen Ausgaben findet die *Application-Block-Struktur* (USER-

```
➡ APPL-Funktionen
      appl_init
      appl_exit
➡ EVNT-Funktionen
      evnt_multi
➡ MENU-Funktionen
      menu_bar
      menu_tnormal
➡ OBJC-Funktionen
      objc_draw
      objc_find
      objc_offset
➡ SCRP-Funktionen
      scrp_read
      scrp_write
➡ GRAF-Funktionen
      graf_dragbox
      graf_mouse
      graf_mkstate
➡ RSRC-Funktionen
      rsrc_load
      rsrc_free
      rsrc_gaddr (R_TREE,R_STRING)
➡ FORM-Funktionen
      form_do
      form_dial (FMD_START,FMD_GROW,FMD_SHRINK,FMD_FINISH)
      form_alert
      form_center
➡ WIND-Funktionen
      wind_get (WF_WORKXYWH,WF_FIRSTXYWH,WF_NEXTXYWH)
      wind_set (WF_NEWDESK)
      wind_update (END_UPDATE,BEG_UPDATE,END_MCTRL,BEG_MCTRL)
```

Bild 9: Im Beispielprogramm benutzte AES-Funktionen

BLK) Verwendung. Somit erhält das Programm immer dann eine Meldung (über USERBLK), wenn das Desktop zu erneuern ist. Das Menü sollte immer die Menüeinträge sperren, die nicht anwählbar sein sollen. Dies geschieht mit *menu_ieanble*, indem die Adresse des Menüs, die Nummer des Eintrags und eine 0 (für desaktivieren) oder eine 1 (für aktivieren) übergeben wird. Das Beispielprogramm macht das nicht, da gar keine Lade- oder Speicher-Routinen implementiert wurden. Das Tastenkürzel steht jeweils am rechten Rand des Menüeintrags. Das entsprechende Tastaturereignis wird von dem Beispielprogramm ausgewertet. Soll das Programm noch unabhängiger gestaltet werden, so besteht die Möglichkeit, den Text des Menüeintrags nach Programmstart zu untersuchen. Dazu wird der Menü-Objektbaum nach den betreffenden Einträgen abgesucht und das Tastenkürzel ausgewertet. Somit wird auch bei einer Veränderung des Resource-Files der Tastaturaufruf geändert. Das Bild 5 zeigt das Beispiel-Programm in Aktion. Eine Alertbox ist auf Bild 6 zu sehen und auf Bild 7 eines der beiden Drop-Down-Menüs.

Wenden wir uns nun dem Programm zu.

Im Listing befinden sich zahlreiche Erklärungen zu den einzelnen Schritten, weshalb diese hier nicht ausführlich erklärt werden sollen. Das Hauptprogramm *main* ist für den groben Rahmen zuständig. Dazu gehört die Initialisierung *init_prg* und das Abmelden *exit_prg*. In der while-Schleife wird solange gewartet, bis die Variable *finish* den Wert TRUE erhält. Für den Fall, daß ein Menüeintrag angeklickt wurde, wird die Routine *hdle_menu* aufgerufen, bei einem Mausklick *hdle_mouse* und bei einem Tastaturereignis *hdle_key*. Die Initialisierungsroutine *init_prg* meldet das Programm zuerst beim AES an, stellt dann die Maus als Biene dar, weil die folgenden Aktionen doch etwas mehr Zeit erfordern. Dann wird das Resource-File geladen und ausgewertet. Direkt nach der Auswertung wird das Desktop in der Größe angepaßt und ausgegeben. Dann wird der Scrap-Pfad gesetzt, falls er noch nicht existiert. Schließlich erfolgt die Ausgabe des Menüs und die Maus wird wieder ein Pfeil. Eine Parameterdatei könnte in dieser Routine auch eingelesen und ausgewertet werden. Die Routine *exit_prg* ist vergleichsweise kurz. Sie meldet das Menü und das Desktop ab, gibt den Resource-Speicher wieder frei und meldet das Programm ab.

Hdle_menu ist wohl die einfachste unter den Handle-Routinen. Entsprechend dem Menüeintrag wird eine Aktion durchgeführt und schließlich der Menütitel wieder normal dargestellt. Die normale Darstellung ist übrigens spätestens dann zu wählen, wenn das Menü wieder verfügbar ist. Der Menütitel sollte solange invertiert bleiben, wie die "blockierende" Aktion andauert. Wird also beispielsweise eine Datei geladen, so hebt man die Invertierung auf, sobald diese geladen ist. Von *hdle_menu* aus wird auch die Informationsdialogbox über *do_info* aufgerufen. In *do_info* wird ein vollständiger Dialog durchgeführt.

Ruft ein Programm mehrere Dialogboxen auf, so empfiehlt es sich, den Vor- und den Nachbereitungsteil in getrennten Funktionen zu deklarieren, denn diese unterscheiden sich in der Regel nicht. Der Durchführungsteil kann sich erheblich unterscheiden, beispielsweise können wie bei einem Fileselector Dateinamen gescrollt werden. Die dritte Handle-Routine - *hdle_key* - ist schon etwas komplizierter. Dort werden nicht nur die oberen 8 Bit des Tastencodes ausgewertet, sondern es wird auch entsprechend der gedrückten Taste der Menütitel invertiert und dann wieder normal dargestellt.

Die Handle-Routine *hdle_mouse* ermittelt zuerst das Objekt, welches sich an der Mausposition befindet. Dann wird zwischen einem Doppel- und einem Einfachklick unterschieden. Bei einem Doppelklick wird das entsprechende Icon invertiert, eine Alertbox ausgegeben und danach die Invertierung wieder aufgehoben. Wurde nur einmal geklickt, ist noch zu unterscheiden, ob die Maustaste mittlerweile losgelassen wurde oder nicht. Ist die linke Maustaste noch gedrückt, wird das Icon bewegt. Ist sie nicht gedrückt, wird das Icon invertiert, wenn sich der Mauszeiger darüber befindet, bzw. die Invertierung aufgehoben, wenn außerhalb des Icons geklickt wurde.

In der Maus-Handling-Routine werden zwei neue Routinen aufgerufen. Dies ist zum einen eine Ausgabe-Routine und zum anderen eine Routine, die die Icon-Bewegung übernimmt. Die Ausgabe-Routine *draw_objc* gibt ein Objekt unter Berücksichtigung der Rechteckliste aus. Dies ist notwendig, da nicht immer ein komplettes Objekt neu gezeichnet werden muß. Denken Sie nur daran, wenn ein Fenster ein Icon halb überdeckt!

Move_objc versteckt zuerst das Objekt und gibt es über *draw2_objc* aus. Die zweite Ausgabe-Routine ist erforderlich, da mit HIDETREE versteckte Objekte nicht direkt angesprochen werden können. Dann wird der Arbeitsbereich - der Bereich, in dem das Objekt bewegt werden darf - ermittelt, die Maus als flache Hand dargestellt und das Objekt via *graf_dragbox* verschoben. Anschließend stehen die neuen Koordinaten des Icons fest. Wenn man nur bestimmte Positionen zuläßt, so ist an dieser Stelle eine Anpassung - beispielsweise auf Byte-Grenze - vorzunehmen. Letztendlich wird die Maus wieder als Pfeil dargestellt und das Icon ausgegeben. Damit seien die Erklärungen beendet. Im Listing finden sich noch weitere Erklärungen und Anregungen. Ich hoffe, daß Ihnen die kleine Einführung in die Benutzung des GEM gefallen hat und sie als nächstes Programm ein wunderschönes GEM-Programm schreiben werden.

Literatur:

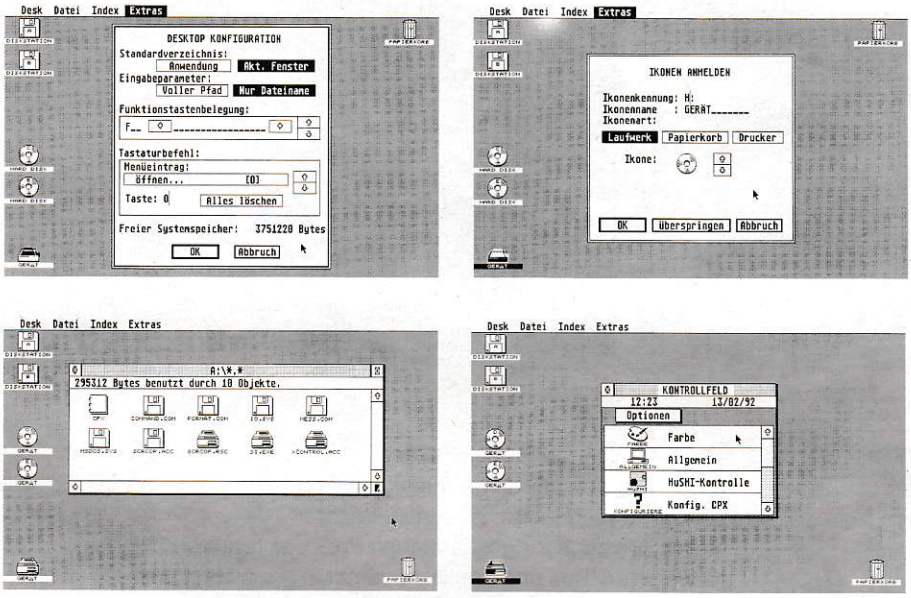
[1] Atari ST Profibuch, H.-D. Jankowski/ J. F. Reschke/ D. Rabich, Sybex

[2] GEM Programmier-Handbuch, P. Balma/ W. Fitler, Sybex

[3] Professionel GEM, T. Oren, ANTIC

"TOS EXPANSIONS CARD"

TOS 2.06 für alle Atari ST Computer



TOS Expansions Card
ist eine Eigenentwicklung aus dem Hause Hard & Soft, die es ermöglicht, in jeden Atari ST Computer das Betriebssystem TOS 2.06 einzubauen.

Die Idee für die Karte entstand dadurch, daß uns keiner der bisher angebotenen kommerziellen Lösungsansätze, sowie Lösungsvorschläge aus Zeitschriften, überzeugen konnte.

Wir haben daher eine Erweiterungskarte entwickelt, welche alle bisher angebotenen Hardwarelösungen in den Schatten stellt.

Die TOS Expansions Card ist auch als Leerkarte ohne TOS 2.06 erhältlich.

Leistungsdaten

- besonders einfacher und platzsparender Einbau
- Umschaltmöglichkeit zwischen bestehendem Betriebssystem und TOS 2.06, auch bei Rechnern mit 6 Betriebsbausteinen ohne Leiterbahnen zu durchtrennen, über den mitgelieferten Schalter
- keine unprofessionellen Flachbandverbindungen, dadurch störungsunanfällig
- Platine sitzt fest im Rahmen
- wird einfach auf die CPU aufgelötet. Wenn sich bereits ein MS DOS Emulator oder eine 16 MHz Erweiterung auf der CPU befinden, wird die TOS EXPANSIONS CARD einfach dazwischen gesteckt, was aufgrund der geringen Bauhöhe von ca. 6 mm ganz einfach möglich ist
- HD-Interface werden unterstützt
- HD-Formatierroutine im Desktop vorhanden

Preise

TOS EXPANSIONS CARD
incl. orig. TOS 2.06

DM 198,—

dto. ohne TOS 2.06

DM 79,—

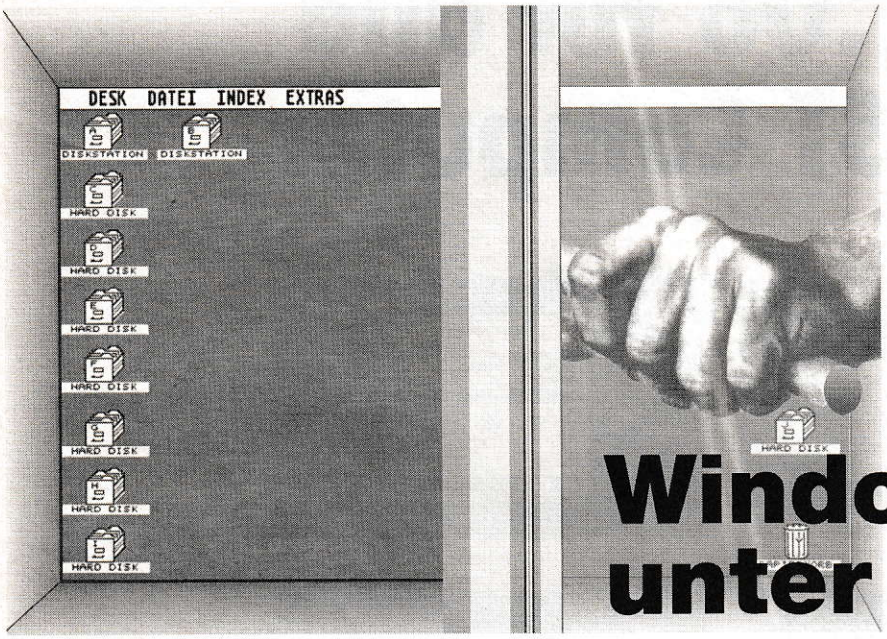
Hard&Soft

Hard & Soft A. Herberg
Obere Münsterstraße 33-35

4260 Castrop-Rauxel

Tel. 02305/18014
02305/18015

Fax 02305/32463



Windows unter GEM

Andreas Lötscher/tw

Eines der wichtigsten Werkzeuge, die eine grafische Benutzeroberfläche wie GEM dem Benutzer zur Verfügung stellt, ist eine Fensterverwaltung. Sie ermöglicht dem Anwender, gleichzeitig mehrere Dinge auf einem einzigen Bildschirm zu tun - so zum Beispiel drei verschiedene Texte zur selben Zeit zu schreiben. Für diese Freiheit muß aber (wie üblich) der Programmierer zahlen. Er sieht sich gezwungen, sein Programm auf eine wesentlich komplexere Art mit der Umgebung kommunizieren zu lassen.

Der folgende Artikel soll dazu dienen, sowohl dem Einsteiger die grundlegenden Routinen vorzustellen, als auch fortgeschrittenen Profis wertvolle Tips und Tricks zu vermitteln. Nach Lektüre des Artikels werden Sie wissen, wie man einen komfortablen und flexiblen Window-Manager erzeugen kann, der das leidige Programmieren von Windows in eigenen Programmen vereinfacht (auf den Disketten zum Heft befinden sich die Routinen zum Window-Manager).

Vorarbeit

Bevor wir in unserem Programm auf dem Bildschirm ein wunderschönes Window erblicken, müssen wir einiges am Vorarbeit leisten. Nach der Anmeldung beim AES (Application Environment System des GEM) und GEM [siehe *gem_init()* im Window-Manager] wird mit *wind_create()* ein Fenster vereinbart:

```
w_handle =
        wind_create()(kind, wx, wy, ww, wh);
```

Als Parameter benötigt die Routine die x-/y-Koordinaten,die Weite (*ww*; Breite) und Höhe (*wh*) des Fensters in der maximalen Größe (meistens ganzer Desktop ohne Menüleiste, weiteres siehe unten).

Außerdem die Komponenten des Fensters (*kind*), die vorhanden sein sollen. Sie umfassen: Titelbalken, Infozeile, Schließecke, Volle-Größe-Box, Bewegungs-Box, Pfeil nach oben, Pfeil nach unten, vertikaler Schieber, Pfeil nach links, Pfeil nach rechts, horizontaler Schieber (vgl. Tabelle 1 und Bild 1). Der Parameter *kind* besteht aus einer ODER-Verknüpfung aller anzuzeigenden Fen-

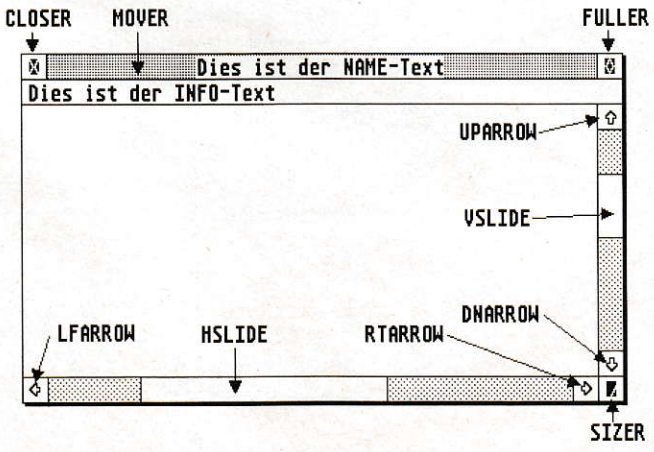


Bild 1: Die Elemente eines Fensters

sterelemente. Wenn ein Element nicht angegeben wird, kann das Programm auch keine Meldung über die Anwahl dieses Elementes bekommen, da es ja nicht auf dem Bildschirm erscheint. Wenn also z.B. der Verschiebebalken nicht genannt wird, kann der Benutzer das Fenster nicht verschieben, und das Programm erhält nie eine Meldung vom Event-Manager (genaueres folgt später).

Als Rückgabewert erhält man die Identifikationsnummer des Fensters (*w_handle*), über die wir im folgenden das Fenster ansprechen können. Sie wird vom AES vergeben, um mehrere Fenster auf dem Bildschirm unterscheiden zu können.

Nachdem GEM nun weiß, wie unser Fenster auszusehen hat und welche Maximalgröße es besitzt, können wir es mit dem folgenden Befehl auf dem Bildschirm darstellen:

```
error =
    wind_open (w_handle, wx, wy, ww, wh);
```

Eingabeparameter sind die oben vom AES vergebene Fensternummer (*w_handle*) sowie die Fenstergröße (*wx*, *wy*, *ww*, *wh*), in der es auf dem Bildschirm dargestellt werden soll.

Der Rückgabewert *error* ist gleich 0, wenn ein Fehler aufgetreten ist.

Haben Sie bei der Erzeugung eines Fensters durch *wind_create()* angegeben, daß das Fenster eine Titel- oder Infozeile oder einen Schieber (Slider) hat, müssen Sie diese vor dem Aufruf von *wind_open()* setzen, da sonst unvorhergesehene Wirkungen auftreten können. Dazu dient die Routine *wind_set()*:

NAME	0x0001
CLOSER	0x0002
FULLER	0x0004
MOVER	0x0008
INFO	0x0010
SIZER	0x0020
UPARROW	0x0040
DNARROW	0x0080
VSLIDE	0x0100
LFARROW	0x0200
RTARROW	0x0400
HSLIDE	0x0800

Tabelle 1: Namen und (hexadezimale) Werte der Fensterelemente

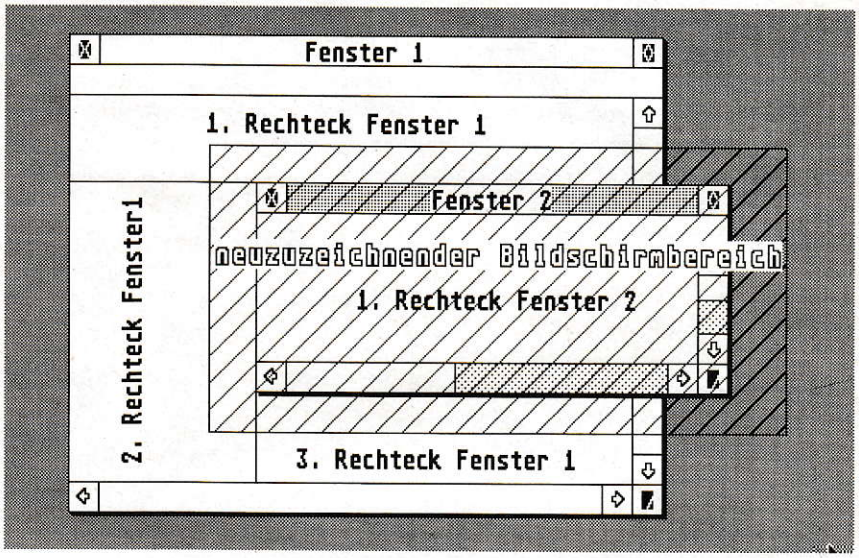


Bild 3: Rechteckliste von zwei Fenstern und neuzuzeichnender Bereich bei Überlagerung durch eine Dialogbox

```
error = wind_set(w_handle, w_field, w1,
                 w2, w3, w4);
```

Wie bei allen Routinen, die Fenster bearbeiten, so muß auch hier die Fensternummer (*w_handle*) angegeben werden. Der nächste Parameter (*w_field*) legt fest, welche Fensterelemente verändert werden sollen. Die letzten vier Parameter variieren, je nachdem welche Funktion ausgewählt wurde [siehe Tabelle 2; nur die mit 'set' gekennzeichneten Funktionen können mit *wind_set()* verwendet werden]. Auch hier ist der Rückgabewert (*error*) 0 bei einem Fehler.

Um den Fensternamen zu setzen, verwenden wir *WF_NAME* und geben in w1 und w2 die oberen bzw. die unteren 16 Bits der Adresse an, an der sich unser Namens-String befindet (in Pure C genügt die Angabe der Adresse; sie muß nicht in einen Low- und High-Teil getrennt werden). Der Aufruf sieht dann folgendermaßen aus:

```
error = wind_set(w_handle, WF_NAME,
                 name, 0, 0);
```

Endlich steht unser Fenster auf dem Bildschirm, doch leider noch nicht ganz perfekt, denn nur der Rahmen wurde gezeichnet, während die sogenannte Arbeitsfläche (Inneres des Fenster) nicht berührt wurde. Wir sollten diese also noch weiß (oder mit anderem Inhalt) füllen. Um die Koordinaten der Arbeitsfläche zu ermitteln, verwenden wir *wind_get()*:

```
error = wind_get(w_handle, w_field, w1, w2,
                 w3, w4);
```

Diese Prozedur ist *wind_set()* ähnlich, verändert aber keine Window-Daten, sondern liefert uns die aktuellen Werte (siehe Tabelle 2). Für die Koordinaten des Arbeitsbereiches setzen wir *w_field* auf *WF_WORKXYWH* und erhalten in *w1*, *w2*, *w3*, *w4* die gesuchten Werte (x/y-Koordinaten sowie Breite und Höhe):

```
error=wind_get( w_handle,
                WF_WORKXYWH, x, y, w, h);
```

Um innerhalb des Fensters etwas ausgeben zu können (Text oder Grafik), müssen wir dafür sorgen, daß während der Ausgabe keine anderen Routinen innerhalb dieses Bereich tätig werden. Dazu schalten wir am Besten erst einmal den Grafik-Cursor ab:

```
v_hide_c(handle);
```

handle ist hier übrigens nicht die Fensternummer, sondern die Nummer der physikalischen Workstation, die innerhalb der *gem_init()*-Routine vom Aufruf *graf_handle()* geliefert wird.

Als zweites setzen wir die Farbe für den Fenster(innen)hintergrund:

```
vsf_color(handle, color);
```

handle ist wieder die Nummer der physikalischen Workstation. *color* ist der Farbindex (0 und 1 sind immer Verfügbar, weitere hängen vom Ausgabegerät ab).

Außerdem müssen wir den Bereich, innerhalb dessen wir etwas ausgaben, auf das Fensterinnere beschränken. Die Koordinaten dazu haben wir mittels *wind_get()* weiter oben bereits ermittelt; allerdings benötigen wir zwei x/y-Koordinaten, so daß wir noch eine kurze Umrechnung vornehmen müssen:

```
x2 = x + w - 1;
y2 = y + b - 1;
```

Nun verwenden wir für die beiden x/y-Paare ein Array und können mit der *vs_clip()*-Funktion die Ausgabe auf den Festerbereich beschränken (clippen):

```
vs_clip(handle, 1, xyarray);
```

Mit der *v_bar()*-Routine zeichnen wir nun ein weißes Rechteck:

```
v_bar(handle, xyarray);
```

Nach dem Zeichnen sollte man nicht vergessen, das Clipping auszuschalten:

```
vs_clip(handle, 0, xyarray);
```

und den Grafik-Cursor wieder einzuschalten:

```
v_show_c(handle);
```

Nun haben wir ein Fenster mit weißem Inhalt und können in ihm weitere Ausgaben tätigen. Zum Beispiel einen Text mittels der Funktion *v_justified()* ausgeben oder grafische Elemente zeichnen. Ihrer Fantasie bzw. Ihren Anwendungen sind keine Grenzen gesetzt. Alle dazu notwendigen Routinen finden Sie innerhalb des VDI. Beachten Sie, daß Sie, soweit dies nicht bereits durch Routinen des Window-Managers geschehen ist, bei Ausgaben das Clipping einschalten sollten.

Programmende

Am Ende eines GEM-Programms müssen alle geöffneten Fenster wieder geschlossen werden. Dies erledigt, im Gegensatz zum Öffnen, eine einzige Routine (jeweils für jedes geöffnete Fenster einzeln):

```
error = wind_close(w_handle);
```

Hierbei ist dann die Fensternummer wieder der Eingangsparameter, und ein

WF_KIND	1	set	
WF_NAME	2	set	Fenstername setzen (Zeiger auf Text in w1, w2)
WF_INFO	3	set	Info-Zeile setzen(Zeiger auf Text in w1, w2)
WF_WORKXYWH	4	get	Arbeitsbereich des Fensters (w1: x-, w2: y-Position, w3: Breite, w4: Höhe)
WF_CURRXYWH	5	set/get	Fenstergröße (w1: x-, w2: y-Position, w3: Breite, w4: Höhe)
WF_PREVXYWH	6	get	Größe des vorherigen Fensters (w1: x-, w2: y-Position, w3: Breite, w4: Höhe)
WF_FULLXYWH	7	get	Fenstergröße in größtmöglicher Ausdehnung (w1: x-, w2: y-Position, w3: Breite, d: Höhe)
WF_HSLIDE	8	set/get	Position für horizontalen Schieber (w1: 0: links, 1000: rechts)
WF_VSLIDE	9	set/get	Position für vertikalen Schieber (w1: 0: oben, 1000: unten)
WF_TOP	10	set/get	set: Fenster wird aktuelles (oberstes) Fenster get: liefert Nummer des aktuellen Fensters (w1: handle)
WF_FIRSTXYWH	11	get	Koordinaten des ersten Rechtecks in Rechteckliste des Fensters (w1: x-, w2: y-Position, w3: Breite, w4: Höhe)
WF_NEXTXYWH	12	get	Koordinaten des nächsten Rechtecks in Rechteckliste des Fensters (w1: x-, w2: y-Position, w3: Breite, w4: Höhe)
WF_RESVD	13	reserviert	
WF_NEWDESK	14	set	neues Desktop setzen (w1: (High) und w2 (Low): Adresse des Objektbaums (ein Nullzeiger setzt GEM-Desktop), w3: Objekt nummer des ersten darzustellenden Objekts)
WF_HSLSIZE	15	set/get	relative Größe des horizontalen Schiebers (w1: 1-1000; -1: Minimalgröße)
WF_VSLSIZE	16	set/get	relative Größe des vertikalen Schiebers (w1: 1-1000; -1: Minimalgröße)
WF_SCREEN	17	get	Adresse (w1: High-Word, w2: Low-Word) und Länge (w3: High-Word, w4: Low-Word) des internen Puffers zum Zwischenspeichern des Hintergrunds bei Drop-Down-Menüs und Alert-Boxen
ab Atari-GEM 3.0 (TT):			
WF_COLOR	18	set	Farben der Fensterelemente (w1: Element, w2: Farbe für 'Fenster aktiv', w3: Farbe für 'Fenster inaktiv') Fensterelemente (Werte für a):
W_BOX	0		Fensterhintergrund
W_TITLE	1		Titelbox
W_CLOSER	2		Schließbox
W_NAME	3		Namenszeile
W_FULLER	4		Volle-Größe-Box
W_INFO	5		Infozeile
W_DATA	6		restliche Elemente
W_WORK	7		Arbeitsbereich
W_SIZER	8		Größenveränderungsbox
W_VBAR	9		Elemente des vertikalen Balkens
W_UPARROW	10		Pfeil nach oben
W_DNARROW	11		Pfeil nach unten
W_VSLIDE	12		Hintergrund für vertikalen Schieber
W_VELEV	13		vertikaler Schieber
W_HBAR	14		Elemente des horizontalen Balkens
W_LFARROW	15		Pfeil nach links
W_RTARROW	16		Pfeil nach rechts
W_HSLIDE	17		Hintergrund für horizontalen Schieber
W_HELEV	18		horizontaler Schieber
WF_DCOLOR	19	set	Standardfarben für Fensterelemente

Tabelle 2: Funktionen für wind_get() und wind_set()

Fehler wird mit der Rückgabe einer 0 angezeigt.

Damit ist unser Fenster vom Bildschirm verschwunden, existiert für GEM aber noch und kann jederzeit wieder mittels *wind_open()* geöffnet werden! Um es aus der GEM-Liste zu löschen, benötigen wir einen weiteren Aufruf:

```
error = wind_delete(w_handle);
```

Jetzt ist es endgültig begraben, seine Identifikationsnummer kann von GEM für ein anderes Fenster verwendet werden, und wir können, nach Abmeldung beim GEM [im Manager die Funktion *gem_exit()*], das Programm beenden.

Würden wir mit den bisher vorgestellten Routinen ein Fenster öffnen, würde dies zwar auf dem Desktop (mit Inhalt) erscheinen, hätte aber einen gravierende Mangel: man könnte zwar mit der Maus versuchen, die Größe oder Lage zu verändern, es zu schließen oder den Inhalt zu scrollen, nur würde unser Fenster stur auf dem ursprünglichen Platz in seiner Eröffnungsgröße verharren. Wir benötigen also weitere Routinen zur Fenstermanipulation.

Let's go ...

GEM-AES nimmt uns für unser Vorhaben eine Menge Arbeit ab. Wir brauchen uns um nichts weiter zu kümmern, als die gewünschten Parameter beim Öffnen eines Fensters richtig zu setzen und dann auf eine Meldung des GEM zu warten, die uns mitteilt, was der Benutzer mit unserem Fenster anstellen möchte, und gleich auch noch die entsprechenden Werte liefert.

Nachdem wir ein Fenster auf den Bildschirm gebracht haben, warten wir, d.h. unser Programm, darauf, daß der Benutzer etwas tut. Die entsprechende Routine, die uns diese Arbeit abnimmt und uns die vom Benutzer getätigten Aktionen meldet, ist *event_mesag()*. Hier der Aufruf:

```
event = evnt_mesage(buffer);
```

Der Eingabeparameter ist ein Zeiger auf einen Speicherbereich von 16 Bytes Länge [Wordweise (16Bit) addresiert] für die Meldungen. Der Rückgabewert ist (zumindest zur Zeit) immer 1.

Nachfolgend eine Liste der möglichen Nachrichten (stehen immer in *buffer[0]*; Wert in Klammern), auf die unser Programm entsprechend zu reagieren hat (eine Zusammenfassung finden Sie in Tabelle 3):

```
typedef struct grect
{
   short g_x;
   short g_y;
   short g_w;
   short g_h;
} GRECT;

typedef struct wind_data  /* Fensterdaten-Struktur */
{
   char  name[80];   /* Fenstername                    */
   char  titel[80];  /* Text für Titelzeile            */
   GRECT max;        /* Maximalgröße                   */
   GRECT work;       /* Arbeitsbereichgröße            */
   WORD  elements;   /* Bestandteile des Fensters      */
   WORD  align;      /* Faktor zur hor. Ausrichtung    */
   WORD  snap;       /* Fenster snappen (TRUE/FALSE)   */
   WORD  full;       /* Full-Flag (TRUE/FALSE)         */
   WORD  scroll_x;   /* Scroll-Wert für x-Achse        */
   WORD  scroll_y;   /* Scroll-Wert für y-Achse        */
   WORD  doc_x;      /* x-Position des Dokumentes      */
   WORD  doc_y;      /* y-Position des Dokumentes      */
   long  doc_length; /* Dokumentslänge                 */
   WORD  doc_width;  /* Dokumentsbreite                */
   void  (*draw)();  /* Pointer auf Zeichenfunktion    */
} WIND_DATA;
```

Listing 1: Definition der GRECT- und Fensterdaten-Struktur

1. MN_SELCTED(10): Meldung wird erzeugt, wenn ein Menüeintrag angeklickt wurde. In diesem Fall enthält *buffer[3]* die Objektnummer des Menütitels und *buffer[4]* die Nummer des angewählten Menüeintrags.

2. WM_REDRAW(20): GEM verlangt von uns, Fenster bzw. Ausschnitte davon neu zu zeichnen, da diese zerstört wurden (z.B. durch Verschieben eines anderen Fensters). *buffer[3]* enthält dabei die Identifikationsnummer (richtig, das handle!) des wiederherzustellenden Fensters, *buffer[4]* bis *buffer[7]* die Koordinaten (x, y, Breite, Höhe) des Fensterbereichs, der neu gezeichnet werden muß.

3. WM_TOPPED(21): Das Fenster, dessen handle in *buffer[3]* steht, wird zum neuen aktuellen Fenster, da der Anwender dieses angeklickt hat. Gehört das Fenster zu unserem Programm, erledigen wir dies mittels wind_set(): *wind_set*(buffer[3], WF_TOP, buffer[3], 0, 0, 0);

4. WM_CLOSED(22): Der Benutzer hat die Schließecke des Fensters angeklickt, d.h. unser Programm soll das Fenster vom Bildschirm verschwinden lassen. Mit *wind_close(buffer[3])* kein Problem!

5. WM_FULLED(23): Der Benutzer hat die Volle-Größe-Ecke angeklickt, und unser Programm soll entweder das Fenster (handle in *buffer[3]*) auf die volle Größe bringen (normalerweise der ganze Desktop bzw. die in *wind_create()* angegebenen Werte) oder, wenn dies bereits der Fall ist, es auf die vorherige Größe verkleinern. GEM ermöglicht dies, da es die vorherigen, die momentanen und die maximalen Koordinaten eines Fensters speichert. Wir können mit *wind_get()* darauf zugreifen. Im Manager erledigt diese Aufgabe eine eigene Funktion *handle_full()*. Sie ermittelt zuerst die vorherigen, momentanen und maximalen Koordinaten, vergleicht die momentane Größe mit der maximalen und vergrößert bzw. verkleinert dann das Fenster wie gewünscht.

6. WM_ARROWED(24): Pfeile oder grauschraffierter Bereich des Scroll-Balkens des Fensters aus *buffer[3]* wurden angeklickt. In *buffer[4]* befindet sich eine genaue Angabe über das angewählte Element (siehe Tabelle 3).

7. WM_HSLID(25): Der horizontale Schieber des Fensters mit der Nummer aus *buffer[3]* wurde bewegt. Neue Position [Wert zwischen 0 (links) und 1000 (rechts)] steht in *buffer[4]*.

8. WM_VSLID(26): Wie 7., nur für vertikalen Schieber (0: oben, 1000: unten).

9. WM_SIZED(27): Die Fenstergröße (Außenmaß) wurde verändert. *buffer[3]* teilt uns die Fensternummer mit (wer hätte das gedacht ...?), *buffer [4]* bis *buffer [7]* die x- und y-Koordinaten sowie die neue Breite und Höhe.

10. WM_MOVED(28): Das Fenster wurde verschoben. buffer [3] enthält wieder die ID des Fenster und buffer[4] bis buffer[7] die neuen Koordinaten (Außenmaß).
11. WM_NEWTOP(29): Funktion ist nicht mehr dokumentiert und sollte auch nicht auftreten (hatte wohl ähnliche Funktion wie 3.)

Die folgenden Meldungen sind nur der Vollständigkeit halber aufgeführt. Sie finden Verwendung bei der Programmierung von Accessories.
12. AC_OPEN(40): *buffer[4]* enthält die Nummer des Accessories, die durch den Aufruf vom *menu_register()* diesem Programm mitgeteilt wurde. Diese Meldung erhält ein Accessory, wenn sein Menüpunkt im Desk-Menü angeklickt wurde.(Übrigens: im PC-GEM erscheint die Nummer des Accessories in buffer[3]!)
13. AC_CLOSE(41): Diese Meldung erhält ein Accessory, wenn a) das laufende Programm beendet wurde, b) der Bildschirm gelöscht wird oder c) Daten der Window-Library reinitialisiert werden. In *buffer[3]* befindet sich die ID des entsprechenden Accessories. Man darf jedoch erst ab Atari-GEM 3.0 (ab TT) sicher sein, daß die Meldung vor dem Ende des Hauptprogramms eintrifft (d.h. vor Verlust von Speicherblöcken und Dateikennungen)!

Eine weitere Meldung existiert noch, die vom modularen Kontrollfeld (XCONTROL) zur Mitteilung von Tasteneingaben genutzt wird:

14. CT_KEY(53)

Sie können auch eigene Mitteilungen definieren und verschicken. So ist eine Kommunikation zwischen verschiedenen Programmen und Accessories möglich. In Gemini wird dies als 'AV-Protokoll' verwendet. Von DigitalResearch werden dafür Mitteilungsnummern größer als 1024 empfohlen.

Anmerkung: Anstelle von *evnt_mesag()*, die nur die von GEM-AES abgeschickten Meldungen übermittelt, können wir auch *evnt_multi()* gebrauchen. Diese GEM-Funktion vereinigt mehrere andere Funktionen in sich. So kann mit ihr auch eine Meldung empfangen werden, für die sonst *evnt_button()*, *evnt_click()*, *evnt_keybd()*,

MN_SELCTED(10):	Ein Menüeintrag wurde angewählt buffer[3]: Objektnummer des Menütitels buffer[4]: Nummer des angewählten Menüeintrags
WM_REDRAW(20):	Fenster(ausschnitt) restauriert werden buffer[3]: handle buffer[4]: x-Koordinate buffer[5]: y-Koordinate buffer[6]: Breite buffer[7]: Weite
WM_TOPPED(21):	Fenster wird aktuelles Fenster buffer[3]: handle
WM_CLOSED(22):	Schließbox wurde angeklickt buffer[3]: handle
WM_FULLED(23):	Volle-Größe-Ecke wurde angeklickt buffer[3]: handle
WM_ARROWED(24):	Pfeile oder Scroll-Balken wurden angeklickt buffer[3]: handle buffer[4]: angewähltes Element: WA_UPPAGE(0) : Balken oberhalb des Schiebers WA_DNPAGE(1) : Balken unterhalb des Schiebers WA_UPLINE(2) : Pfeil nach oben WA_DNLINE(3) : Pfeil nach unten WA_LFPAGE(4) : Balken links des Schiebers WA_RTPAGE(5) : Balken rechts des Schiebers WA_LFLINE(6) : Pfeil nach links WA_RTLINE(7) : Pfeil nach rechts
WM_HSLID(25):	horizontaler Schieber wurde bewegt buffer[3]: handle buffer[4]: neue Position (0: links, 1000: rechts)
WM_VSLID(26):	vertikaler Schieber wurde bewegt buffer[3]: handle buffer[4]: neue Position (0: oben, 1000: unten)
WM_SIZED(27):	Fenstergröße (Außenmaß) wurde verändert buffer[3]: handle buffer[4]: x-Koordinate buffer[5]: y-Koordinate buffer[6]: Breite buffer[7]: Weite
WM_MOVED(28):	Fenster wurde verschoben; neue Werte sind Außenmaß buffer[3]: handle buffer[4]: x-Koordinate buffer[5]: y-Koordinate buffer[6]: Breite buffer[7]: Weite
WM_NEWTOP(29):	Funktion nicht mehr dokumentiert (hatte ähnliche Funktion wie WM_TOPPED)
AC_OPEN(40):	Accessory-Eintrag wurde angeklickt buffer[4]: Accessory-Nummer
AC_CLOSE(41):	Accessory wurde geschlossen (Start oder Ende eines Programms; Fenster muß nicht geschlossen werden) buffer[4]: Accessory-Nummer
CT_KEY(53):	Meldung von Tasteneingaben (wird von XCONTROL verwendet)

Tabelle 3: Mitteilungs-Ereignisse des AES; Nummer steht in buffer[0]

evnt_mesag(), *evnt_mous()* und *evnt_timer()* zuständig sind. Das heißt z.B., daß das Programm auf eine Menüselektion, eine Window-Aktivität, einen Tastendruck, eine Mausbewegung und einen Timer-Event gleichzeitig warten kann. Diese Routine wird, durch Ihre Universabilität, in den meisten Fällen eingesetzt. Allerdings sollte man bei der Verwendung von *evnt_multi()* beachten, daß durch die Vielzahl der Parameter, bedingt durch das Zusammenfassen mehrerer Ereignis-Routinen, beträchtlich Rechenzeit verbraucht wird (von DigitalResearch wird deshalb im GEM/3-Toolkit eine Alternative angeboten, bei der nur ein Zeiger auf eine Struktur übergeben wird).

Mit Hilfe der vom AES gelieferten Meldungen können wir nun auf die Wünsche des Anwenders reagieren und das Fenster verschieben, verkleinern oder vergrößern und den Inhalt verändern.

Stellen Sie sich den Fall vor, daß Sie in Ihrem Programm am linken Bildschirmrand eine 'Tool-Kiste' (anwählbare Symbole für Operationen; z.B. bei einem Layoutporgramm) haben. Diese sollte immer zu sehen sein. Außerdem möchten Sie nicht, daß Teile des Fensters außerhalb des Bildschirms zu liegen kommen. Dieses können Sie mit den bisher vorgestellten Routinen nicht, da Sie keinen Einfluß auf die Schiebungen des Anwenders haben. Wir brauchen also eine Routine zum 'Snappen'.

Jetzt wird 'gesnappt'!

Um die oben erwähnten Effekte zu verhindern, können wir das ganze Fenster snappen, d.h. es in einen definierten Bildschirmbereich zurückschieben. „Snappen" bedeutet also, dafür zu sorgen, daß ein Fenster immer ganz innerhalb eines bestimmten Bereiches zu sehen ist (meist auf dem ganzen Desktop, natürlich ohne Menüleiste!).

Hinzu kommt, daß man dafür sorgt, daß die Koordinaten (der Arbeitsfläche) unseres Fensters nur ganzzahlige Vielfache eines definierten Wertes annehmen. Dies ist. z.B. nützlich, wenn man in einem Fenster Text ausgeben möchte. Denn sind die Fenstergrenzen an Wortgrenzen ausgerichtet (Align-Wert 8), läuft die Textausgabe etwas schneller ab. Für diese beiden Vorhaben brauchen wir zwei Funktionen, die wir im folgenden kennenlernen werden.

Zum ersten ...

snap() erwartet als Parameter zwei Pointer auf eine *GRECT-Struktur*, d.h. zwei Pointer auf die Koordinaten eines Rechtecks. Das erste definiert den Bereich, in dem sich das Fenster befinden soll [bei Aufruf normalerweise gleich den durch *wind_create()* mitgeteilten Maximalkoordinaten], und das zweite spezifiziert die Größe des Fensters.

In einem Schritt vergleicht die Routine die x- und y-Koordinate des Fensters mit denen des definierten Bereichs. Falls das Fenster außerhalb des Bereichs liegt (d.h. falls die x- und/oder y-Koordinate kleiner als die des Bereichs ist), schiebt snapp() das Fenster in den Bereich zurück; Schritt 1 in Bild 2 veranschaulicht die Problematik hierzu.

Im zweiten Schritt überprüft *snap()* die rechte untere Ecke des Fensters. Wenn deren Koordinaten (Addition von x-Koordinate mit der Breite bzw. von y-Koordinate mir der Höhe) außerhalb des Bereichs liegen, verfährt *snap()* mit dem Fenster wie in Schritt 1 (Bild 2, Schritt 2). Nun sollte unser Fenster eigentlich vollständig innerhalb des Bereichs liegen. Schritt 3 in Bild 2 zeigt aber, daß das nicht immer der Fall sein muß. Angenommen, das Fenster besitzt schon fast seine maximale Höhe und der Benutzer betätigt SIZER-BUTTON, um es noch weiter zu vergrößern, dann passiert folgendes: Schritt 1 tritt nicht in Kraft, da die Größenänderung keinen Einfluß auf die x- bzw. y-Koordinaten des Fensters hat. In Schritt 2 setzt *snap()* die rechte untere Ecke des Fensters gleich der des Bereichs, da die des Fensters infolge der Größenänderung außerhalb des definierten Rechtecks zu liegen kam. Jetzt befindet sich die y-Koordinate des Fensters aber außerhalb des Bereichs (vgl. Schritt 3 in Bild 2)! Es wird also nötig, einen dritten Schritt einzuführen, der, falls der oben geschilderte Zustand eintritt, die x-Koordinate und die Breite bzw. die y-Koordinate und die Höhe gleich den Werten des eingeschränkten Rechtecks setzt. Damit liegt unser Fenster immer innerhalb des angegebenen Bereichs!

Es drängt sich auf, *snap()* beim Auftreten einer *WM_SIZED*- bzw. einer *WM_MOVED*-Mitteilung zu verwenden, um dem Anwender auf die Finger (bzw. auf die Maus) zu schauen.

Mit Hilfe einer vorher zu definierenden Variablen (wird später in eine Struktur mit wichtigen Fensterwerten integriert) kann festgelegt werden, ob das Snappen bei einem Fenster erwünscht ist oder nicht. Falls ja, führen wir dies aus und übergeben *snap()* als Begrenzung die in einem Array festgelegten Maximalkoordinaten des Fensters und als snappende Werte die von *event_mesag()* in *buffer[4]* bis *buffer[7]* übergebenen neuen Koordinaten.

[**Amerkung:** Eigentlich erwartet *snap()* Pointer auf *GRECT*-Strukturen. Im ersten Fall ist dies zwar erfüllt, doch im zweiten Fall übergebe ich die Adresse von *buffer[4]*, was zwar nicht ganz sauber ist, C aber durchläßt, da ich ein Array von n WORD-Variablen (WORDarray[n];) auch als Struktur von n WORD-Variablen (struct Array {WORD x1,x2,x3, ...,xn}array;) beschreiben kann und die einzelnen Variablen im Speicher bei beiden Varianten hintereinander gespeichert werden. (Der Zugriff auf die einzelnen Elemente ist natürlich verschieden: array[n] bzw. array.xn).]

... und zum zweiten

Die Funktion *align()* ermöglicht es uns, einen Koordinatenwert auf ein ganzzahliges Vielfaches einer Konstanten zu begrenzen.

Zuerst erhöhen wir die Koordinate um die Hälfte des Align-Wertes minus Eins [n>>1 bedeutet, daß die Variable n um 2^1 nach rechts geshiftet (geschoben) wird, was einer Division durch Zwei entspricht]. Dann teilen wir diesen Wert durch den Align-Wert, nehmen davon den ganzzahligen Teil und multiplizieren ihn wieder mit dem Align-Wert, womit wir bereits das ausgerichtete Ergebnis hätten. Beispiel: k=111 und v=16. Zuerst erhöhen wir k um 16 geteilt durch 2 minus 1, also 7 ergibt k=118. Dann teilen wir dies durch 16 und nehmen den ganzzahligen Teil: 7. Dies mal 16 ergibt den gesuchten Wert 112, der ein ganzzahliges Vielfaches von 16 ist und dem ursprünglichen Wert 111 am nächsten liegt.

Was geschieht nun, wenn man ein Fenster über ein anderes schiebt und es danach wieder wegnimmt? Ist dann nicht der Inhalt des zweiten Fensters an den Stellen, die vom ersten bedeckt waren, zerstört? Und was können wir dagegen unternehmen?

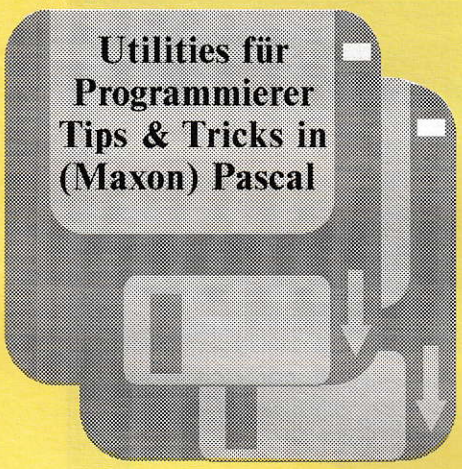
NEU

Programmiererpower im Doppelpack !

Einfach einlegen und los geht's mit den Disketten zum

Sonderheft ST-Computer

Doppelpack PASCAL (2 Disketten)



Doppelpack C (2 Disketten)

Doppelpack BASIC (2 Disketten)

Aus dem Inhalt:

Alle Disketten enthalten Routinen und Beispielprogramme zu folgenden Bereichen:

Grundlagen:	z.B. Window-Routinen, Fractale, Submenüs, ...
Grafik:	z.B. Scrolling, Überblenden, Radieren, Ausschneiden und Kopieren, Zoomen, ...
Betriebssystem:	z.B. eigener Desktop, schnelle Dialogboxen, Checkboxen, Short-Cuts in Menüs, ...
Tricks:	z.B. Formelinterpreter, Editor-Modul

sowie 12 Utilities für den ProgrammiererInen
z.B.: Dateisuche auf Laufwerken, Hard-Disk-Info, etc.

Heim Verlag

Bestellcoupon

Heidelberger Landstraße 194
6100 Darmstadt-Eberstadt
Telefon (0 61 51) 5 60 57
Telefax (0 61 51) 5 60 59

Ja, ich bestelle

___	Doppelpack BASIC Disketten	a	15,-- DM
___	Doppelpack PASCAL Disketten	a	15,-- DM
___	Doppelpack C Disketten	a	15,-- DM
___	alle 3 Doppelpacks	a	40,-- DM
___	Sonderheft(e)	a	14,-- DM

Name _____

Straße _____

Plz, Ort _____

zuzüglich DM 3,- Versankosten (Ausland DM 5,-) unabhängig von der bestellten Stückzahl
Alle Preise sind unverbindlich empfohlene Verkaufspreise

☐ per Nachnahme ☐ Scheck anbei

Der (sichtbare) Inhalt des überlappten Fensterbereichs ist zerstört, und dagegen unternehmen kann man nichts. (Ist wohl auch nicht im Sinne des Erfinders; soll doch der Anwender den Bildschirm so aufbauen können, wie er möchte!) Doch wir können etwas danach in die Wege leiten.

Redraw

Unter Redraw versteht man ein Wiederherstellen des alten Fensterinhaltes, wenn dieser von einer Dialogbox oder einem anderen Fenster überlagert wurde und nun wider offen zutage liegt.

Zum Glück nimmt uns auch hierbei das AES ein Teil der Arbeit ab (Sie erinnern sich möglicherweise noch an die Nachricht *WM_REDRAW* der *event_mesag()*-Routine?). Wir brauchen bloß zu warten, bis wir mittels *evnt_mesag()* [*evnt_multi()* kann auch verwendet werden] in *buffer[0]* den Wert von *WM_REDRAW* erhalten und führen dann das Redrawing in einer managerinternen Routine durch.

Und wie? Die Funktion *do_redraw()* erledigt für uns das Drum und Dran beim Redrawing. Wir übergeben ihr den von *evnt_mesag()* gelieferten Puffer und lassen sie dann arbeiten:
Nachdem sie mittels

```
v_hide_c(handle)
```

den Cursor ausgeschaltet hat, der beim Neuzeichnen stören würde, und danach mit

```
wind_update(BEG_UPDATE)
```

dem AES mitgeteilt hat, daß sie nun die Kontrolle übernimmt (mit anderen Worten, kein Menü mehr heruntergeklappt und auch alle anderen Benutzeraktionen ignoriert werden sollen), beginnt die eigentliche Routine. AES übermittelt uns in *buffer[4]* bis *buffer[7]* die Koordinaten des zerstörten Bildschirmbereichs. Das genügt aber nicht, denn wir müssen wissen, welche Bereiche eines Fensters neu gezeichnet werden müssen (Desktop-Bereiche zeichnet GEM selbständig neu). Dazu brauchen wir die sogenannte Rechteckliste dieses Fensters. Darin ist der sichtbare Arbeitsbereich in verschiedene Rechtecke aufgeteilt (Bild 3 verdeutlicht den Sachverhalt). Ist das Fester ganz sichtbar, besteht die Liste natürlich nur aus einem Rechteck (vgl. Fenster 2 in Bild 3), bei teilweise verdeckten Fenstern dagegen aus mehreren (vgl. Fenster 1 in Bild 3).

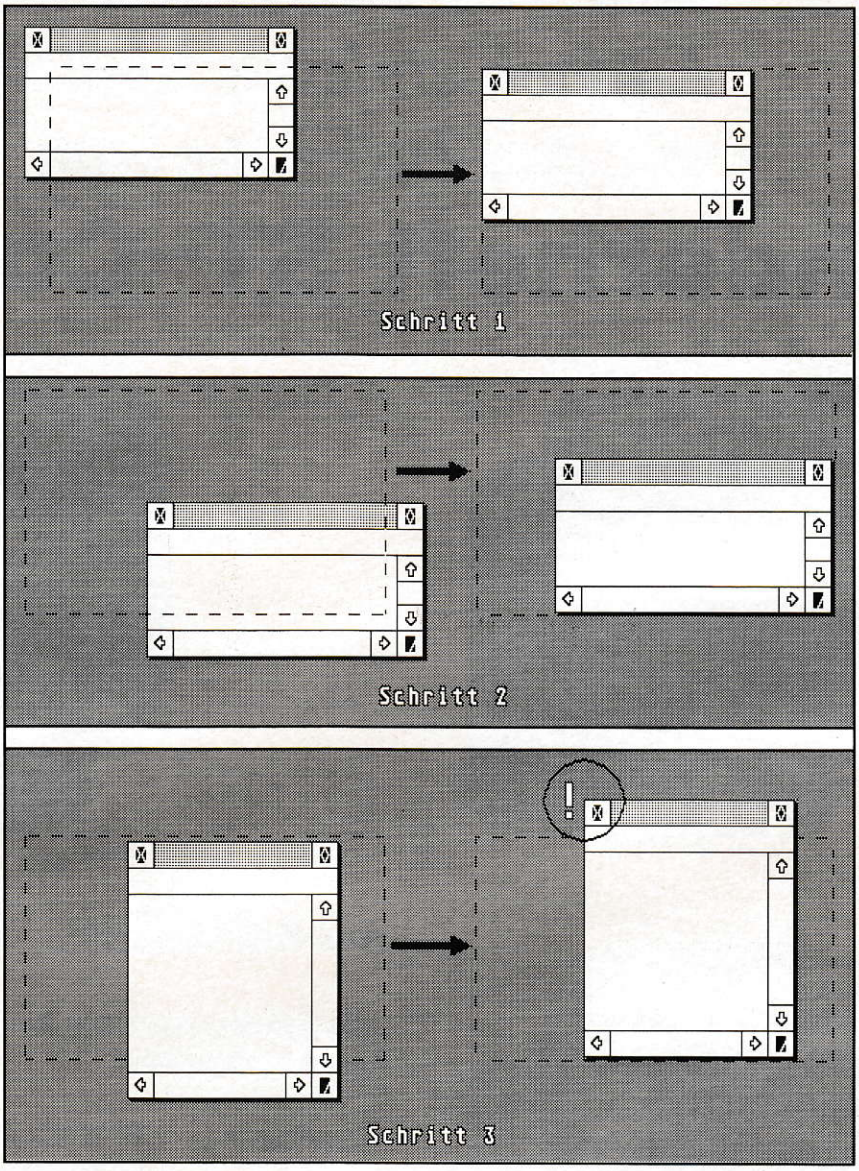


Bild 2: Snappen eines Fensters

Die Koordinaten des ersten Rechtecks erhalten wir durch

```
wind_get(buffer[3], WE_FIRSTXYWH, x, y, b, h),
```

die nächsten durch wiederholten Gebrauch von

```
wind_get(buffer[3], WF_NEXTXYWH, x, y, b, h).
```

Wenn Breite und Höhe der erhaltenen Koordinaten gleich Null sind, sind keine weiteren Rechtecke vorhanden. Bild 3 zeigt, was passiert, wenn z.B. eine Dialogbox einen Teil des Bildschirms verdeckt hat und der darunterliegende (schraffierte) Bereich neu gezeichnet werden muß. *do_redraw()* nimmt sich in einer Schleife alle vorhanden Rechtekke vor und untersucht auf Überschneidungen mit dem zerstörten Bidschirmbereich. Dazu werden in *rc_intersect()* die Koordinaten beider Bereiche verglichen, bei einer Überlappung die Koordinaten des Schnittbereichs berechnet und TRUE zurückgegeben. Ist dies der Fall, d.h. muß ein Bereich neu gezeichnet werden, clippt *do_redraw()* mit *vs_clip()* die Bildschirmausgabe, d.h. es werden nur Ausgaben in den angegebenen Bereich erlaubt. Darauf löscht sie dieses Rechteck. Jetzt wird es Zeit, die eigentliche Draw-Routine für das

Fenster aufzurufen (genaueres siehe weiter unten).

Nach Abschluß dieser Arbeit, also nachdem alle beschädigten Bereiche eines Fensters neugezeichnet worden sind, wird das Clipping wieder ausgeschaltet (sonst sind normale Ausgaben nicht mehr möglich) und mit

```
wind_update(END_UPDATE)
```

dem AES die Kontrolle wieder übergeben sowie der Cursor angeschaltet

```
v_show_c(handle, TRUE),
```

bis eine neue REDRAW-Message vor der Tür steht ... (AES schickt für jedes betroffene Fenster eine eigene Meldung, die bearbeitet werden muß).

Die Adresse der Draw-Routine wird in der oben bereits erwähnten Fensterstruktur (näheres folgt in Kürze) eingetragen. Damit kann die *do_redraw()*-Routine für jedes beliebige Fenster die Zeichenroutine aufrufen.

Anmerkung: Der Methode, derer *do_redraw()* sich bedient (Ermitteln der Rechteckliste bei einer REDRAW-Meldung mit anschließendem Neuzeichnen dieser), sollten alle GEM-Applikationen folgen, auch wenn sie nur ein Fenster öffnen, denn es kann ja z.B. durch ein Accessory überdeckt werden. Außerdem kommt hier eines der grundlegenden Konzepte von GEM, nämlich das Message-Prinzip, voll zur Geltung - eines der am meisten mißverstandenen Konzepte, das an sich flexible Programmgestaltung und ein gewisses Multitasking erlaubt (siehe Accessories).

Welche Operationen fehlen uns denn noch? Sämtliche Manipulationen, die etwas mit dem Inhalt des Fenster zu tun haben (horizontales oder vertikales Scrollen des Textes).

Als erfahrene GEM-Benutzer ist Ihnen dies bestimmt mehr als geläufig; um z.B. in FirstWord den Text zu scrollen, greifen Sie zur Maus und klicken einen der Rollpfeile an, verschieben die Rollbox an eine andere Position oder klikken das graue Feld des Rollbalkens an (vergleichen Sie auch mit Bild 1, dort sehen Sie die Elemente eines Fensters beschrieben). Was geschieht, wenn Sie einen Rollpfeil anklicken? Der Text bewegt eine Zeile nach oben oder unten bzw. einen Buchstaben nach rechts oder links, je nachdem, welchen der vier Rollpfeile Sie angeklickt haben. Wenn Sie den grauen Bereich eines Rollbalkens anklicken, verschiebt sich der Fensterinhalt um eine Seite vorwärts oder rückwärts bzw. nach links oder rechts, wobei die oberste bzw. unterste Zeile der vorherigen Seite sichtbar bleibt. Ganz anders aber, wenn Sie die Rollbox mit der Maus verschieben. Dann springt der Text an die entsprechende Stelle. Es ist so, daß die Länge des Rollbalkens die Länge des Dokuments, das sich hinter dem Fenster versteckt, darstellt. Die weiße Rollbox steht für den momentan sichtbaren Abschnitt des Fensters. D.h. das Fenster stellt eine Art von Sichtfenster dar, durch das Sie einen Ausschnitt des größeren Dokumentes sehen können, je nachdem, über welchem Teil Sie es mit dem Rollbalken plaziert haben.

Die Realisierung

Soweit, so gut. Das Ganze sieht ziemlich einfach aus, jedenfalls für den Benutzer, für den Programmierer stellen sich jedoch einige Schwierigkeiten. Im folgenden werde ich Ihnen beschreiben, wie die Verwaltung von Rollbalken in unserer Window-Bibliothek eingebunden ist.

Zuvor jedoch ein paar Worte zu der bereits öfters angesprochenen Fensterstruktur. Um einen Window-Manager zu schreiben, der unabhängig vom eigentlichen Programm arbeiten soll, muß es Zugriff auf bestimmte Fenstergrößen haben. Die in der Fensterstruktur zusammengefaßten Daten können Sie Listing 1 entnehmen (hier in C-Notation). Zum einen finden Sie dort Zeiger auf Texte für den Namen und die Info-Zeile. Diese werden benötigt, damit die *open_window()*-Routine des Managers ein Fenster auch ohne Fehler öffnen kann. Danach finden Sie die Größe der Maximalausdehnung und des aktuellen Arbeitsbereichs (beide in einer *GRECT*-Struktur, deren Aussehen Sie ebenfalls Listing 1 entnehmen können). Neben den Elementen, mit denen das Fenster ausgestattet ist, sind in dieser Struktur auch die Werte für die *snap()*- und *align()*-Funktionen. Ein Flag gibt Auskunft darüber, ob das Fenster in seiner maximalen Ausdehnung dargestellt wird oder nicht. Die Scroll-Werte für die x- und y-Richtung sowie die Position innerhalb des Dokuments und dessen Länge werden für die folgenden Routinen benötigt. An letzter Stelle finden Sie den Zeiger auf die Draw-Routine, der von der Redraw-Routine benötigt wird. Der Window-Manager definiert 8 Variablen dieser Struktur mit dem Namen windows, mit deren Hilfe auf die verschiedenen Fenster zugegriffen wird.

Auf die Felder der Window-Struktur, auf die unsere *scroll_wind()*-Routine zugreift, möchte ich noch etwas näher eingehen:

work: enthält die aktuellen Koordinaten des Arbeitsbereichs [die Fläche innerhalb der Titelzeile (evtl. Infozeile) und den Rollbalken]

scroll_x: Scroll-Wert für x-Richtung, um diesen Betrag wird der Fensterinhalt bei Betätigung des horizontalen Schiebers gescrollt. Bei Textausgabe normalerweise 8 (entspricht den Pixeln auf dem Bildschirm).

scroll_y: Scroll-Wert für y-Richtung, um diesen Betrag wird der Fensterinhalt bei Betätigung des vertikalen Schiebers gescrollt. Bei Textausgabe normalerweise 16.

doc_x: horizontale Position der linken oberen Ecke des Fensters im Dokument (nicht mit Bildschirmkoordinaten zu verwechseln). Zeigt die linke Spalte z.B. den dritten Buchstaben eines Tex-

```
gem_init();
  ...
w_handle = open_window(...);
      ...
      ...
   evnt_mesag(buffer); /* oder evnt_multi(...); */
   handle_window(buffer);
      ...
      ...
gem_exit();
```

Bild 4: Prinzipieller Verlauf eines GEM-Programms mit Fenster

tes mit x-Scroll-Weite 8 an, dann beträgt doc_x 16.

doc_y: vertikale Position der linken oberen Ecke des Fensters im Dokument. Zeigt die oberste Zeile des Fensters z.B. die dritte Zeile eines Textes mit y-Scrollweite 16 an, denn beträgt doc_y 32.

doc_length: Länge des Dokumentes. Angabe in Pixeln. Ein Text von 100 Zeilen ist so z.B. bei einer Zeilenhöhe von 16 Pixeln 1600 Pixel lang.

doc_width: Breite des Dokumentes. Ein Text mit der Breite von 65 Buchstaben und 8 Pixeln pro Buchstabe ist daher 520 Pixel breit.

Zu wissen, welche Aktion der Benutzer getätigt hat, ist kein Problem; dies meldet uns ja das AES mittels der Messages *WM_ARROWED*, *WM_HSLID* und *WM_VSLID*. Eine dieser drei Meldungen erhalten wir, wenn wir unser Fenster mit den entsprechenden Elementen beim Öffnen ausgestattet haben und der Benutzer eines davon angeklickt hat. *WM_ARROWED* bedeutet dabei, daß einer der Rollpfeile oder einer der grauen Bereiche angeklickt wurde und der Benutzer entweder eine Zeile oder eine Seite weiter- bzw. zurückblättern oder nach rechts oder links blättern möchte. *WM_HSLID* meldet AES bei Verschiebung der horizontalen Rollbox, analog *WM_VSLID* bei Verschieben der vertikalen Rollbox.

Um diese Meldungen zu handhaben, existieren entsprechende Funktionen, die die notwendigen Schritte durchführen: Mittels *scroll_wind()* verwalten wir eine *WM_ARROWED*-Message. Der Funktion übergeben wir das handle des Windows (buffer[3]) und den Inhalt von *buffer[4]*, der die vom Benutzer getätigte Aktion genauer spezifiziert (siehe Tabelle 2). Vergleichen Sie hierzu das switch-statement in *scroll_wind()*. Je nachdem wird dann die Dokumentenposition um den Scroll-Wert vermindert oder erhöht. Beispiel:

```
case W_UPLINE:
  windows[w_handle].doc_y -=
  windows[w_handle].scroll_y.
```

Um aber zu verhindern, daß der neue Wert größer als die Dokumentenlänge bzw. kleiner als Null wird, muß eine zusätzliche Überprüfung durchgeführt werden:

```
if(windows[w_handle].doc_y < 0)
  windows[w_handle].doc_y = 0
```

bzw.

```
if(windows[w_handle].doc_y >
  windows[w_handle].doc_length)
    windows[w_handle].doc_y =
    windows[w_handle].doc_length
```

Diese Methode funktioniert bei zeilenweisem Rollen; bei seitenweisem Verschieben kommt aber noch hinzu: es wäre sinnvoll, jeweils die letzte Zeile der vorherigen Seite mit anzuzeigen. Eine Seite entspricht hier immer der aktuellen Größe des Arbeitsbereiches. Wir müssen die Dokumentsenposition also um die Größe des Arbeitsbereichs plus/minus der x- bzw.. y-Scroll-Weite vermindern/erhöhen.

Beispiel:

```
case W_DNPAGE:
  windows[w_handle].doc_y +=
  windows[w_handle].work.g_h -
  windows[w_handle].scroll_y
```

Aufruf: gem_init()
Funktion: GEM-Initialisierung
Parameter: keine
Rückgabewert: keiner

Aufruf: gem_exit()
Funktion: 'Abmeldung' beim GEM
Parameter: keine
Rückgabewert: keiner

Aufruf: open_window(w_name, w_title, redraw, kind, algn, snp, s_x, s_y, doc_l, doc_w, x1, y1, w1, h1, mx, my, mw, mh);
Funktion: initialisiert Fensterstruktur, meldet Fenster bei GEM an, öffnet es, löscht den Inhalt und setzt die Schieberpositionen
Parameter:
w_name, w_title: Zeiger auf Text für Namens- und Titelzeile
redraw: Zeiger auf Zeichenfunktion
kind: Elemente des Fensters (siehe Tabelle 1)
algn: 0: keine Funktion, >0: align-Wert
snp: 0: keine Snappen, 1: Snappen wird mit maximal Koordinaten (mx, my, mw, mh) durchgeführt
s_x, s_y: Scrollw-Werte in x- und y-Richtung
doc_l, doc_w: Länge und Breite des Dokuments
x1, y1, w1, h1: Eröffnungskoordinaten
mx, my, mw, mh: Maximalkoordinaten; Desktop falls mw = 0
Rückgabewert: Fensternummer

Aufruf: handle_window(buffer)
Funktion: bearbeitet Anwenderaktionen am Fenster (Verschieben, Änderung der Größe, Inhalt scrollen ...)
Parameter: Zeiger auf Message-Puffer von evnt_mesag()
Rückgabewert: keiner

Aufruf: clear_window(w_handle)
Funktion: löscht Fensterinhalt
Parameter: w_handle: Fensternummer
Rückgabewert: keiner

Aufruf: full_redraw(w_handle)
Funktion: zeichnet Fenster komplett neu: Fenster löschen und Zeichen-Routine aufrufen
Parameter: w_handle: Fensternummer
Rückgabewert: keiner

Die Überprüfung auf Über- oder Unterlauf der maximalen bzw. minimalen Größe bleibt gleich. Nachdem die Dokumentenposition nun so verändert wurde, müssen die Slider (= Rollbalken) neu positioniert werden, da sie ja die Größe und die relative Position des sichtbaren Fensterinhaltes zum ganzen Dokument wiedergeben sollen. Hierzu dient *set_slider_pos()*, der wir das Handle des entsprechenden Fensters übergeben. Um die Position der Sliders setzen zu können, muß man wissen, daß die GEM-Entwickler die maximale Länge des Rollbalkens auf 1000 festgesetzt haben. Wir müssen also die Dokumentenposition entsprechend umrechnen. Hier die verwendete Formal:

```
Schieberposition =
  Dokumentenposition * 1000/
  (Dokumentengröße-Arbeitsbereichsgröße)
```

Ein spezieller Fall muß aber berücksichtigt werden: ist die Arbeitsbereichsgröße größer als die Dokumentengröße, würde die Schieberposition ungewollterweise negativ, wir müssen diesen Fall durch eine if-Abfrage abfangen. Nach der Berechnung der neuen Position können wir diesen Wert GEM mittels wind_set() mitteilen.

Für *WM_HSLID* und *WM_VSLID* existieren zwei eigene Funktionen: *wind_hslide()* und *wind_vslide()*. Beide berechnen im Prinzip dasselbe, nur entweder horizontal oder vertikal, nämlich die neue Dokumentenposition. Dies ist der umgekehrte Fall von vorhin, bei der *set_slider_pos()* war die Dokumentenposition bekannt, und die des Sliders mußte berechnet werden. Jetzt ist die des Sliders (*buffer[4]*) bekannt, und wir müssen die neue Dokumentenposition ermitteln. Aus der oben genannten Formel läßt sich dies leicht ableiten:

```
Dokumentenposition =
  Schieberposition * (Dokumentengröße -
  Arbeitsbereichsgröße)/1000.
```

Schließlich darf man nicht vergessen, die neue Slider-Position mittels *wind_set()* zu setzen, da dies GEM nicht selbständig macht. Zu allerletzt wird das ganze Fenster neu gezeichnet, da sich ja der Inhalt verschoben hat. Dazu rufen wir *full_redraw()* auf, das nichts anderes macht, als den Fensterinhalt mittels *clear_window()* zu löschen und je nach Fenster die entsprechende Redraw-Routine aufzurufen.

Sehr schön, aber ...

... darf man keinesfalls vergessen, daß, wenn das Fenster in der Größe verändert wird, sich die Schiebergrößen ändern, da die Rollbalkenlänge verändert wird. Wir rufen deshalb, nachdem das Fenster in der Größe verändert wurde, die Routinen *wind_calc_work()* und *set_slider_size()* auf. *Wind_calc_work()* berechnet aus den Koordinaten des gesammten Fensters diejenigen des Arbeitsbereiches und trägt die neuen Daten in die Struktur ein. *Set_slider_size()* berechnet, wie der Name schon sagt, die neue Größe der Slider und teilt diesen Wert GEM mit, das dann die Slider neu zeichnet. Des weiteren müssen wir auch daran denken, daß bei einer Betätigung des Volle-Größen-Ecks die Fenstergröße verändert wird.

Endlich

Sie haben nun eine praktische Window-Bibliothek in den Händen, die es Ihnen ermöglicht, Fenster in GEM einfach zu handhaben. Die einzelnen Funktionen des Window-Managers, welche Aufgaben sie erledigen und welche Werte Sie übergeben müssen bzw. zurückgeliefert bekommen, entnehmen Sie bitte dem Kasten. In Bild 4 sehen Sie den prinzipiellen Aufbau eines GEM-Programms bei Verwendung des Window-Managers. Sie können die komplette Bibliothek extra übersetzen und zum eigentlichen Programm dazulinken; vergessen Sie aber nicht die Header-Datei mit der Definition der Fensterstruktur. Natürlich steht es Ihnen frei, die Bibliothek Ihren Wünschen anzupassen und zu erweitern - Möglichkeiten gäbe es noch genug ...

Submenüs unter GEM

Uwe Hax

Welcher ATARI ST-Besitzer hat nicht schon einmal neidisch zu anderen Computern, wie zum Beispiel dem AMIGA oder dem Mac, hinübergeschielt, wenn es um die Darstellung von Submenüs ging? Es geht allerdings auch auf dem Atari. Dazu genügen einige wenige Routinen, die zum eigenen Programm hinzugebunden werden müssen.

Die hier vorgestellte Lösung zeigt, wie man unter GEM "echte" Submenüs implementiert. Unter "echten" Submenüs verstehe ich solche, die schon dann herausklappen, wenn der Mauszeiger einen Menüeintrag nur berührt und die automatisch auch wieder verschwinden, wenn sich die Maus weiterbewegt - "echte" Submenüs eben... Um jetzt zuerst einmal die Ungeduldigen und Ungläubigen zufriedenzustellen, sollte ich hier als erstes vielleicht die Vorgehensweise zur Installation der auf der Diskette befindlichen Programme erklären. Sie heißen *MENUDEMO* und *SUBMENU*. Während *SUBMENU* alle für die Darstellung und Verwaltung der Submenüs notwendigen Routinen enthält, stellt *MENUDEMO* lediglich eine Beispielapplikation dar, die eine Menüleiste auf den Bildschirm bringt und auf das Anklicken von Submenüs reagiert, indem sie die Indexnummer des angeklickten Objekts ausgibt. Im folgenden wird anhand dieses Demoprogramms beispielhaft die Konstruktion und Bearbeitung von Submenüs beschrieben.

Zusätzlich zum abgedruckten *MENUDEMO*-Programm sind noch zwei Resource-Files notwendig, die nach folgenden Angaben mit einem RCS konstruiert werden sollten. Auf einen Abdruck der Resource-Files in Form von Strukturen habe ich hier verzichtet, da die Konstruktion der Menüleisten mit einem RCS mit Sicherheit schneller geht als seitenlanges Abtippen, zumal genaue Größen oder ähnliches nicht wichtig sind. Anschließend sind beide Programme, also *MENUDEMO* und *SUBMENU*, zu compilieren und zusammenzulinken.

Was muß nun genau getan werden? Es müssen **zwei Resource-Files** erzeugt werden, die jeweils eine Menüleiste enthalten. Die Betonung liegt dabei auf zwei Resource-Files. Warum das so sein muß, wird weiter unten noch erklärt. Dabei enthält das eine Resource-File die ganz normalen Resourcen des Programms, wie zum Beispiel Dialogboxen und eben auch die ganz normale Menüleiste. Das andere Resource-File hingegen enthält lediglich eine einzige Menüleiste mit allen Submenüs. Das Aussehen der ersten Menüleiste eines fiktiven Programms kann man in Abbildung 3 sehen.

Dies ist die ganz normale Menüleiste mit dem Desk- und zwei weiteren Drop-Down-Menüs. Sie ist im RCS mit dem Namen *MENU* zu versehen und die ganze Datei muß als *MENU.RSC* abgespeichert werden. Die einzelnen Einträge erhalten der Einfachheit halber nach

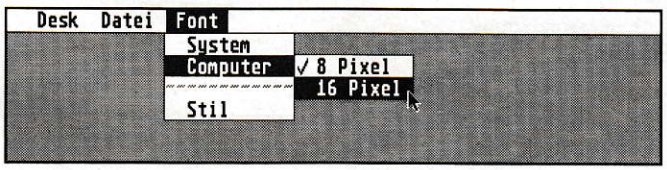


Abbildung 1

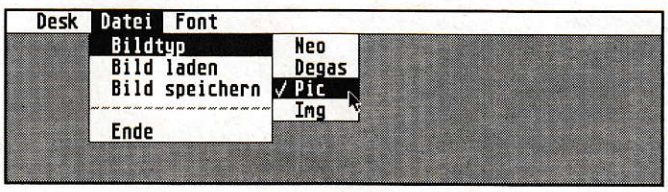


Abbildung 2

Gewußt wie!

Oberflächlich betrachtet: schöne neue Programme. **Interface.** *Der Beste im Test.*

Interface ist der Resource-Editor für Atari ST(E) und TT. Komfortabel, flexibel und ein Freund von Icons. Eine Idee von SHIFT.

Herausragende Features: Unterstützung aller Formate inkl. MS-DOS, Ausgabe als C-Source, Anzeige benutzerdefinierter Objekte durch externe (eigene) Programme, schneller, komfortabler Icon-Editor mit Zeichenfunktionen, Grafikbibliothek und Maskenberechnung, Bedienung per Maus und Tastatur mit fliegenden Dialogen, ...

Interface ist eine 100%ige GEM-Applikation, auflösungsunabhängig und durch eigene Routinen erweiterbar. Zum Lieferumfang gehören C-Bindings, mit denen die Programmierung von GEM-Anwendungen erheblich erleichtert wird.

Das schönste: *„das beste Resource Construction Set"* (siehe ST-COMPUTER 11/91 und TOS 11/91) kostet nur 98 DM (unverb. Preisempfehlung).

SHIFT
KOMPAGNIESTRASSE 13
W-2390 FLENSBURG
☎ (0461) 2 28 28 FAX 1 70 50

shift
Sachen gibt's...

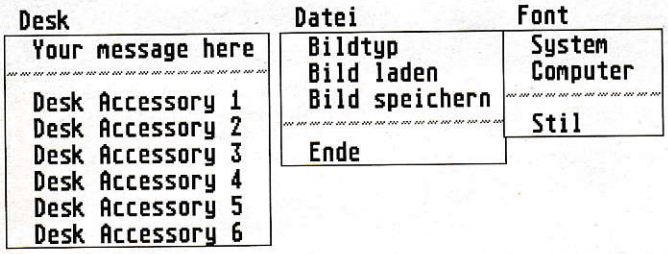


Abb. 3: Die erste Menüleiste mit den aufgeklappten Menüs

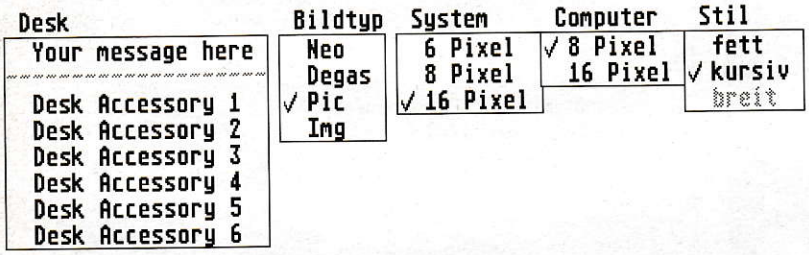


Abb. 4: Die zweite Menüleiste mit allen Submenüs

Möglichkeit die gleichen Namen wie der in ihnen stehende Text, also:

Datei	Font
Bildtyp	System
Laden	Computer
Speicher	Stil
Ende	

Die restlichen Einträge, wie zum Beispiel im Deskmenü, brauchen nicht benannt zu werden, da sie von *MENUDEMO* nicht benutzt werden.

Abbildung 4 zeigt die zweite Menüleiste; sie enthält alle Submenüs. Die Verteilung der Haken und "disabled"-ten Einträge ist dabei willkürlich gewählt. Im Grunde genommen ist wie bei der obigen Menüleiste der gesamte Textinhalt eigentlich völlig belanglos, aber irgendetwas Sinnvolles sollten die Einträge ja nun doch enthalten. Wer will, kann die Menüs ganz nach Lust und Laune gestalten, die Namen der Einträge müssen jedoch die gleichen bleiben!

Diese Menüleiste ist als *SUBMENU* zu definieren und als *SUBMENU.RSC* abzuspeichern. Die Überschriften der Menüs, also die Menütitel, sind eigentlich völlig egal; sie werden nicht benötigt. Um jedoch deutlich zu machen, welcher Menüeintrag in der in Abbildung 3 gezeigten Menüleiste ein Submenü bekommen soll, sollte man hier der Übersicht halber die Titel identisch wählen. Falls man aber mal vor dem Problem stehen sollte, daß man wegen zu langer Menütitel nicht alle Submenüs in eine Zeile bekommen sollte, kann man die Namen natürlich auch beliebig kurz wählen; ein Zeichen reicht völlig aus, da die Titel nirgendwo im späteren Programm erscheinen. Hier jetzt die Namen der obigen Einträge; man beachte dabei, daß die Titel keinen Namen bekommen:

Neo	System6	Comp8	Fett
Degas	System8	Comp16	Kursiv
Pic	System16		Breit
Img			

So, nachdem jetzt die erforderlichen Vorarbeiten, d.h. das Erstellen der Resource-Files, geleistet wurden, müssen nur noch die beiden Programme abgetippt und anschließend compiliert und zusammengelinkt werden. Eigentlich müßte jeder, der bis jetzt noch gezweifelt hat, die Hände über dem Kopf zusammenschlagen. - Entweder, weil die Submenüs so toll sind, oder weil man tatsächlich alles richtig gemacht hat...

So weit, so gut (oder auch schlecht, kommt darauf an...), nun noch die oben versprochene Erklärung, warum man keine zwei Menüleisten in einem Resource-File anlegen kann; d.h. man kann schon, aber dummerweise werden diese vom RCS anschließend in einer Struktur abgespeichert. Auch das wäre im Grunde genommen eigentlich nicht weiter tragisch, wenn nur auch die Indizes zur Adressierung der Objekte umgerechnet werden würden... Da dies aber nun einmal nicht der Fall ist, bleibt eben nichts anderes übrig, als zwei Resource-Files zu verwenden.

Auch hierbei gibt es jedoch wieder einen Haken: "Offiziell" kann GEM lediglich ein Resource-File verwalten; wie man dieses Problem löst, kann dem Listing *MENUDEMO* entnommen werden; eine genaue Beschreibung des Vorgehens ist schon in zahlreichen anderen Artikeln, u.a. auch in der ST-Computer, beschrieben worden.

Deshalb hier nur eine Kurzbeschreibung der Vorgehensweise in *MENUDEMO*: Nach dem Laden des ersten Resource-Files werden diesem einfach mittels *rsrc_gaddr()* die benötigten Adressen entnommen und anschließend die in *global[5]* stehende Adresse des Resource-Files zwischengespeichert. Anschließend kann das nächste Resource-File eingeladen werden. Zur Freigabe des ersten Files wird am Ende des Programms einfach wieder die Adresse zurückgeschrieben.

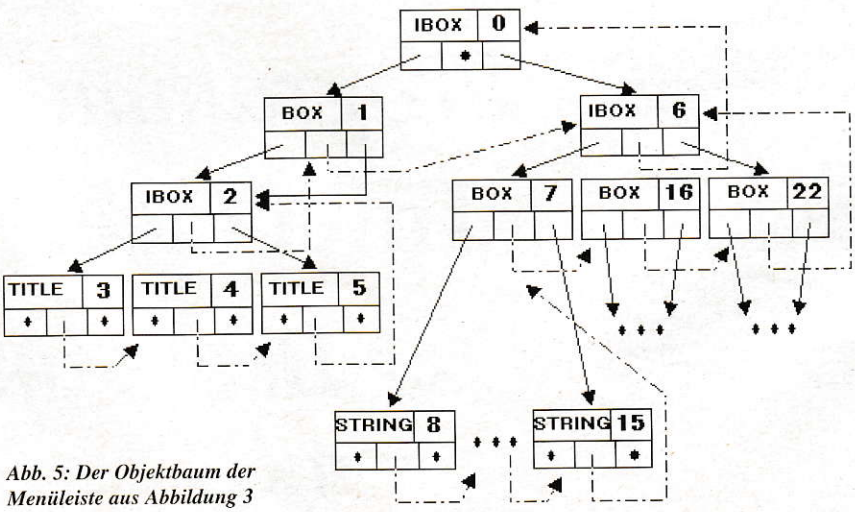


Abb. 5: Der Objektbaum der Menüleiste aus Abbildung 3

Drucker

PL26 29,90

Alles was Sie für Ihren Drucker brauchen ist in diesem Paket auf *10 Disketten* enthalten. Seien es die unterschiedlichsten Treiber, Ausdruckprogramme, Etikettendruckprogramme, Posterdruck, Scheckdruck, Formulardruck ....

Einsteiger

PL19 29,90

Die Standardausrüstung für den Computerneuling oder Anfänger. Von der aktuellsten Textverarbeitung, dem besten Virenkiller, dem neuesten Kopierprogramm, den wichtigsten Utilities bis hin zum entspannenden Spiel ist in diesem Paket auf *6 Disketten* alles enthalten.

Astronomie

PL27 29,90

Wenn Sie sich für Astronomie interessieren, sollten Sie sich dieses Paket zulegen. So ist auf *12 Disks z.B. enthalten:*

Kepler, Astrolabium, Orbit, Sternbild, Planet, Swing-By, Sternzeit, Epfhemeriden II, Astro, Cluster, Sunshin, Weltall, Gnomplot, N-Körper, Skymenu, Sky 2000, Starfinder, Sonneuhr, Kalender, Sternkatalog, Sternuhr, Sternkugel .....

TeX

11 Disketten

PL20 39,90

Die komplette Umsetzung des Satzsystems TeX 3.1 für den ST. Neben TeX selbst enthält das Paket alle Druckertreiber (auch für Laser und Post Script) Fonts, Metafont sowie TeX-Draw: Vektorzeichenprogramm und ZPCAD: CAD-Programm mit Schnittstelle zu TeX.

Midi

jeweils 5 Disketten

PL3 29,90
PL16 29,90
PL28 39,90
PL29 39,90 NEU

Sequenzer laden, AMP auf 10 stellen, Cubase^, Cubeat^, Twenty Four^ oder Twelve* laden und mit unseren PD-Midi-Songs abfahren. Bei den neuen Paketen 28 und 29 liegen die Midi-Files im C-LAB, Twenty-Four-Format und MIDI-Standart vor. Paket 28 enthält ausschließlich deutsche Songs, während Paket 29 ausschließlich englische Songs enthält.

Vector

NEU

5 Disketten

PL30 39,90

Jede Menge Grafiken im CVG- und GEM-Format (Vektorformat). Diese Vektorgrafiken eignen sich besonders für DTP. Die Grafiken wurden alle selbst vektorisiert, so daß Überschneidungen mit anderen Serien ausgeschlossen sein dürften. Einige Beispiele sehen Sie in diesem Kasten.

NEU

Der Katalog

Die, die ihn kennen, wissen ihn zu schätzen. Unseren gedruckten Public-Domain Katalog. Bestellen Sie nicht die Katze im Sack. In unserem Katalog finden Sie die besten Public- Domain Programme thematisch sortiert und gut beschrieben.

5,- DM Schutzgebühr (Briefmarken)

Signum/Script

6 Disketten

PL17 29,90

Dieses Paket ist für Anwender von Signum oder Script zusammengestellt worden. Es enthält jede Menge Grafiken, Zeichensätze und spezielle Tools wie z.B. Funktionstastenbelegung, große Fonts, gedrehte Fonts, Lineal ...

Fonts

je 7 Disks

PL6a 29,90
PL6b 29,90

Diese Pakete enthalten jeweils ca. 100 Signum- bzw. Script-PD-Zeichensätze. Jeder Zeichnsatz liegt für 9-, 24-Nadel und Laserdrucker bei.

Spiele

je 12 Disks

PL21c (s/w) 29,90
PL21d (Farbe) 29,90

Auf jeweils 12 Disketten (s/w - Farbe) erhalten Sie die besten PD-Spiele. Damit sind viele unterhaltsame Stunden garantiert.

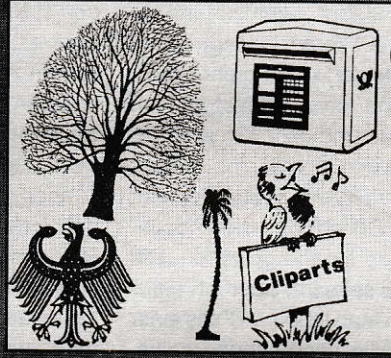
Picto

12 Disks

PL24 39,90

Weit über 500 Pictogramme zu den unterschiedlichsten Themengebieten. Jede Grafik ist im CVG, GEM und IMG-Format abgespeichert.

Cliparts



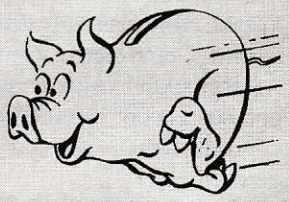
Pac-Grafiken

PL8 29,90
PL14a 29,90
PL18a 29,90

IMG Grafiken

PL14b 39,90
PL18b 39,90
PL18c 39,90

Paket 8, 14a und 18a enthalten jeweils 5 Disketten gefüllt mit Grafiken im PAC-Format zum direkten Einbinden in Signum- oder Scriptdokumente. Die übrigen Pakete (14b, 18b, 18c) enthalten Grafiken im IMG-Format auf jeweils 10 Disks. Die Grafiken wurden alle selber gescannt, so daß Sie in bisherigen PD.-Serien nicht enthalten sein dürften.



Weitere Hard- und Software auf Anfrage. * Ladenlokal in Düsseldorf, Irenenstr. 76c

Ab sofort verwenden wir nur noch **TDK** Bulk Disks

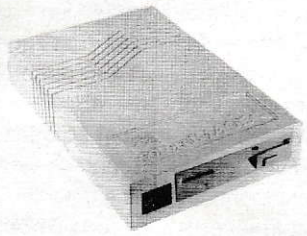
Portfolio

NEU

7 Disketten gefüllt mit Programmen für den Portfolio. An dieser Stelle nur ein paar Beispiele: Disk Tools mit Backup-PRG, Clock, Filter, Adressverwaltung, UP91, VDE152, MMCALC ... Disk Tools 2 mit DBFREAD, UNITIO, VOK-MAN, PORTTOOLS ... / Disk DFÜ mit ACOM, FT, XTERM1, XTERM2 PORTFOLI .... / Disk Grafik mit PGEDIT, PGSHOW, PGCOMP, SNATCH ... / Disk Spiele mit Portris, Tetris, Touch, Spacemen / Disk Basic mit PBASIC v4.1, TBASIC V1.0 / Disk Programm mit FORTH, SMALL-C.

PL30 59,-

Hand-Scanner



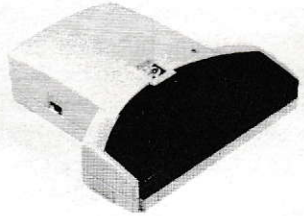
Handscanner 32 Graustufen Bildbearbeitungssoftware Repro Studio ST junior 2.0.

komplett nur L108 498,-

wie oben jedoch zusätzlich noch das Vektorisierungsprogramm Avant-Trace

komplett nur L109 598,-

3,5" -Laufwerk



Komplett anschlußfertig * voll abgeschirmt * atarifarben * 6 Monate Garantie * mit Track-Display

3,5" nur L122 219,-

Vectorfonts

Wir bieten Ihnen Vectorfonts aus eigener Herstellung für Calamus*. Über 200 Vektorfonts zum unglaublich günstigen Preis von

L110 249,-

Für alle die skeptisch sind und sich von der Qualität der Schriften erstmal überzeugen wollen, hier 15 Fonts für nur

Serif **Schnupperpaket L111** 29,-

NEU Nochmals 50 Vectorschriften und 30 Vectorgrafiken für nur

Hobo **Schnupperpaket 2 L124** 49,-

Script F1 **RAHMEN**

Superhigh

*Calamus ist eingetragenes Warenzeichen der Firma DMC.

Hard-, Software

Logi Mouse	**L113**	79,-
Script2	**L114**	278,-
Phoenix	**L115**	378,-
X-Boot	**L116**	69,-
NVDI	**L117**	94,-
Signum3	**L118**	498,-
T S C	**L119**	129,-
TOM Extension Card		
TOS 2.06	**L120**	198,-
Overscan	**L121**	120,-

Rahmen/Zierrat

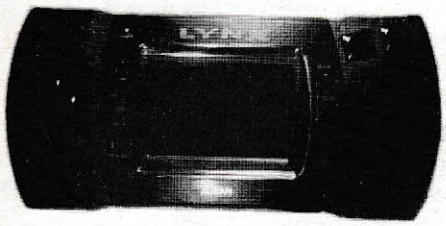
Jeder, der mit DTP oder Textprogrammen arbeitet, die IMG- oder Vectorformate verarbeiten können, werden sich über dieses Paket freuen. Denn jetzt können Sie Ihre Dokumente noch besser gestalten (z.B. Geburtstagskarten, Menuekarten, Plakate, u.v.m.). Alle Grafiken liegen im IMG-, CVG- und GEM-Format vor (insgesamt 10 Disk).

L112 39.90

Lynx - das Spielgenie

NEU

Lynx nur **L123** 199,-



Netzteil 220V Adapter	L124	24,90
Zigarettenanz.	L125	34,90
Tasche klein	L126	24,90
Tasche groß	L127	34,90

Über 30 Spiele vorrätig

Hard Driving (neu)	**L100**	79,-	Checkered Flag (neu)	**L104**	89,-
Turbo Mub (neu)	**L101**	89,-	Tourn. Cyberball(neu)	**L105**	79,-
Scrapyard Dog (neu)	**L102**	89,-	Viking Child (neu)	**L106**	79,-
Awesome Golf (neu)	**L103**	78,-	Ishido (neu)	**L107**	79,-

Bitline
Hard- und Software Vertriebs GmbH
Neikenstr. 2
4053 Jüchen 2

Bitline GmbH ■ Postfach 30 10 33 ■ 4000 Düsseldorf 30 ■ Tel.: 0211/429876

FAX.: 0211/429876 ● BTX.: *WOHL# Versand: Nachnahme = 4,- / Vorrauskasse = 6,- / Ausland (nur Euroscheck) = 12,- (Es gelten unsere allgemeinen Geschäftbedingungen)

Im folgenden nun das weitere Vorgehen nach dem Laden der Resource-Files. Zur Installierung der Submenüs muß ein Aufruf der Form *init_submenus()* gemacht werden. Dabei müssen folgende Parameter übergeben werden: (Siehe Tab.1).

Anschließend kann die Menüleiste dann ganz normal mit *menu_bar()* installiert werden.

Daran schließt sich eine *evnt_multi()*-Schleife an, die bis zum Anklicken des "Ende"-Eintrags durchlaufen wird. In ihr wird auf das Anklicken der Menüeinträge reagiert (MU_MESAG) und in regelmäßigen Abständen (MU_TIMER) die externe Variable *subnum* abgefragt, in der der Index des angeklickten Submenüeintrags zurückgemeldet wird. Dieses Vorgehen ist notwendig, da es sonst keine andere Möglichkeit (zumindest nicht ohne Betriebssystemeingriffe) gibt, den Index zurückzugeben. Dieser Index kann dann zusammen mit der Adresse *submenu* ganz normal zum Zugriff auf das betreffende Resource-File benutzt werden, wie *menu_icheck()* verdeutlicht.

Außerdem ist noch ein weiterer Punkt zu beachten: Wenn man einen Menüeintrag anklickt, wird das betreffende Menü vom Betriebssystem sofort wieder eingeklappt, ohne daß man irgendeine Möglichkeit hätte, dies festzustellen. Deshalb muß bei Anklicken eines Menüeintrags, bei dem ein Submenü herausklappt, das Submenü anschließend wieder vom Bildschirm entfernt werden.
Dazu dient der Aufruf *redraw_bg()*. Eigentlich könnte man annehmen, daß so ein Eintrag sowieso nicht angeklickt wird, weil viel mehr als das Herausklappen eines Submenüs (wow!!!) eigentlich gar nicht mehr passieren kann, aber man glaubt es kaum, was für Leute manchmal so vor einem Computer sitzen...

Einen klitzekleinen Haken hat die Sache mit dem Einklappen der Menüs allerdings noch... Wer an dieser Stelle schon mal ein bißchen mit den Submenüs herumgespielt hat, wird wahrscheinlich festgestellt haben, daß bei Anklikken eines Submenüeintrags mitunter zwar das Submenü vom Bildschirm verschwindet, nicht so jedoch das Menü, so daß man wie gewohnt noch einmal irgendwohin klicken muß, um auch dieses verschwinden zu lassen. Dies ist

typedef struct object

```
{
  int            ob_next;    /* Index-Nr. des nächsten Objektes */
  int            ob_head;    /* Index-Nr. des ersten „Kindes"   */
  int            ob_tail;    /* Index-Nr. des letzten „Kindes"  */
  unsigned int   ob_type;    /* Objekt-Typ, z.B. BOX, TITLE     */
  unsigned int   ob_flags;   /* Objekt-Flags, z.B. SELECTABLE   */
  unsigned int   ob_state;   /* Objekt-Status, z.B. CHECKED     */
  char           *ob_spec;   /* siehe unten                     */
  int            ob_x;       /* x-Koordinate obere linke Ecke   */
  int            ob_y;       /* y-Koordinate obere linke Ecke   */
  int            ob_width;   /* Breite des Objekts              */
  int            ob_height;  /* Höhe des Objekts                */
```

Abb. 6: Definition der OBJECT-Struktur

fast immer dann der Fall, wenn mindestens der dritte Eintrag eines Submenüs angeklickt wird; bei den ersten beiden Einträgen klappt eigentlich immer alles einwandfrei. Woran das liegt, konnte ich bisher (leider) noch nicht ermitteln; dafür habe ich aber festgestellt, daß im Normalfall dieser Effekt nicht mehr auftritt, wenn einmal eine Dialogbox auf dem Bildschirm gestanden hat, sei es nun die Dialogbox eines Accessories oder eine eigene gewesen. Die Sache ist zwar seltsam, aber ich glaube, damit kann man leben... - Falls einer der Leser zufällig herausfinden sollte, aus welchem Grund sich das Betriebssystem so verhält und/oder mir eine Gegenmaßnahme nennen kann, wäre ich für eine entsprechende Mitteilung dankbar.

So, und das war auch schon alles, was der 08/15-Programmierer wissen muß, um die Routinen in *SUBMENU* nutzen zu können. Wie man sieht, sind insgesamt nur zwei Funktionsaufrufe notwendig, um die Submenüs zu handhaben, davon ein Aufruf sogar nur einmal am Anfang des Programms zur Initialisierung. - Einfacher geht's bald nicht mehr! Man sollte sich jedoch tunlichst die globalen Variablen ansehen, die in *SUBMENU* benutzt werden, da diese auf keinen Fall erneut deklariert oder verändert werden dürfen! Wen die Funktionsweise der SUBMENU-Routinen nun nicht weiter interessiert, kann an dieser Stelle aufhören zu lesen und das Heft in die Ecke schmeißen, um sich an den tollen Submenüs zu erfreuen, die jetzt eigentlich schon auf dem Bildschirm zu sehen sein sollten... Für alle anderen erkläre ich ab hier die Arbeitsweise der Submenüroutinen.

Um die Funktionsweise verstehen zu können, ist erst einmal der interne Aufbau einer Menüleiste, so wie ihn das Betriebssystem sieht, wichtig. Als Beispiel benutze ich hier wieder die Menüleiste aus Abbildung 3. Abbildung 5 mag zwar auf den ersten Blick etwas chaotisch und unübersichtlich aussehen, aber wer genauer hinsieht, wird feststellen, daß eigentlich alles ganz geordnet zugeht.

Dem geübten Programmierer sollten die abgebildeten Strukturen sowieso nicht so ganz fremd sein; jedes dieser Rechtecke steht dort stellvertretend für eine Struktur des Typs OBJECT, deren genaue Definition (in "C") man in Abbildung 6 finden kann.

Die einzige verwunderliche Tatsache mag für Leute, die schon mit OBJECTs zu tun hatten, höchstens sein, daß nicht nur Dialogboxen und ähnliche Objekte aus OBJECT-Strukturen aufgebaut werden, sondern eben auch Menüs. Und da diese im Normalfall immer nur aus Text bestehen, wird man dann eben leicht dazu verleitet, zu glauben, daß

1.	handle:	VDI-Gerätenummer
2.	menu:	Adresse der normalen Menüleiste
3.	submenu:	Adresse der zweiten Menüleiste mit den Submenüs
4.	MAX_SUBMENU:	Anzahl der Submenüs
5.	m_index:	Adresse eines Feldes, das in fortlaufender Reihenfolge die Indizes der Menüeinträge enthält, bei denen ein Submenü herausklappen soll.

Tab. 1

etwas anderes als Textdarstellung in Menüs auch gar nicht möglich ist. Wenn man sich die OBJECT-Struktur einmal genauer ansieht, wird man jedoch schnell feststellen, daß sie eine Menge Möglichkeiten zur Manipulation bietet, zum Beispiel braucht man bloß den Objekt-Typ auf BOX zu setzen, *ob_spec* anzupassen, und schon hat man statt eines Textes eben eine BOX im Menü! Eine andere Möglichkeit ist zum Bleistift auch das Einbinden von Grafiken - wer genug Phantasie hat, dem eröffnen sich hier ungeahnte Perspektiven...

Jetzt aber zur Erklärung des oben abgebildeten Menübaumes. Wie bereits erwähnt, steht jedes der abgebildeten Rechtecke für eine OBJECT-Struktur. Abgebildet sind dabei die Einträge *ob_type, ob_head, ob_next* und *ob_tail* (in dieser Reihenfolge). Außerdem habe ich noch zur Veranschaulichung die Indexnummer jedes Objekts hinzugefügt, da die Strukturen in dieser Reihenfolge im Speicher abgelegt werden.
Das Wurzelobjekt mit der Indexnummer 0 und der Bezeichnung IBOX (=Invisible Box) ist dabei ein Rahmen, der den gesamten Rest des Baumes umfaßt und nicht sichtbar ist (wie der Name schon suggeriert). Normalerweise umfaßt der Rahmen den gesamten Bildschirm. Vom Wurzelobjekt ausgehend spaltet sich der Menübaum dann in zwei Teilbäume:

- links *(ob_head)* in den Baum für die Darstellung der eigentlichen Menüzeile; dieser enthält lediglich die Menütitel,
- rechts *(ob_tail)* in den Baum für die Darstellung der Menüs.

Der Eintrag *ob_next* ist hier nicht benutzt und deshalb wie auch alle anderen nicht benutzten Einträge mit einem Punkt versehen. Gehen wir nun systematisch vor und fangen beim linken Teilbaum an: Die Box mit der Indexnummer 1 ist genau der weiße Streifen, der die eigentliche Menüzeile darstellt und sich am oberen Bildschirmrand über die ganze Breite zieht. Wen das schon immer gestört hat, kann hier zum Beispiel die Breite ändern und schon erstreckt sich die Menüleiste beispielsweise nur noch bis zur Mitte des Bildschirms - sieht faszinierend aus (ausprobieren)! Aber nicht vergessen, den rechten Teil der alten Menüzeile mit der Hintergrundfarbe zu übermalen, sonst sieht man nix davon...

Aber weiter im Menübaum: In die Menüzeile müssen natürlich noch die Menütitel gezeichnet werden. Deshalb folgt auf die BOX (Indexnummer 1) noch eine IBOX (2), also wieder ein unsichtbarer Rahmen, der alle zu zeichnenden Menütitel - bezeichnet mit *TITLE* (3-5) - umfaßt. Wie man hier sehr schön sieht, deutet *ob_next* jeweils auf das nächste gleichartige Objekt in der Reihe. Ist kein gleichartiges Objekt mehr vorhanden, zeigt *ob_next* zurück auf das jeweils zugehörige Elternobjekt. Im Grunde genommen kann man zwar nicht von Zeigern sprechen, da in den entsprechenden Einträgen immer nur die Indexnummern der Objekte stehen und nicht Zeiger auf sie, aber vom Prinzip her dürfte klar sein, was gemeint ist.

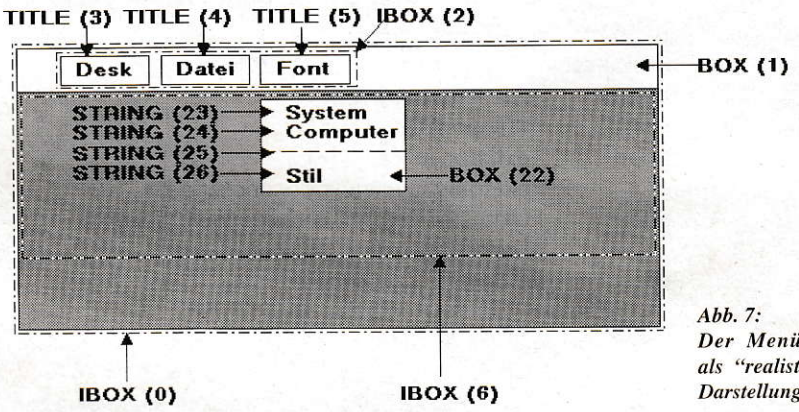


Abb. 7: Der Menübaum als "realistische" Darstellung

Während der linke Teil des Menübaums also für die Darstellung der Menütitel verantwortlich ist, dreht sich im rechten Teil alles um die Darstellung der Menüinhalte. Wie gehabt, beginnt auch hier alles mit einer IBOX (Indexnummer 6), die alle (ausgeklappten) Menüs umfaßt. Von ihr geht - korrespondierend zu den TITLE-Objekten auf der linken Seite - eine Liste aus, die alle BOX-Objekte für die Darstellung der Menüs enthält. Jede dieser Boxen bildet dabei selber wieder ein Wurzelobjekt für eine Liste von Objekten, in diesem Fall (wie auch normalerweise) für die Menü-Texte (STRINGs). Wegen Platzmangel sind die meisten STRINGs in Abbildung 5 lediglich durch Pünktchen angedeutet.

Der gesamte Sachverhalt ist noch einmal so, wie er "in der Realität" auf dem Bildschirm aussieht, in Abbildung 7 zu sehen. Genau an dieser Stelle, nämlich den STRINGs, setzt nun das Verfahren für die Darstellung der Submenüs ein. In jeder OBJECT-Struktur gibt es den bisher nur kurz erwähnten Zeiger *ob_spec*, der je nach Objekt-Typ auf unterschiedliche Strukturen deutet oder auch andere Informationen enthält; bei STRINGs zeigt er auf den auszugebenden Text. Der Vollständigkeit halber habe ich alle Möglichkeiten in Abbildung 8 aufgelistet, benötigt wird hier jedoch nur die Möglichkeit, *ob_spec* auf eine eigene Zeichenroutine zeigen zu lassen (*G_PROGDEF*). Tabelle 1 ist [1] entnommen und an ein paar Stellen mit Hilfe von [2] noch ein bißchen erweitert bzw. korrigiert.

Um nun ein benutzerdefiniertes Objekt zu installieren, muß der Objekt-Typ in *G_PROGDEF* geändert werden und *ob_spec* muß auf einen *APPLBLK* (Application Block) zeigen, der wie in Abbildung 9 definiert ist.

Und genau so - naja, um ehrlich zu sein: ungefähr so - geht die Routine *init_submenus()* vor. Zuerst wird für jedes Submenü eine Struktur angelegt, die ich *ext_appl_blk* (für Extended APPLBLK) genannt habe, da man mit der normalen APPLBLK-Struktur für meine Zwecke nicht genug Parameter übergeben konnte. Dadurch, daß man die Struktur frei im Speicher anlegen kann und sich an fast keine Konventionen halten muß, ist es ohne weiteres möglich, die normale APPLBLK-Struktur beliebig zu vergrößern und zu erweitern; lediglich die ersten beiden Einträge müssen wie bei APPLBLK sein.

Der genaue Aufbau von *ext_appl_blk* kann dem Listing *SUBMENU.C* entnommen werden; es hat sich im Grunde nicht viel getan, außer daß noch zwei

Einträge hinzugekommen sind. Bei dem einen Eintrag *(char *text)* handelt es sich um den geretteten Zeiger auf den Menütext; dieser wird ja auch weiterhin noch benötigt. Der andere Eintrag *(LONG regA4)* ist Megamax-C-spezifisch und muß bei Benutzung anderer Compiler vermutlich angepaßt werden bzw. kann ganz entfallen. In der Anfangsphase der Entwicklung ist mir das Programm nämlich ständig abgestürzt, bis ich herausgefunden hatte, daß bei Aufruf der neuen Zeichenroutine aus dem Betriebssystem heraus fast alle Register des Prozessors geänderte Werte aufweisen, so auch das Adreßregister *A4*. Da dieses vom Megamax-Compiler jedoch zur Adressierung der globalen Variablen benutzt wird, kann man davon ausgehen, daß in so einem Fall das Programm zumeist mit einer Bomben-Stimmung ins Computer-Nirwana eingeht. Deshalb wird bei Initialisierung der Strukturen das *A4*-Register hierher gerettet und später bei Aufruf der Zeichenroutine wiederhergestellt.

Wer also einen Compiler besitzt, der die globalen Variablen über ein anderes Register adressiert, muß dann eben statt des Registers *A4* das entsprechende andere Register nach *regA4* schreiben; wer einen Compiler besitzt, der die globalen Variablen absolut adressiert, kann den Eintrag *regA4* getrost komplett vergessen. Nach der Speicherreservierung für die APPLBLKs folgt eine Zeile, die man vielleicht auch noch erklären sollte, da sie auf den ersten Blick ziemlich dubios aussieht. Ich meine dabei folgende Zeile:

```
submenu_index=submenu [submenu
[submenu[0].ob_tail].ob_head].ob_next;
```

Wenn man sich gleichzeitig mit dieser Zeile aber noch die Abbildung 5 ansieht, wird das Ganze vermutlich wesentlich einfacher. Die Abbildung zeigt zwar den Menübaum und nicht den Baum für die Submenüs, aber Menüleiste ist Menüleiste und hier soll es ja auch nur ums Prinzip gehen; nehmen wir also einfach mal an, der Baum in Abbildung 5 würde die Submenüs beschreiben. In der oben genannten Zeile wird nämlich der Index der ersten Submenü-BOX im Submenübaum ermittelt. Gehen wir einmal systematisch von innen nach außen vor:

1. *submenu[0].ob_tail* zeigt auf den rechten Teil des Menübaums und ergibt so die Indexnummer 6.
2. *submenu[0].ob_tail* ersetzen wir nun durch diese Indexnummer und erhalten so als nächstes Konstrukt: *submenu[6].ob_head*. Das ist der linke Pfeil zur BOX mit der Indexnummer 7.
3. Wiederum eingesetzt erhalten wir jetzt *submenu[7].ob_next* und somit den Zeiger auf die Box für das erste Menü. Eigentlich ist es ja das zweite Menü, aber das erste muß übergangen werden, weil es sich dabei immer um das Deskmenü handelt!

Dieses gerade beschriebene Verfahren muß aus dem Grund angewendet werden, weil je nach Zahl der Menüs die

typedef struct appl_blk

```
{
  int   (*ub_code)(); /* Zeiger auf eine Zeichenroutine */
  long  ub_parm;      /* Parameter, der bei Aufruf der Zeichenroutine
                         übergeben wird  */
} APPLBLK;
```

Abb. 9: Definition der APPLBLK-Struktur

Objekttyp	Bedeutung von ob_spec
G_BOX	Byte 0 und Byte 1: Farbe des Objekts Für die Objektfarbe sind die Bits folgendermaßen belegt: aaaabbbb cdddeeee aaaa: Rahmenfarbe (0-15) bbbb: Textfarbe (0-15) c: Text transparent (0) oder deckend (1) ddd: Füllmuster (0-7) mit ansteigender Dunkelheit (0 kein Füllmuster, 7 solides Muster) eeee: Farbe des Objektinneren Byte 2: enthält die Dicke des Objektrandes: 0 = kein Rand 1-128 = Dicke des Randes in Pixeln, nach innen zählend von den Ecken des Objekts -1-(-127) = Dicke des Randes in Pixeln, nach außen zählend von den Ecken des Objekts Byte 3 = 0
G_TEXT	Zeiger auf die zugehörige TEDINFO-Struktur, in der die Merkmale des Textes festgelegt werden
G_BOXTEXT	wie bei G_TEXT
G_IMAGE	Zeiger auf Struktur vom Typ BITBLK, in der das Bit-Image beschrieben wird
G_PROGDEF	Zeiger auf eine Struktur vom Typ APPLBLK, die das benutzerdefinierte Objekt beschreibt (APPLBLK enthält die Adresse der Routine, die das Objekt zeichnet)
G_IBOX	wie bei G_BOX
G_BUTTON	wie bei G_TEXT
G_BOXCHAR	wie bei G_BOX, aber Byte 3 enthält das darzustellende Zeichen
G_STRING	Zeiger auf den String selbst
G_FTEXT	wie bei G_TEXT
G_FBOXTEXT	wie bei G_TEXT
G_ICON	Zeiger auf eine Struktur vom Typ ICONBLK, in der das Icon genauer beschrieben ist
G_TITLE	Zeiger auf den Text für den Menüeintrag

Abb. 8: Die Bedeutungen von ob_spec

Indexnummern völlig unterschiedlich sein können. Das sicherste Verfahren ist somit das Durchhangeln vom Wurzelobjekt aus.

Im weiteren passiert *in init_submenus()* eigentlich nichts Weltbewegendes mehr. In einer Schleife über alle Submenüs wird nach der Initialisierung der *ext_appl_blk's* zuerst nach dem Index der Menübox gesucht, bei der irgendwo ein Submenü herausklappen soll. Mit Hilfe des gefundenen Indexes werden dann die Koordinaten der Box ermittelt und anschließend wiederum mit deren Hilfe die Koordinaten für die Box des Submenüs neu berechnet, denn das Submenü soll ja schließlich unmittelbar neben dem Menü herausklappen und nicht irgendwo auf dem Bildschirm. Natürlich kann man durch Verändern der Koordinaten das Submenü auch sonstwo auf dem Bildschirm herausklappen lassen, beispielsweise links neben dem Menü. Platz für Spielereien gibt es hier genug, ob das Ganze dann aber noch sinnvoll ist, ist natürlich eine ganz andere Frage... Nach Beendigung der Schleife muß dann nur noch der Zwischenspeicher für den Submenühintergrund angefordert werden; um Speicherplatz zu sparen, wird dieser nicht für jedes Submenü angefordert, sondern nur für das größte.

Damit wäre dann auch schon die Initialisierung und Installation der Submenüs abgeschlossen, und den Rest kann man getrost dem Betriebssystem überlassen, es meldet sich dann schon, wenn es etwas will, zum Beispiel ein Submenü zeichnen...

In so einem Fall wird nämlich die in *ob_spec* eingetragene Adresse angesprungen und die Kontrolle anschließend der damit aufgerufenen Routine, in unserem Fall *draw_submenu()*, überlassen. Und wehe, diese Routine arbeitet nicht vernünftig... - die Effekte reichen von "Bildschirm-Allerlei" bis "ATARI-Gardine".

Damit man bei Aufruf der Zeichenroutine aber nicht ganz allein, einsam, verlassen und sonstwie im Regen (bzw. Bombenhagel) steht, übergibt das Betriebssystem netterweise jedoch noch einen Zeiger auf eine Struktur vom Typ *PARMBLK* (Parameter-Block), die wie in Abbildung 10 definiert ist. In Gedanken höre ich jetzt manche Leute stöhnen: *"Schon wieder eine Struktur!"*, aber in so einem Fall kann ich nur sagen:

typedef struct parm_blk

```
{
  OBJECT  *pb_tree;          /* Zeiger auf den Objektbaum, der das
                                benutzerdefinierte Objekt enthält  */
  int     pb_obj;            /* Indexnummer des Objekts            */
  int     pb_prevstate;      /* vorheriger Status, z.B. NORMAL     */
  int     pb_currstate;      /* neuer Status, z.B. SELECTED        */
  int     pb_x;              /* X-Koordinate des Objekts           */
  int     pb_y;              /* Y-Koordinate des Objekts           */
  int     pb_w;              /* Breite des Objekts                 */
  int     pb_h;              /* Höhe des Objekts                   */
  int     pb_xc;             /* X-Koordinate für das Clipping      */
  int     pb_yc;             /* Y-Koordinate für das Clipping      */
  int     pb_wc;             /* Breite für das Clipping            */
  int     pb_hc;             /* Höhe für das Clipping              */
  long    pb_parm;           /* übergebener Wert ub_parm aus der
                                APPLBLK-Struktur                   */
} PARMBLK;
```

Abb. 10: Definition der PARMBLK-Struktur

"Leute, seid froh, daß Ihr keinen AMIGA besitzt, dort muß man sich selbst für einfache Textausgaben durch einen wahren Dschungel von Strukturen wühlen...!!!" - Ich weiß, wovon ich spreche!

Mit Hilfe dieser PARMBLK-Struktur bzw. ihren Einträgen läßt sich dann auch schon eine ganze Menge anfangen, wie wir gleich sehen werden. Unter anderem kann man damit nämlich auch auf den APPLBLK zugreifen, von dem aus die Zeichenroutine aufgerufen wurde. Wenn *parmblock* der übergebene Zeiger ist, dann enthält *parmblock-> pb_obj* den Index des Menüeintrags, von dem aus die Routine aufgerufen wurde, *parmblock->pb_tree* einen Zeiger auf den Objektbaum (den Menübaum), und mit *parmblock->pb_tree[parmblock -> pb_obj]. ob_spec* erhält man dann den gesuchten Zeiger auf die APPLBLK-Struktur. Dieser Zeiger wird immer dann benötigt, wenn man die normale APPLBLK-Struktur erweitert hat und Zugriff auf die restlichen (neuen) Einträge benötigt.

Vielleicht noch ein paar Anmerkungen zu den PARMBLK-Einträgen *pb_prevstate* und *pb_currstate:* Wenn beide Variablen den gleichen Wert haben, muß das Objekt komplett gezeichnet werden, andernfalls ist nur der Status zu ändern. Erwähnenswert sind auch noch die Clipping-Koordinaten; sie geben normalerweise den Bereich an, in den das Objekt gezeichnet werden soll, bei Aufrufen aus einem Menü heraus (wie hier) sind diese Koordinaten jedoch Null und dürfen deshalb nicht benutzt werden!

Der jeweils erste Aufruf von *draw_submenu()* geschieht in dem Augenblick, in dem man die Maus in die Menüleiste bewegt und dadurch das Menü herausklappt, denn auch das Zeichnen des völlig normalen Texteintrags im Menü muß jetzt selbst übernommen werden, da *ob_spec* ja nicht mehr auf den Text zeigt! Weiter muß in diesem Augenblick nichts getan werden.

Erst wenn der Mauszeiger auf dem Menütext steht, wird *draw_submenu()* erneut aufgerufen, denn in diesem Fall

typedef struct fdbstr

```
{
  long  fd_addr;       /* Adresse des Speicherblocks              */
  int   fd_w;          /* Breite des Speicherblocks in Pixeln     */
  int   fd_h;          /* Höhe des Speicherblocks in Pixeln       */
  int   fd_wdwidth;    /* Breite des Speicherblocks in Worten     */
  int   fd_stand;      /* 0 = Rasterkoordinaten (Standardform)
                          1 = normalisierte Koordinaten (geräte-
                              spezifische Form)                   */
  int   fd_nplanes;    /* Anzahl der Farbebenen                   */
  int   fd_r1;         /* reserviert                              */
  int   fd_r2;         /* reserviert                              */
  int   fd_r3;         /* reserviert                              */
} FDB;
```

Abb. 11: Definition der MFDB-Struktur

muß erstens der Eintrag selektiert und zweitens das Submenü gezeichnet werden. Zur Darstellung der Selektion werden die Objektkoordinaten aus dem PARMBLK ausgelesen und der Routine *switch_entry()* übergeben, die den Text mit einem schwarzen Balken übermalt.

Bevor nun das Submenü gezeichnet werden kann, muß zuerst der Hintergrund gerettet werden, wozu die VDI-Routine *vro_cpyfm()* benutzt wird. Hier an dieser Stelle eine Anmerkung dazu: Vielleicht ist ja schon jemandem aufgefallen, daß in *draw_submenu()* inklusive aller Unterroutinen nirgendwo eine AES-Routine benutzt wird, und das darf auch auf keinen Fall geschehen! Das AES hat nämlich den Nachteil, nicht reentrant zu sein, d.h. bei Aufruf einer AES-Routine aus dem AES heraus kommt es zu einem wahren Bombenhagel; VDI-Routinen können dagegen ohne Einschränkung benutzt werden!

Zurück zu *vro_cpyfm()*: Mit dieser Routine ist es ziemlich einfach möglich, rechteckige Bildschirmbereiche in einen zusammenhängenden Speicherbereich zu transferieren und umgekehrt, ohne daß man sich weiter um das unterschiedliche Format kümmern müßte. Wenn ich dabei sage, "ziemlich einfach", so meine ich damit nichts anderes, als daß man mal wieder einige Strukturen initialisieren muß...

Da *vro_cpyfm()* eine oft viel zu wenig genutzte Funktion ist, weil keiner weiß, wie er sie eigentlich genau zu benutzen hat, möchte ich hier an dieser Stelle genauer darauf eingehen. Die meisten Informationen zu diesem Thema sind übrigens in [1] zu finden, die ich hier, so wie sie in *draw_submenu()* benötigt werden, zusammengefaßt habe.

Als erstes werden wieder einmal zwei Strukturen benötigt, und zwar diesmal vom Typ *MFDB* (Memory Form Definition-Block), der laut Abbildung 11 definiert ist. Das Fehlen des M bei MFDB ist übrigens kein Fehler, sondern die Definition lautet (seltsamerweise) wirklich so.

Da die eine MFDB-Struktur ein Rechteck auf dem Bildschirm beschreiben soll, nennen wir sie mal sinnigerweise *screen*, und die andere, die für einen Speicherbereich zuständig sein soll, dagegen *memory*. In der *screen*-Struktur brauchen wir lediglich den ersten Eintrag zu initialisieren, nämlich *fd_addr*:

```
memory.fd_addr=(LONG)menu_buffer;
memory.fd_wdwidth=sub_menu[sm_index].ob_width/16+1;
memory.fd_stand=0;
memory.fd_nplanes=resolution ? 2/resolution : 4;
```

Tab. 2

```
screen.fd_addr=0L;
```

Weitere Initialisierungen sind hier nicht notwendig. Das Betriebssystem erkennt bei Aufruf von *vro_cpyfm()* nämlich an der Null, daß die Struktur einen Bildschirmausschnitt beschreibt und trägt dann an dieser Stelle automatisch die Adresse des Bildschirms ein und initialisiert auch die restlichen Parameter mit den richtigen Werten.

Bei der *memory*-Struktur sind dagegen schon einige Einträge mehr notwendig: (Siehe Tab. 2).

Menu_buffer ist die Adresse eines in *init_submenus()* angelegten Speicherbereichs zur Zwischenspeicherung der Submenühintergründe und *submenu[sm_index].ob_width* ist die Breite des Submenüs in Pixeln. Da jedoch die Breite in Worten, also 16 Bits, benötigt wird, ist die oben vorgenommene Umrechnung notwendig.

Fd_stand ist grundsätzlich Null bei Standard-, d.h. Rasterkoordinaten, die beim ST normalerweise benutzt werden, es sei denn, es wird zum Beispiel mit solchen Programmen wie GDOS gebootet, dann muß *fd_stand* eine 1 enthalten. Leider reicht es jedoch nach meinen Erfahrungen in so einem Fall nicht aus, in *fd_stand* einfach eine 1 hineinzuschreiben; ganz offensichtlich wären auch noch zahlreiche andere Änderungen am Programm notwendig, deshalb funktioniert die abgedruckte Fassung **nicht** mit GDOS!

Fd_nplanes muß die Anzahl der Bitplanes enthalten, die benutzt werden; in hoher Auflösung ist dies nur eine, in mittlerer Auflösung zwei und in niedriger Auflösung vier. Da *Getrez()* jedoch die Werte 2, 1 und 0 zurückgibt, müssen auch diese entsprechend umgerechnet werden.

Wie man sieht, werden auch hier nicht alle Einträge der MFDB-Struktur initialisiert; da man an *vro_cpyfm()* noch ein Feld mit den Quell- und Zielkoordinaten *(copy_array)* übergeben muß, ist dies auch nicht notwendig, weil einem das Betriebssystem mal wieder die halbe Arbeit abnimmt und die restlichen Einträge selbst initialisiert. Dabei enthalten *copy_array[0]* und *copy_array[1]* die Koordinaten der linken oberen Ecke und *copy_array[2]* und *copy_array[3]* die Koordinaten der rechten unteren Ecke des zu rettenden Bildschirmausschnitts. Für den Zielbereich kann auf Koordinaten verzichtet werden, denn hierbei handelt es sich ja um einen fortlaufenden Speicherbereich und zur späteren Restaurierung werden deshalb nur die Breite und Höhe des Ausschnitts benötigt.

Damit sind alle Werte korrekt initialisiert und einer Benutzung von *vro_cpyfm()* steht nichts mehr im Wege. Um den Hintergrund wieder zu restaurieren, müssen lediglich die ersten vier Werte von *copy_array* mit den letzten vier Werten vertauscht werden. Und beim Aufruf von *vro_cpyfm()* müssen natürlich auch die letzten beiden Parameter, also Quell- und Zieladresse, gewechselt werden. Dies macht übrigens die Routine *redraw_bg()*, die - wie oben schon erwähnt - mitunter auch vom Programmierer direkt aufgerufen werden muß.

Nachdem jetzt endlich der Hintergrund gerettet wurde (wurde auch langsam Zeit, lange hätte er bestimmt nicht mehr durchgehalten), kann das Submenü gezeichnet werden. Anschließend werden dann die Texte hineingeschrieben, wobei die Flags CHECKED und DISABLED beachtet werden, denn üblicherweise wird doch nicht mehr benutzt. Wer unbedingt noch Einträge haben will, die OUTLINED, SHADOWED oder sonstwas sind, soll gefälligst selber noch ein bißchen an *draw_submenu()* herumbasteln; ich bin von der Notwendigkeit jedenfalls nicht überzeugt, denn weniger ist oft mehr!

Nach dem vollständigen Zeichnen des Submenüs kann jedenfalls *draw_submenu()* auch schon wieder beendet werden, nur um bei der nächsten Mausbewegung dann wieder aufgerufen zu werden... - aber das ist eine andere Geschichte. Erwähnen sollte ich wohl noch, daß der Status des Objekts, also NORMAL oder SELECTED, unbedingt wieder an das Betriebssystem zurückgegeben werden muß, wenn man kein Chaos in den Menüs provozieren will! Der ganze Ablauf von *draw_submenu()*

ist noch einmal in Abbildung 12 als Struktogramm zu sehen.

Kommen wir also zu der anderen Geschichte: Was passiert, wenn die Maus sich bewegt? *Annahme:* Sie wird von der Katze gefressen... *Behauptung:* falsch! *Beweis:* Wenn die Maus sich bewegt, wird sie ausgeschaltet (zumindest kurzfristig) und verschwindet so selbst für die guten Augen einer Katze... Aber von der Welt der Katzen und Mäuse nun wieder zurück zur Welt der Mäuse und Submenüs: Sobald die Maus bewegt wird, wird *draw_submenu()* erneut aufgerufen; ist der Mauszeiger immer noch im gleichen Menüeintrag, passiert überhaupt nichts und die Routine wird wieder verlassen. Hat der Zeiger jedoch den Eintrag verlassen, muß zuerst überprüft werden, ob er sich innerhalb des Submenüs befindet; wenn ja, wird *do_submenu()* aufgerufen und übernimmt die Verwaltung des Submenüs. Andernfalls wird das Submenü wieder "eingeklappt", indem einfach der Hintergrund restauriert wird und anschließend müssen dann nur noch der zugehörige Menüeintrag wieder normalisiert und die Routine erneut verlassen werden.

Jetzt stellt sich vermutlich nur noch eine Frage, die den geneigten Leser interessieren dürfte: Wie funktionuckelt die Verwaltung eines Submenüs? Die Antwort: denkbar einfach. Betrachten wir uns also zu guter Letzt noch die Routine *do_submenu()*. Sie wird ausgeführt, solange sich der Mauszeiger innerhalb eines Submenüs befindet; verläßt die Maus auch nur einen Moment lang das Submenü, wird auch die Routine verlassen und das Submenü wieder eingeklappt.

Gehen wir also davon aus, der Zeiger befindet sich innerhalb des Submenüs. Dann werden als erstes die Y-Koordinaten aller Einträge mit der Y-Position des Mauszeigers verglichen, um herauszufinden, über welchem Eintrag die Maus gerade "schwebt". Wurde der Eintrag gefunden, wird er noch mit dem letzten angewählten verglichen, um festzustellen, ob die Maus einen neuen Eintrag angewählt hat.

Wenn ja, muß der zuvor angewählte Eintrag normalisiert werden, sofern er nicht DISABLED ist, und anschließend wird dann der neu angewählte Eintrag selektiert. Jetzt fehlt nur noch die Überprüfung, ob eine Maustaste gedrückt wurde, und falls ja, wird der Index des gerade angewählten Eintrags in *subnum* übergeben und *do_submenu()* verlassen. Dabei wird jedoch nicht auf eine bestimmte Maustaste getestet, so daß man innerhalb eines Submenüs auch die rechte Taste benutzen kann. Die ganze Routine ist im Überblick noch einmal in Abbildung 13 als Struktogramm zu sehen.

So, und damit wäre ich so ziemlich am Ende angelangt, oder genauer gesagt: ganz am Ende (oder auch: fix und fertig). Jedenfalls sollte es jetzt so ziemlich jedem absolut klar sein, wie das Prinzip der Submenüs aussieht und wie man eigene benutzerdefinierte Objekte in Menüs anlegt. Und ich möchte ab sofort keine Programme mit völlig überladenen Menüs mehr sehen, wo doch jetzt Submenüs auch unter GEM möglich sind! Rein vom Prinzip her müßten Submenüs eigentlich auch auf GEM für PCs möglich sein; ich kann mir jedenfalls nicht vorstellen, daß man dort auf benutzerdefinierte Objekte verzichtet hat, aber auf diesem Gebiet kenne ich mich leider nicht so gut aus; das überlasse ich gerne anderen.

Literatur:

[1] H. Lemcke, V. Dittmar, M. Sommer: Programmierlexikon für den ATARI ST. Hüthig Verlag

[2] H.-D. Jankowski, D. Rabich, Julian F. Reschke: ATARI ST Profibuch Sybex Verlag

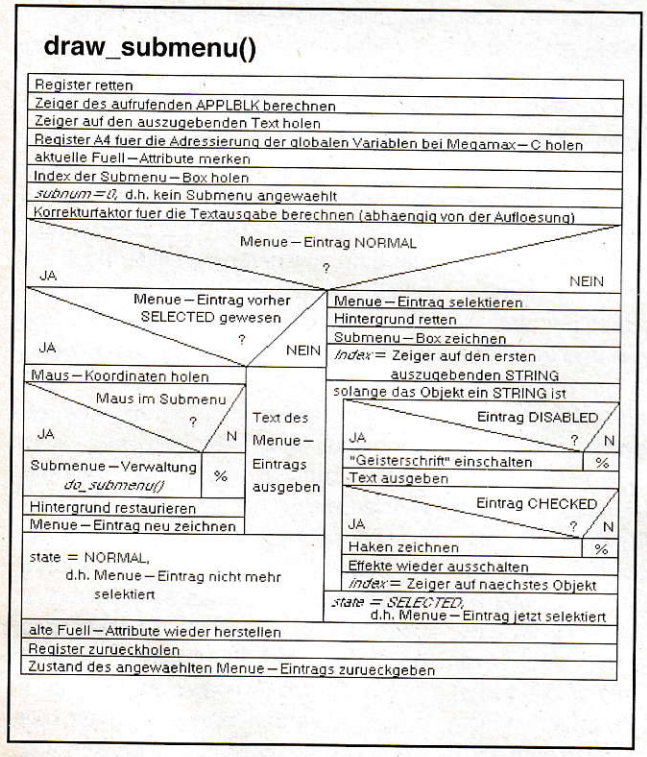


Abb. 12: Struktogramm von draw_submenu()

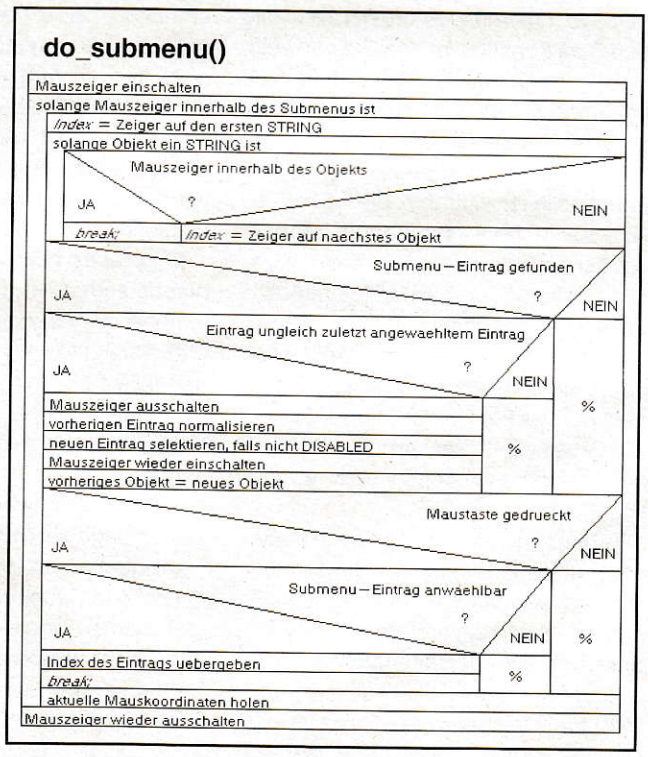


Abb. 13: Struktogramm von do_submenu()

weitere Hard-und Software Produkte

ATARI®- Produkte

1040 STE, 1 MB	798,-
1040 STE, 4 MB	1198,-
MEGA STE1, 1 MB	1398,-
MEGA STE1 + 48 MB HD	1898,-
Monitor SM 144	348,-
Monitor SM 124	288,-
Portfolio	379,-

FESTPLATTEN

HD KIT's + Contr. für MEGA STE

.. 48 MB Seagate	489,-
.. 52 MB Quantum	798,-
..105 MB Quantum	1129,-
..210 MB Quantum	1898,-
..320 MB - 520 MB	a.A.

Externe Festplatten anschlußfertig mit Software

..TUM 48 MB Seagate	848,-
..USD 52 MB Quantum	1099,-
..USD 105 MB Quantum	1479,-
..USD 210 MB Quantum	2149,-

2 Jahre Garantie auf Quantum Laufwerke

EMULATOREN

AT-Speed C16	449,-
Vortex 386SX	777,-
Spectre GCR (ohne ROM's)	629,-

SOFTWARE MS-DOS®

MS-DOS 5.0	269,--
WORD 5.5	1398,--
AutoSketch, CAD	498,--
Windows 3.0	349,--

Mäuse

That's a Mouse	79,-
Logimouse ST	75,-
Logimouse Amiga	75,-
Logimouse PC	85,-

Speichererweiterungen

..Micro RAM für 260/520/1040 ST

auf 2.5 MB, teilsteckbar	398,-
auf 2.5 MB, vollsteckbar	444,-
auf 4 MB, teilsteckbar	589,-
auf 4 MB, vollsteckbar	633,-

..für 1040 STE/MEGA STE

auf 2 MB	198,-
auf 4 MB	395,-

TDK BULK Disketten

(ohne Label, ohne Shutterdruck)

50 Stück MF2DD	55,-
50 Stück MF2HD	110,-

Versand- und Lieferbedingungen

Verkauf solange Vorat reicht. Irrtum und Preisänderung vorbehalten. Versand erfolgt per Vorkasse (PD's: 4 DM/HS 8 DM) oder Nachnahme (PD's 6 DM/HS 12 DM) zzgl. Versandkosten. Ausland nur Vorkasse (Euroscheck, PD's 10 DM/HS 25 DM).

ATARI, Calamus und MS-DOS sind eingetragene Warenzeichen.

Peripherie

..Laufwerke

3.5" Floppy, extern	219,-
3.5" Floppy, intern	298,-
5.25" Floppy extern	269,-

..Drucker

Panasonic KXP-1123	549,-
Einzelblatteinzug 1123	329,-
Canon Bubble Jet 300	1198,-

Tintenstrahldrucker, 360 DPI, 64 Düsen
Endlospapier, Epson LQ kompatibel!

Einzelblatteinzug BJ 300	249,-

vollautomatisch, Endlospapierfunktion ist weiterhin möglich!

2ter Einzelblatteinzug	159,-

für BJ-300, nur in Verbindung mit erstem Einzelblatteinzug möglich

..Scanner

CHARLIE Handyscanner	548,-

32 Graustufen, 400 Dpi
Farbbildverarbeitung optional

ATARI®- DTP Paket

1040 STE Computer, 2 MB RAM, SM124
DTP Software Calamus 1.09N und
Textverarbeitung That's Write V1.45
unser Preis nur 1498,-

DTP

..Software

Calamus V1.09N	398,-
Calamus SL	1349,-
ArtWorks Business	398,-

Gestaltungshilfen zu Calamus

Design Studio 'à la carte'	149,-

Ornamente, Rahmen, Vektorgrafiken speziell für die Gestaltung von Speisekarten

Vektorgrafiken	249,-

über 750 Vektorgraf. (*.GEM)

..Hardware

ProScreen TT	1898,-

19 " Mon. für TT

Proscreen STE	2698,-

19" Mon. VME für MEGA STE
Wir konfigurieren Ihnen Ihre komplette DTP Anlage.
Rufen Sie uns an.

Knüller

Signum 3	498,-
Tempus Word	500,-
T's Write 1.45	99,-
Adimens 3.1+	149,-
Easy Base	239,-
Draw 3.0	129,-
K-Spread 4	235,-
Pure C	398,-
Kobold	79,-
FCopy Pro	79,-
ST-Profi-Buch	79,-

Neue Fonts für Ihren Calamus® Fontpakete

Bodona (2)	Americano (4)
Capitol (2)	Alt Berlin bold (2)
Cochin	CARDPLAY (2)
Futur	Funny Bunny (1)
Garamont (2)	Florence
Bridget reg.	Melody (2)
Elan light	Malaga
Gate reg.	Isabelle Bold
Parisienne (2)	Novo bold
Pisa rounded	PAINTCUT

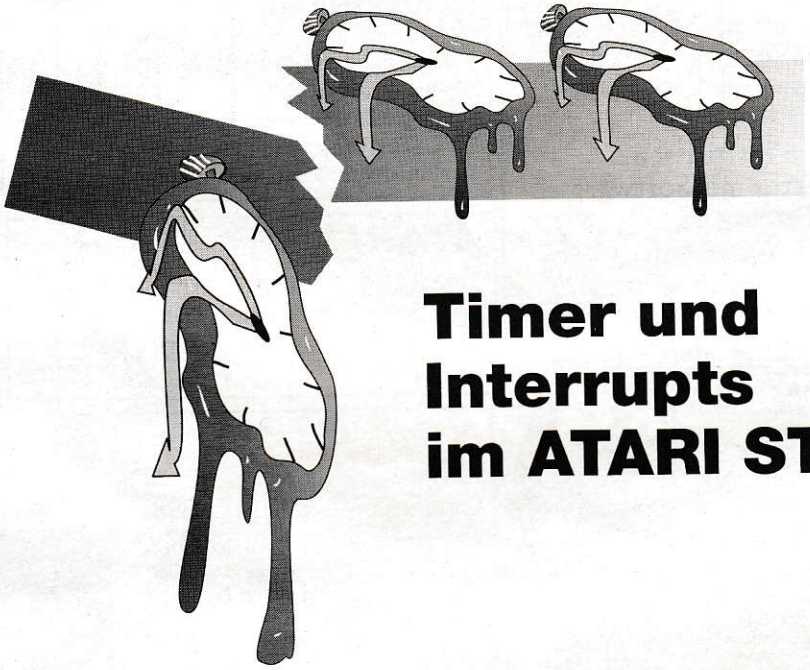
je Paket nur 179,-
in Klammern die Anzahl der Schnitte

IHR ST-TEAM PARTNER

T.U.M.
Soft- & Hardware GbR
T.Helfers * U.Jeddeloh
Hauptstr. 67/PF. 1105
2905 Edewecht
☎04405/6809 Fax: 228

Duffner Computer
Waldkircher Str. 61
7800 Freiburg
☎0761/515550
FAX: 0761/5155530

Wem die Stunde schlägt ...



Timer und Interrupts im ATARI ST

M. Schumacher

Das neudeutsche Wort 'Timer' dürfte wohl jedem Computerbesitzer vom Hörensagen bekannt sein; es bezeichnet ein elektronisches Bauteil, das nach Ablauf einer (hard- oder softwaremäßig) vorgegebenen Zeitdauer einen elektronischen Impuls aussendet. Wenn man so will, ist ein Timer also nichts anderes als ein 'beleuchtungsfreies Blinklicht'.

Die Anwendungsmöglichkeiten reichen von der Meß- und Regelungstechnik bis hin zur elektronischen Tonerzeugung im Synthesizer. Was aber sucht dieses ominöse Teil im ST? Um diese Frage zu beantworten, müssen wir uns in den Alltag eines Betriebssystems versetzen. Dieser beginnt gewöhnlich mit dem Einschalten des Rechners, woraufhin das Betriebssystem naturgemäß seine Gemächer bezieht, alle für den geordneten Betrieb notwendigen Systemvariablen und Peripheriegeräte initialisiert und anschließend auf irgendeine Art und Weise dem Benutzer mitteilt, er könne jetzt mit seiner Arbeit beginnen. Startet dieser ein Programm, so gibt das Betriebssystem die Kontrolle an selbiges weiter und wartet darauf, daß es wieder aktiviert wird und den nächsten Befehl ausführen darf ... In der Steinzeit der „Computer'" (vor etwa 20 Jahren), als die Bits noch in die Röhre schauten, da war das tatsächlich so. Aber heute würde jedes Betriebssystem, das etwas auf sich hält, bei dem Vorwurf, bloß die Schnittstelle zwischen Mensch und Maschine zu sein, vor lauter Wut seine Arbeit einstellen. Bietet es dem geplagten Software-Engineer (früher hieß das mal Programmierer) doch eine ganze Menge Erleichterungen: Wer kümmert sich um die Programmierung der Controller für die Peripherie? Wer um die Speicherplatzverwaltung? Fensterln und 'Mausen' (rein grafisch natürlich!)? Kein Problem dank GEM & Co. Und schließlich die Krönung: Wie aus gut unterrichteten Kreisen verlautbarte, soll es sogar Rechner geben, an denen mehrere Personen und sogar jeweils mehrere Programme gleichzeitig arbeiten können! Und wer, glauben Sie, organisiert dieses Chaos? Richtig: das Betriebssystem. Und das geht eben nur, wenn das System ständig die Kontrolle über den Rechner, sich selbst und natürlich über die Anwenderprogramme hat. Wenn die Maschine aber leider nur eine Ceh-Peh-Uh (nein, nicht Erzwo-Dezwo!) hat und die Kontrolle nur leichtfertig an ein Benutzerprogramm abtritt: Wie soll ein Betriebssystem dann in der Lage sein, die

Wacht zu halten? Dazu braucht man Timer! Die nämlich machen den Prozessor in kurzen Abständen höflich, aber bestimmt, darauf aufmerksam, doch bitte wieder das Betriebssystem anzustoßen, damit Ordnung herrsche im Lande der Busse und Leiterbahnen. Doch nicht nur hier werden diese 'Wekker' gebraucht; im Prinzip lassen sie sich überall dort gewinnbringend einsetzen, wo bestimmte Aufgaben periodisch ausgeführt werden müssen: vom Blinken des Cursors bis hin zum Überprüfen von Tastatur und Floppy.

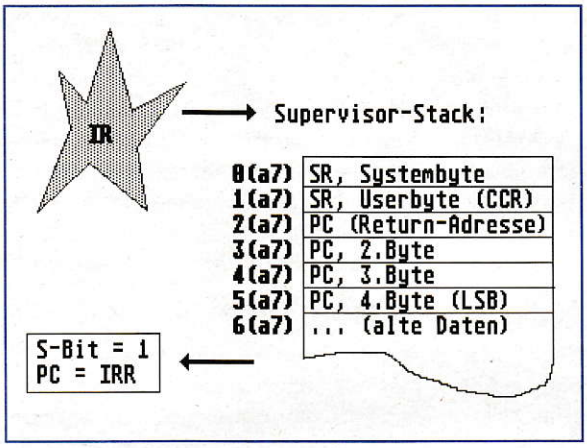


Bild 1: Nach Auftreten eines Interrupts (IR) legt die MC68k Statusregister (SR) und Programmzähler (PC) auf den Supervisorstack, geht in den S-Mode und springt anschließend in die Interrupt-Routine (IRR).

Exceptions, Traps und Interrupts

Nach diesem nicht allzu ernst gemeinten Trip in die Welt der Betriebs-Software nun zum Trap, denn wir wollen ja wissen, wie diese Vorgänge realisiert werden. Eine der größten Stärken des 68000er, so steht es geschrieben, sei seine Fähigkeit, Interrupts zu verwalten. In der Tat stehen dem System 256 Vektoren zur Verfügung, die in Ausnahmefällen dem Rechner und dem Betriebssystem das Leben retten können! Im allgemeinen laufen Anwenderprogramme im sog. User(U)-Mode; hier sind einige der Maschinenbefehle nicht erlaubt, und auch innerhalb des Speichers gibt es 'Sperrbezirke', die für Programme tabu sind. Diese Maßnahmen sollen dazu dienen, das Betriebssystem gegen unbefugten Zugriff und damit gegen 'Absturz' zu schützen.

Im MOTOROLA-Jargon ist alles, was die CPU in den Supervisor-Modus (S-Mode) versetzt, eine Ausnahme (engl.: exception). Das soll nicht etwa darauf hinweisen, daß so etwas selten vorkäme, sondern darauf, daß irgendjemand irgendetwas vom Betriebssystem erwartet, oder daß irgendwo irgendetwas irgendwie schiefgegangen ist. Genaugenommen muß man also zwischen zwei verschiedenen Exceptions unterscheiden: den Traps, die prinzipiell von der Software explizit durch Befehl oder aber durch Laufzeitfehler (Division durch Null etc.) ausgelöst, und den Interrupts, die ausschließlich von der Hardware erzeugt werden.

Im ST ist das bekanntermaßen so gelöst, daß der Anwender mittels TRAP-Befehl die Funktionen des TOS anfordern kann, während er normalerweise mit Interrupts (Interrupt) überhaupt nichts zu tun hat. Diese Arbeit übernimmt glücklicherweise allein das Betriebssystem.

Von IPL ...

Was kann denn nun alles einen Interrupt erzeugen? Z.B.: ein Monitorwechsel, volle Sende- und Empfangspuffer der diversen Schnittstellen, Floppy- und Harddisk-Controller und: die Timer! Was sich beim Auftreten einer Exception abspielt, wird in Bild 1 dargestellt. Zwei Ausnahmen bei den Ausnahmen: Beim Auftreten eines Bus- oder Adreßfehlers legt das Mikroprogramm der CPU noch ein paar Bytes mehr auf den Supervisor-Stack, um eine effektivere Post-Mortem-Analyse (Fehleranalyse nach Betriebssystemabsturz) zu ermöglichen. Aber das nur nebenbei. Stellen wir uns nun vor, die RS232-Schnittstelle hätte beim Empfangen 'den Kanal voll' und könnte keine weiteren Daten mehr empfangen. Bevor sie im Strom der Bits versackt, verursacht sie einen Interrupt, um damit die CPU davon zu überzeugen, daß ein Abholen der Daten wünschenswert sei. In diesem Moment trifft das Ereignis eines Zeilenrücklaufes ein, das ebenfalls der CPU gemeldet wird. Aus einer Laune heraus schiebt der Benutzer die Maus ein paar Zentimeter über den Tisch und der Drucker bekommt - nachdem er gerade ein Listing verdaut hat - wieder Hunger auf Daten. Wenn dann auch noch der Floppy-Controller die erfolgreiche Abarbeitung eines zuvor abgesetzten Kommandos bekanntgibt, ist das Chaos perfekt. Die serielle Schnittstelle wurde zwischenzeitlich ein Opfer der DFÜ, der Elektronenstrahl ist bereits wiederholt zurückgelaufen, ohne daß es irgendjemand interessiert hätte, die Maus steht immer noch auf dem Papierkorb statt auf Disk A, der Drucker ist verhungert, und der Controller schmollt (in Anbetracht solcher Ignoranz!) neben dem Sound-Chip vor sich hin. Klarer Fall: So geht's nicht! Zwei Möglichkeiten bieten sich an: Entweder die CPU fragt im Ringelreihen (Round Robin-Verfahren) alle erdenklichen Interrupt-Quellen ab (Polling) und entscheidet dann von Fall zu Fall, ob eine Verarbeitung erforderlich ist, oder man gibt jeder Interrupt-Quelle eine Zahl mit auf den Weg, die angibt, wie wichtig ihre Wortmeldung, also die Interrupt-Anforderung (IRQ, Interrupt Request), ist; man nennt dies Prioritätensteuerung. Im ST sind beide Formen realisiert; der 68000er kennt aber nur das Prioritätenprinzip. Hierzu verfügt die CPU über die drei Eingänge IPL0, IPL1

Nr.:	Signal
0	Busy (Centronics)
1	DCD (RS232)
2	CTS (RS232)
3	n. b.
4	Timer D (für Baudrate)
5	Timer C (200Hz, System)
6	ACIAs (Keyboard/MIDI)
7	FDC und DMA
8	Timer B (Zeilenrücklauf)
9	Sendefehler (RS232)
10	Sendepuffer leer (RS232)
11	Empfangsfehler (RS232)
12	Empfangspuffer voll (RS232)
13	Timer A (frei verfügbar)
14	RI (RS232)
15	Monochrom Monitor

Tab. 1: Die 16 Interruptkanäle des MFP 68901: Kanal 0 besitzt geringste, Kanal 15 höchste Priorität

und IPL2. Die Abkürzungen stehen für Interrupt Priority Level, also Unterbrechungsprioritätenebene (bleiben wir doch lieber bei IPL ...). Tritt ein Interrupt-Ereignis ein, liegt die jeweilige Priorität an diesen Pins an. Die Bits 8 bis 10 im Statusregister des Prozessors bilden nun die Interrupt-Maske. Mit ihnen hat es folgende Bewandtnis: Steht die Maske z.B. auf 4, können nur Interrupts erkannt werden, deren Priorität größer oder gleich 4 ist. Setzt man diese Überlegung gewissenhaft fort, folgt hieraus, daß bei einer Maske von 0 alle Interrupts erlaubt sind, während eine 7 alle anderen Interrupts, außer der Stufe 7, sperrt. Da man Interrupts auf diesem Level nicht ausmaskieren kann, spricht man hier auch vom Non Maskable Interrupt (NMI). Im ST ist die Leitung IPL0 permanent auf logisch '0' gesetzt, so daß nur noch die Levels 2, 4 und 6 zur Verfügung stehen.

Auf Stufe 2 erscheint der HBL(Horizont Blank)-Interrupt. Wie oben bereits erwähnt, ist er so unwichtig, daß man sich fragen darf, weshalb er der CPU überhaupt 'Guten Tag' sagen darf. Er entsteht bei jedem Zeilenrücklauf und hat sinnvollerweise die einzige Aufgabe, die Interrupt-Maske auf 3 zu setzen, um weitere HBL-Interrupts zu unterbinden. Vergessen wir ihn also schnell wieder.

Die nächste beim ST erreichbare Interrupt-Stufe ist 4. Hier meldet sich der VBL(Vertikal Bank)-Interrupt, und zwar bei jedem Bildrücklauf (1/70 s). Im ST ist dann etliches zu tun: Test auf Monitorwechsel, evtl. Veränderung der Auflösung und der Farbpalette, Blinken des Cursors, evtl. Ändern der Bildschirm-Anfangsadresse, Test auf Diskettenwechsel usw. Zwar kann man ohne größeren Aufwand noch einige Routinen einflicken, aber schon rein aufgrund der zeitlichen Begrenzung steht fest, daß man hier nur kleinere Arbeiten ausführen lassen kann.

... über den MFP ...

Die höchste (ST-)Stufe ist 6. Hier residiert der MFP. Der MFP 68901 ist ein MehrFunktions-Prozessor(MFP), der im ST als Interrupt-Controller zum Einsatz kommt. Er kann 16 Interrupts erzeugen, die alle gegeneinander priorisiert sind, teilweise auch durch Polling abgefragt werden. Tabelle 1 zeigt eine Auflistung dieser Interrupt-Quellen. Für uns ist genau eine dieser Quellen von besonderer Bedeutung: der Timer A. Von den insgesamt 4 im MFP enthaltenen Timern ist er der einzige, den man den Programmierern noch für eigene Zwekke gelassen hat; alle anderen dienen systeminternen Zwecken und müssen unangetastet bleiben, so man den Griff zum Reset-Knopf vermeiden möchte ...

... zum Timer A

Wie eingangs erwähnt, erzeugt ein Timer periodisch Signale, die natürlich auch als Interrupt dienen können. Nun wäre das aber nur halb so interessant, wenn die Zeit zwischen zwei solchen Signalen nicht einstellbar wäre. Wir benötigen also schon einmal ein Datenregister. Einfacherweise wird der Wert in diesem Register kontinuierlich dekrementiert, bis er bei 0 angelangt ist. Just in diesem Moment soll ein Interrupt erfolgen und der Zähler wieder zurückgesetzt werden. Abschalten muß man den Timer natürlich auch können; allein mit dem Datenregister ist das nicht möglich, denn selbst bei Eingabe einer 0 wird solange dekremenetiert, bis wieder die 0 kommt. Und das ist bei einem 8Bit-Register nicht sehr lange. Man benötigt also noch ein Kontrollregister. Wir definieren (der MFP tut das im übrigen genauso!) einen Timer als ausgeschaltet, wenn in seinem Kontrollregister eine 0 steht. Ein von 0 verschiedener Wert aktiviert also den Timer; er erfüllt im MFP sogar noch einen weiteren Zweck; je nachdem, welcher Wert gesetzt ist, müssen entsprechend viele Taktsignale am MFP ankommen, bis der Wert im Datenregister dekrementiert wird (Vorteilung!), oder die Betriebsart des Timers wird geändert (Count And Jump - also so wie oben - oder Zählen externer Impulse bzw. von deren Länge). Die letzten beiden Modi sind für uns uninteressant, weil sie vorab den Gebrauch eines Lötkolbens voraussetzen; für den ersten Modus steht in Tabelle 2 der Vorteilungsfaktor in Abhängigkeit vom Inhalt des Kontrollregisters.

Wert:	Vorteilung
0	Timer aus
1	4
2	10
3	16
4	50
5	64
6	100
7	200

Tab. 2:
Der Wert des Kontrollregisters bestimmt die Vorteilung von Timer A

Jetzt können wir also den Timer ein- und ausschalten. Damit der MFP aber überhaupt weiß, daß, was da aus Richtung Timer ankommt, auch ein Interrupt ist, müssen wir ihm das explizit beibringen, und zwar, indem wir in einem weiteren Register (Interrupt-Enable) ein Bit setzen. Nein, nicht irgendeines; die Nummer 5 muß es sein, denn nur sie schaltet den Timer A zum MFP durch! Die XBTIMER()-Funktion setzt dieses Bit automatisch; programmiert man von Hand zu Fuß, gilt; vergißt man dieses Detail, empfehlen sich Baldriantropfen zur Beruhigung ...

Eine weitere Besonderheit, die es zu beachten gilt: Wird im MFP ein Interrupt erzeugt (egal durch wen), wird das zum Verursacher gehörende Bit im Interrupt-Service-Register gesetzt. Dadurch werden alle Interrupts niedriger Priorität ausmaskiert, also nicht mehr erkannt. Aus Gründen des Fairplays gebietet es sich, das besagte Bit nach Beenden der Interrupt-Routine (IORR) wieder auf 0 zu setzen! Für den Timer A ist wieder das Bit 5 zuständig. Zwar könnte man jetzt noch alle sonstigen Möglichkeiten und Eigenschaften des MFP erklären, aber das würde zu weit führen. Nur noch eins: Die Vektoren, über die beim Auftreten eines Interrupts des MFP gesprungen wird, stehen ab Adresse $100. Man erhält die Adresse des Vektors eines Interrupts durch Addition des Vierfachen der Interrupt-Nummer zu diesem Offset. In Tabelle 3 sind alle für die Programmierung des Timers A notwendigen Register und deren Adressen angegeben.

Zur besseren Demonstration programmiert das Beispielprogramm den Timer A so, daß durch häufig auftretende Interrupt-Anforderungen soviel Rechenzeit von der CPU 'abgegraben' wird, daß für die Ausführung des eigentlichen Benutzerprogramms schlimmstenfalls nichts mehr übrigbleibt: der ST steht still! Da die Bedienung aus dem Programmkopf hervorgeht, widmen wir uns nun den heiklen Seiten der Interrupt-Programmierung.

Rien ne va plus ...

Nicht besonders witzig ist folgende Situation: Keine Bomben, die Maus reagiert wie gewohnt, das Desktop erstrahlt im vollen Glanz ... Aber beim Auswählen der Drop-Down-Menüs in der Menü-Zeile tut sich nichts mehr, und die Tastatur hat in GEM-Programmen sowieso nicht viel zu melden. Diagnose: Das AES ist abgeschossen worden, jener Teil des GEM, der sich so fürsorglich um Menüs, Dialoge und Fenster etc. kümmert. Da hilft nur ein Reset (und manchmal nicht einmal der ...). Trat dieser Fauxpas in Zusammenhang mit der Timer-Programmierung auf, war unter Garantie die Blockierung anderer Interrupts der Sündenbock. Man muß sich eins vor Augen halten. Der MFP erscheint mit seinem Interrupt auf der für die CPU höchsten Prioritätsstufe. Damit wird durch Eintreten eines Interrupts an diesem Baustein automatisch der VBL-Interrupt blockiert, weil dieser auf Level 4 arbeitet und der Prozessor die Interrupt-Maske auf den aktuellen Interrupt-Level setzt; will heißen: Wenn der MFP einen Interrupt auf den Bus legt, setzte der 68000er die Interrupt-Maske sofort auf 6! Da im ST dies die höchste Stufe ist, werden bis zum Zurücksetzen der Maske auf die vorher eingestellte Stufe nur Interrupts vom MFP bearbeitet. Zum zweiten ist der Timer A auch innerhalb des MFP auf sehr hohe Priorität gesetzt: Vor ihm rangieren nur noch zwei weitere Interrupt-Quellen: RI(Ring Indicator), der nur bei angeschlossenem Modem von Bedeutung ist, und Monochrome Detect. Wer nicht permanent an seinem Monitorstecker herumspielt, hat hier auch nichts zu befürchten. Ergo: Wenn Timer A in Aktion tritt (einen Interrupt auslöst), sind alle anderen Interrupts ausmaskiert! Die oberste Pflicht einer eigenen Timer-Routine muß also sein, gleich am Anfang alle Interrupts wieder freizugeben; schließlich wollte man ja nur die Programmkontrolle haben. Es muß des weiteren darauf geachtet werden, daß die VBL-Interrupt-Routine unter keinen Umständen abgebrochen wird, weil hier vorwiegend sehr kritische Dinge passieren, die keinen Aufschub dulden. Ob man diese Routine unterbrochen hat, kann man an Bit 10 des Statusregisters ablesen, das ja der Prozessor auf den Stack abgelegt hat. Ist dieses Bit (IPL2) gesetzt, d.h. steht mindestens auf 4, empfiehlt sich ein beschleunigtes Verlassen der Interrupt-Routine! Ebenfalls zu beachten: Da man vielleicht nicht weiß, wie lange man in der Interrupt-Routine bleiben wird, empfiehlt sich das sofortige Ausschalten des Timers, weil sonst ein erneuter Interrupt auftreten könnte [Interrupts mit höherer oder gleicher(!) Priorität sind ja weiterhin zugelassen] und man mit dem alten noch gar nicht fertig ist ... Am Ende der Routine muß man den Timer dann wieder neu programmieren. Der nächste wichtige Punkt: die Floppy bzw. Harddisk. TOS setzt an der Adresse $43E beim Betreten einer Floppy-Routine ein Semaphore und will damit sagen, es möchte jetzt nicht gestört werden. Also müssen wir auch in diesem Fall unsere Zelte vorzeitig abbrechen. Nur wenn diese Speicherzelle 0 enthält, dürfen wir loslegen, sonst liefert die Floppy Daten, die das Betriebssystem noch gar nicht verarbeiten kann (Wirkung: Daten auf Disk A: defekt? Bitte überprüfen Sie ...)! Schließlich noch ein wichtiger Punkt: Man darf in der Timer-Routine fast alles anstellen, aber eines ist unter Absturzgefahr verboten: das Aufrufen des Betriebssystems über den TRAP-Befehl! Der Grund hierfür liegt darin, daß man u.U. eine Betriebssystemfunktion unterbrochen hat, und bei einem zweiten Aufruf aus der Interrupt-Routine würde man die geretteten Register etc. des unterbrochenen Programmes überschreiben (die Trap-Handler sind nicht reentrant)!!! Es gibt nur zwei Möglichkeiten für einen Betriebssystemaufruf aus einer Interrupt-Routine: Entweder man ruft die gewünschte Funktion direkt auf (Parameter auf den Stack legen, dann JSR ...), oder man schreibt sie sich selbst. Der letztere Fall ist der 'saubere' und meist auch der zeitgünstigere, da man nur die Teile einer Routine neuschreibt, die man auch benötigt.

Name:	Adresse:	Funktion:
TACR	$FFFA19	Kontrollregister, bestimmt Timerfunktion
TADR	$FFFA1F	Datenregister, Dekrement-Offset
IREA	$FFFA07	IR-Freigabe (Bit 5)
IRISA	$FFFA0F	Zeigt aktiven IR an (Bit 5)
IRPA	$FFFA0B	Bit 5 ist bei Timer A-IR gesetzt
IRMA	$FFFA13	IR-Maske (Bit 5)

Tab. 3: Überblick über die für den Timer A wichtigen Register. TREA und IRMA werden von XBTIMER() automatisch gesetzt. IRPA (Bit 5) zeigt einen "hängenden" Timer-Interrupt an; ein Rücksetzten ist nicht unbedingt notwendig.

Ihr Einsatz, bitte!

So schön ein sich in Super-Zeitlupe aufbauendes Fenster auch sein mag (probieren Sie es mit dem Demoprogramm doch einmal aus ...): sicherlich gibt es wesentlich sinnvollere Anwendungen für einen Timer. Zwei davon möchte ich Ihnen kurz als Anregung vorstellen. Da wäre zunächst die Implementation eines Spoolers (Simultaneous Peripheral Operartion On Line, was zu deutsch soviel bedeutet wie: Parallele Ausführung von Ein- und Ausgabe), die es erlaubt, alle Druckerausgaben im Hintergrund ablaufen zu lassen, also ohne Wartezeit für den Benutzer. Dazu benötigen wir Speicherplatz zur Zwischenablage des zu druckenden Dokuments. (Man kann natürlich das ganze auch dynamisch mit einer Zwischenablage auf Festplatte oder Diskette aufbauen.) Zwei anzulegende Pointer verweisen auf die aktuelle Schreib- und Leseposition innerhalb des Puffers. Statt die Daten direkt an den Drucker zu geben, werden sie in den Puffer (auf Platte) geschrieben und der Schreibpointer entsprechend erhöht. Ist der Puffer voll, wird wieder beim ersten Byte des Puffers weitergeschrieben, wobei man beachten muß, daß die aktuelle Leseposition nicht überschritten wird (Turn-around). Über den Timer A (oder eine andere Interrupt-Quelle) steuern wir nun eine Interrupt-Routine an, die überprüft, ob der Puffer noch auszugebende Daten enthält (Schreibposition > Leseposition, Überlauf beachten!). Ist dies der Fall, wird pro Interrupt 1 Byte mit einer eigenen Routine an den Drukker geschickt (oder über das TOS, siehe aber oben!) und der Lesezeiger inkrementiert. Fertig!

So einfach wie der Spooler ist die zweite Anregung nicht mehr: Multitas-

king heißt die Devise. Ein wenig seltsam ist der 68000er ja schon: Da stellt er einerseits dem Systemprogrammierer einen mächtigen Semaphore-Befehl (TAS adr) zur Verfügung, der im Multitasking-Betrieb sicherstellen kann, daß immer nur ein Prozeß auf einen kritischen Bereich zugreifen darf, und ist anderseits nicht dazu in der Lage, einen doppelten Busfehler zu beheben: Geben Sie mal im Supervisor-Modus den Befehl move.b d0,-(a7). Wenn sich dann noch irgendetwas rührt, empfehle ich das Aufschrauben des Gehäuses zwecks Überprüfung, ob da auch wirklich ein 68000er seinen Dienst versieht ... Trotzdem läßt sich bei sauberer Programmierung das quasiparallele Ausführen von Prozessen erreichen. Es würde Bände füllen, auf alle Besonderheiten und Eventualitäten bei der Implementation einzugehen, deshalb hier nur ein paar allgemeine Hinweise: Jeder der parallelen Prozesse benötigt eigene Stacks (für User- und Supervisor-Modus). Der Dispatcher, also der Prozeß-Umschalter, muß bei einem Jobwechsel die Register des aktuellen Prozesses retten und die des nächsten laden. Alle Trap-Handler müssen mit einem Semaphore ausgestattet werden, der von einem Prozeß vor Eintritt ins Betriebssystem abgeprüft wird (TAS-Befehl!). War er gleich Null, wird er auf -1 gesetzt, und der Prozeß darf die Routine aufrufen, sonst muß er gewartet werden, bis der Job, der sich gerade im Betriebssystem befindet, den Semaphore wieder auf Null gesetzt hat (Dies muß genau vor RTE, also dem Rücksprung ins Programm, erfolgen). Was sonst noch getan werden muß, hängt ganz davon ab, welche Forderungen man an das System stellt; hier sind also Ihre Phantasie und Intuition gefragt! Wie man sieht, kann man mit Timern und Interrupts eine ganze Menge anstellen.

Grafikroutinen

Der Programmierer findet im VDI des ST/TT bereits eine Reihe von Grafikfunktionen, die ihm das Leben leichter machen. Ob es ums Linienziehen, Rechteckzeichnen, Rechteckfüllen, Text- oder sogar Sprite-Ausgabe geht, ein VDI-Aufruf und die Funktion wird vom Betriebssystem übernommen. Leider handelt es sich bei den Routinen jedoch 'nur' um die einfachsten Grafikroutinen. Eine Möglichkeit zum Zoomen sucht man vergebens. Um Ihnen das Leben beim Programmieren von Grafikanwendungen noch ein wenig leichter zu machen, finden Sie nachfolgend weitere Grundfunktionen für Ihre Grafikbibliothek. Viele Routinen stehen hier, wie auch bei den übrigen Rubriken dieses Sonderhefts, in drei Sprachen zur Verfügung (C, PASCAL und BASIC). Aus Geschwindigkeitsgründen haben wir einzelne Routinen in Assembler belassen. Diese können Sie jedoch durch Aufruf auch in Ihrer Programmiersprache nutzen.

Scrollen in alle Richtungen

Eine Scroll-Routine muß, damit der Bildaufbau nicht ruckartig wird, möglichst schnell arbeiten. Eine Programmierung in Assembler ist deshalb unumgänglich. Außerdem muß die Parameterübergabe so einfach wie möglich ausfallen, da auch dies die Routine verlangsamt. Die hier vorgestellte Scroll-Routine benötigt als Parameter nur einen Zeiger auf eine Struktur, der sie dann alle benötigten Daten entnehmen kann.

Ulrich Witte

Damit entfällt eine weitere Einschränkung: werden nämlich direkt in der Routine Werte geändert (auch eine Möglichkeit, um Übergabeparameter zu sparen ...), braucht man für jeden zu scrollenden Block eine eigene Routine, bzw. es müßte jedesmal die Initialisierungsroutine aufgerufen werden, wenn ein anderer Block gescrollt werden soll, was nicht besonders zur Geschwindigkeitssteigerung beiträgt.

Die Struktur besteht aus folgenden Komponenten: words enthält die Wörter (16 Bit) pro Zeile, zeilen die Zeilenzahl, beide um 1 erniedrigt. offset enthält die Breite in Bytes, wird zur Korrektur des an der abzuarbeitenden Zeile langlaufenden Zeigers benötigt, adr ist die Adresse auf dem Monitor, pixel die Breite in Pixeln (wird nur für Links-rechts-Scrollen benötigt).

Den Scroll-Routinen wird neben dem Zeiger auf diese Struktur ein Wert für die Pause übergeben, die nach jeder Zeile beim vertikalen bzw. nach jedem Pixel beim horizontalen Scrollen gemacht werden soll, die Zahl der Zeilen bzw. Pixel, die pro Aufruf der Routine gescrollt werden sollen, und ein Flag für die Richtung. Die Pause verlangsamt das Scrollen absolut, auch für die andren zu scrollenden Bereiche, während die Pixel-Angabe das Scrollen des einen relativ zu den anderen Bereichen ändert. So kann z.B. ein Block bei jedem Durchlauf 4 Pixel scrollen, während die anderen nur 1 Pixel pro Durchlauf scrollen dürfen.

Die Routine scrollinit errechnet die für das Scrollen nötigen Parameter. Zu beachten ist, daß das Flag, das die Richtung bestimmt, den gleichen Wert haben muß wie beim Aufruf der Scroll-Routinen, da sonst falsche Bereiche gescrollt werden. Es findet keine Überprüfung auf negative Werte statt, die entstehen, wenn man das zweite x/y-Koordinatenpaar kleiner angibt als das erste; also Vorsicht!

Im (Assembler)Source sind an einigen Stellen Pfeile (==>), die deutlich machen, wo die Routinen an die Bildschirmbreite angepaßt werden sollten, damit wenigsten monochrom auf größeren Bildschirmen gescrollt werden kann.

Die Routinen sind so geschrieben, daß sie ohne globale Zugriffe auskommen, es müssen nur die beiden Strukturen vereinbart werden, was je in einer Header-Datei geschehen kann. Extra compiliert braucht man das File dann nur noch hinzuzulinken. Natürlich sind die Routinen weniger geeignet, wenn es darum geht, den ganzen Bildschirm zu verschieben. Die wortweise Schleife ist dann einfach zu langsam. Für

Aufruf: scrollinit(k, scroll, lo)
Funktion: initialisiert eine Scroll-Struktur
Parameter: k: Zeiger auf eine Rechteckstruktur; enthält die beiden gegenüberliegenden Eckpunkte
scroll: Zeiger auf Scroll-Struktur
lo: 0: setzt Startzeiger auf Ende des Rechtecks (rechtes unteres Eck); für Scrollen nach rechts oder unten
1: setzt Startzeiger auf Anfang des Rechtecks (linkes oberes Eck); für Scrollen nach links oder oben

Aufruf: scroll_lr(scroll, warten, pixel, richtung)
Funktion: scrollt Bildschirmbereich nach rechts oder links
Parameter: scroll: Zeiger auf Scroll-Struktur
warten: Pausenlänge nach Scrollen
pixel: Anzahl der zu scrollenden Pixel
richtung: 0: rechts
1: links

Aufruf: scroll_ou(scroll, warten, pixel, richtung)
Funktion: scrollt Bildschirmbereich nach oben oder unten
Parameter: scroll: Zeiger auf Scroll-Struktur
warten: Pausenlänge nach Scrollen
pixel: Anzahl der zu scrollenden Pixel-Zeilen
richtung: 0: unten
1: oben

kleinere Bereiche (und dann auch gerne mehrere) lassen sich aber doch recht flüssige Bewegungen erzielen.

Besondere Effekte erzielt man beim Übereinanderlegen von scrollenden Blöcken. Rechts + Oben = Diagonal, das ist ja wohl klar; überkreuzen sich z.B. 2 Blöcke, die in die gleiche Richtung scrollen, wird der überlappende Bereich auseinandergezogen, scrollen sie entgegengesetzt, scheint der Schnittpunkt stillzustehen. Weisen die beiden auch noch unterschiedliche Geschwindigkeiten auf, läuft der Schnittpunkt plötzlich spiegelverkehrt! Ein nettes Feld zum Experimentieren ergibt sich auch, wenn 2 senkrechte und 2 waagrechte 'Stäbe' so eingestellt sind, daß die Pixel im Kreis (bzw. im Rechteck) laufen.

Eine 'ernsthafte' Anwendung ist z.B. eine Laufschrift, wie im Beispielprogramm. Vielleicht kommt ja mal jemand auf die Idee, wichtige Alert-Boxen mit scrollenden Stopschildern auszustatten? Die werden dann bestimmt nicht mehr übersehen!

Beim Demo sind einige Tasten des Zehnerblocks belegt: '+' und '-' erhöhen bzw. erniedrigen die relative Geschwindigkeit der horizontalen Schrift, '*' und '/' die der vertikal scrollenden Schrift, '(' und ')' erniedrigen/erhöhen die absolute Geschwindigkeit (Pause). Enter setzt die Parameter zurück, und mit ESC können Sie das Programm verlassen.

Skalierung

Stellen Sie sich vor, Sie haben ein Programm geschrieben, das Formeln auswertet und das Ergebnis grafisch darstellt. Leider haben Sie jedoch außer einem einfachen Achsenkreuz keine weiteren Bezugslinien. Wenn Sie die Grafik betrachten, wüßten Sie gerne, welchen Wert (in etwa) die Kurve an einer bestimmten Stelle hat. Wie wäre es denn dann mit einer Beschriftung Ihres Achsenkreuzes (oder Ihrer Skalen)?

Jost Jahn

Sie haben zwei Möglichkeiten, das zu verwirklichen. Zum einen können Sie eine Routine dafür schreiben, oder aber Sie verwenden unser 'Achsenkreuz'. Mit dieser Routine können Sie beliebige Skalen und Achsen zeichnen, beschriften und, wenn Sie wollen, auch mit einem Raster unterlegen lassen. Ihr Programm muß dann 'nur noch' die Kurve(n) eintragen. Wäre das nicht was ...?

Unsere Routine Achsenkreuz benötigt zum Arbeiten den Bereich, innerhalb dessen das Achsenkreuz erscheinen soll, die maxi- und minimalen Wert für die x- und y-Achse, die Angabe, ob mit oder ohne Kasten, und einen Wert für das Raster. Diese Routine ruft ihrerseits ein Unterprogramm auf, das die einzelnen Achsen zeichnet. Sollten Sie in einer Anwendung nur Achsen benötigen, können Sie diese Routine verwenden. Sie benötigt folgende Parameter: Start- und Endpunkt der zu zeichnenden Achse sowie Start- und Endpunkt der anderen Achse (wird für Raster benötigt), Anfangs- und Endwert der Skalierung, die Ausrichtung und einen Wert fürs Raster.

Aufruf:	Achsenkreuz(Xl,Xr,Yu,Yo,Xmin,Xmax, Ymin,Ymax,Kasten,Raster)
Funktion:	zeichnet Achsenkreuz mit Skalierung; nach Wahl Unterteilung innerhalb des Achsenkreuzes
Parameter:	Xl, Xr: Pixel-Wert der linken und rechten Kante Yu, Yo: Pixel-Wert der unteren und oberen Kante Xmin, Xmax: mini- und maximaler Skalenwert für x-Richtung Ymin, Ymax: mini- und maximaler Skalenwert für y-Richtung Kasten: 0 nur Achsenkreuz, 1 auch Kästen Raster: 0 kein Raster, sonst Rasterbreite in 4er-Pixeln
Aufruf:	Achse(beg, end, beg2, end2, min, max, x_y, Raster)
Funktion:	zeichnet Achse mit Skalierung, Rasterunterteilung innerhalb der Achsen möglich
Parameter:	beg, end: Koordinaten des Anfangs- und Endpunktes der Achse (nur x- oder y-Wert) beg2, end2: Anfangs- und Endpunkt der anderen Achse min, max: mini- und maximaler Wert der Skalierung x_y: Ausrichtung der Achse (>0 y-Achse, <0 x-Achse; Beschriftung: 1 rechts, 2 links, -1 über und -2 unter der Achse) Raster: 0 kein Raster, sonst Rasterbreite in 4er-Pixeln

Schnelle und variable Textausgabe

Um eine schnelle Textausgabe zu erreichen, wurden die dazu notwendigen Routinen in Assembler geschrieben. Sie können aus jeder beliebigen Hochsprache heraus aufgerufen werden.

M. Malich/E. Grah

Ein Beispiel dafür befindet sich auf der Diskette. Den von diesem Programm ausgegebenen Text können Sie durch Bewegungen mit der Maus verschieben, durch Drücken der Maustasten verkleinern und vergrößern und mit Hilfe der '+'- und '-'-Taste drehen.

Das Maschinenprogramm besteht aus 4 Teilen:

1. der Initialisierung, in der eine Tabelle berechnet wird, die einen schnellen Zugriff auf die Daten jedes einzelnen Zeichens ermöglicht
2. einer Routine zur Einstellung der Ausgabeparameter (Zeichenfarbe, Maske für den Linienstil und Grafikmodus)
3. einer Routine zur Einstellung der Textparameter (Sinus und Cosinus des Drehwinkels, Kursiv-Parameter, Größe in x- und y-Richtung)
4. der eigentlichen Textausgabe.

Die Adressen zu den einzelnen Routinen liegen am Anfang des Assembler-Programms [INIT_POINTER bei Byte 0, SET_PARAMETER bei Byte 4, SET_COLOR Byte 8, und TEXT_OUT ab Byte 12 ($0B)].

Zur einfachen Handhabung der Zeichen sind sie als Polygonzüge in einer 20*40-Matrix definiert. Die Tabelle ist dabei folgendermaßen aufgebaut: Zu Anfang steht der ASCII-Code des jeweiligen Zeichens, anschließend folgt ein Polygonzug. Beginnend mit der x/y-Koordinate der Startposition folgt eine Liste von (x/y-) Vektoren. Zwei Null-Bytes kennzeichnen das Ende eines Polygonzuges. Das Ende der Polygonzüge eines ASCII-Zeichens wird durch ein auf zwei Null-Bytes folgendes $FF gekennzeichnet.

Bei Manipulationen an den einzelnen Zeichen braucht man sich nun nicht um jeden einzelnen Punkt des Zeichens zu kümmern. Die Berechnungen beziehen sich nur auf die Eckpunkte eines Polygonzugs:

Die Darstellung der Zeichen in beliebiger Größe wird durch die Formel: x_neu = x * x_Size erreicht. Um die Schrift bei Kursivdarstellung entsprechend zu neigen, wird die x-Ordinate des Zeichens um einen bestimmten Betrag verschoben, der von der zugehörigen y-Ordinate abhängt: x_neu = x + Kursiv * y.

Bei der Rotation des Zeichens um seinen Ursprung (linkes unteres Eck) wird die Berechnung komplexer, da es sich um eine echte Koordinatentransformation mit Hilfe einer Rotationsmatrix handelt. Multipliziert man die Rotationsmatrix mit den jeweiligen Vektoren, erhält man die folgenden Gleichungen: x_neu = x * cos(+) - y * sin(+); y_neu = x * sin(+) + y * cos(+). Zu diesen Werten werden noch die Bildschirmkoordinaten addiert. Nach Berechnung zweier Koordinaten wird eine Linie des Zeichens auf dem Bildschirm dargestellt.

Die Assembler-Routinen liefern keine Rückgabewerte. Die benötigten Parameter entnehmen Sie der nachfolgenden Aufstellung. Übergabe siehe Listing auf den Disketten.

Routine: INIT_POINTER
Funktion: Initialisierung
Parameter: keine

Routine: SET_COLOR
Funktion: Farbe, Linienart und Grafikmodus setzten
Parameter: Color: Zeichenfarbe
Mask: Linienmaske (16 Bit)
Mode: Grafikmodus (0, 1, 2 oder 3)

Routine: SET_PARAMETER
Funktion: Paramater für Schriftdarstellung tlegen
Parameter: Alpha: Drehwinkel in Bogenmaß [0-2*π]
Kursiv: Kursivkonstante (0-300)
xsize, ysize: Zeichengröße (0-600)

Routine: TEXT_OUT
Funktion: Textausgabe
Parameter: Text: Zeiger auf auszugebenden Text
xpos, ypos: Bildschirmposition in Pixel

Cursor an GEM-Textattribute anpassen

Mit Hilfe dieser Routine können Texte an beliebiger Stelle mit Ausnutzung aller GEM-Textattribute eingelesen werden. Texteingaben im rechten Winkel oder auf dem Kopf sind kein Problem.

Tom Quellenberg

Die Texteingabe erfolgt mit den gerade aktuellen Attributen, die von der Routine, soweit nötig, selbständig erkannt werden. Eine Korrektur mit Backspace ist möglich. Der Clou aber ist, daß die Größe des Cursors immer der Größe der gerade gewählten Schrift automatisch angepaßt wird. Ist die Schrift im rechten Winkel gedreht, wird auch der Cursor um diesen Winkel versetzt.

Zu Beginn können verschiedene Textattribute eingestellt werden. Dies sind die Schriftgröße (0 bis 27), der Ausgabewinkel (0, 90, 180 oder 270) der Textstil (0 bis 31). Außerdem kann festgelegt werden, ob die anschließende Ausgabe im Replace- oder Transparent-Modus erfolgen soll.

Nun wird der Cursor auf dem Bildschirm dargestellt und kann mit Hilfe der Maus an eine beliebige Stelle bewegt werden.

Mit der rechten Maustaste kann die Funktion abgebrochen werden, mit der linken Maustaste oder einer beliebigen Taste auf der Tastatur erfolgt der Start der Eingabe.

Nach Beendigung der Eingabe mit der RETURN-Taste wird der Text mit dem am Anfang festgelegten Schreibmodus ausgegeben. Erfolgt die Ausgabe im Transparentmodus, wird der Hintergrund, auf dem der Text geschrieben wird, nicht gelöscht (auf schwarzem Hintergrund wären die Buchstaben nicht zu sehen). Im Replace-Modus wird dagegen der Hintergrund gelöscht (allerdings löschen sich Buchstaben, wenn sie kursiv und groß geschrieben werden, teilweise gegenseitig aus).

Der Cursor kann nun wieder frei auf dem Bildschirm bewegt werden, und mit einem Druck auf die linke Maustaste wird die Ausgabe mit den gleichen Textattributen fortgesetzt.

Durch einen Druck auf die rechte Maustaste läßt sich das Programm beenden. Die Einstellungen der Textattribute können damit auch verändert und dann die Textausgabe mit den neuen Einstellungen fortgesetzt werden.

Aufruf:	Gem_text
Funktion:	Festlegung der Textattribute und Texteingabe/-ausgabe
Parameter:	keine

Geglättete Polygonzüge

Polygonzüge haben einen Nachteil - es sind Polygonzüge! Die hier vorgestellte Routine nimmt dem Polygonzug seine Ecken, sie interpoliert zwischen den einzelnen Punkten.

Dietmar Rabich

Eine Kurve ohne die typischen Ecken von Polygonzügen ist nicht ohne eine zusätzliche Berechnung möglich. Interpolation bietet sich geradezu an, um eine Kurve sichtbar zu glätten. Vielleicht werden derartige Routinen auch mal in einigen Grafikprogrammen aufgenommen!

Aber erst eine kurze Einführung in die mathematischen Notwendigkeiten. Interpolation wird benötigt, wenn man statt einer Funktionsvorschrift nur einzelne Werte kennt, beispielsweise bei einer Reihe von Meßwerten. Mit Hilfe mathematischer Formeln werden Zwischenwerte berechnet, so wie sie tatsächlich sein könnten. Zumindest sollte eine Interpolation so gute Werte liefern, daß sie von tatsächlichen nur minimal abweichen.

Bei unserem Programm werden ebenfalls Werte angenähert. Vorgegeben ist eine Reihe (mindestens vier) von Wertepaaren (x/y) für Bildschirmkoordinaten. Durch diese Punkte wird eine Kurve gelegt. Das Beispielprogramm liefert schon eine recht anschauliche Kurve. Die Marker zeigen, welche Punkte vorgegeben waren. Es sind wirklich nur zwölf!

Gerade war schon von Funktionswerten die Rede. Diese sind hier die Bildschirmpositionen, also x/y-Koordinatenpaare. Neben diesen Werten sind noch Ableitungen nötig. Die erste Ableitung einer Funktion sagt etwas über ihr Wachstumsverhalten aus, die zweite etwas über ihr Krümmungsverhalten - und genau das interessiert uns! Wir brauchen nunmehr neben den Bildschirmkoordinaten auch das Krümmungsverhalten der Kurve in den jeweiligen Punkten.

Das ist nicht so schlimm, denn die Aufgabe der Berechnung übernimmt die Routine. Allerdings müssen zwei Werte vorgegeben werden; das Programm kann das Krümmungsverhalten am Anfangs- und Endpunkt der Kurve nicht berechnen. Aber zwei Werte sind ja leicht anzugeben, notfalls nimmt man immer 0, was bedeutet, daß die Kurve an ihren beiden Endpunkten nicht gekrümmt ist.

Bevor die Kurve ausgegeben werden kann, ist eine Näherung zu berechnen. Diese Aufgabe übernimmt die Routine Make_Splines. Die Ausgabe übernimmt dann Draw_Splines. Mehr ist bei der Bedienung nicht zu beachten.

Draw_Splines hat die Aufgabe, die Kurve auszugeben. Die bereits berechneten Werte werden in die Spline-Interpolierende (die Routinen benutzen Spline-Interpolation), ein kubisches Polynom, eingesetzt und ausgewertet. Die Auswertung ist lediglich die Ausgabe der Punkte, d.h Linien zwischen den einzelnen Punkten. Wieviel interpolierende Punkte berechnet werden sollen, ist auch der Routine Draw_Splines anzugeben. Da mit Linien gearbeitet wird, brauchen es nicht sehr viele zu sein. Je mehr Punkte berechnet werden sollen, desto länger dauert die Ausgabe auf dem Monitor.

Die Hauptaufgabe fällt der Make-Splines-Routine zu. Zuerst werden Parameter berechnet. Hat man beispielsweise eine Funktion f(x) gegeben, so existieren neben den Bildwerten f(x) auch Urbildwerte x (Eingangsparameter für die Funktion). Die Bildschirmkoordinaten werden nur als Bildwerte angesehen, was fehlt, sind Urbildwerte.

Make_Parameter liefert die Urbildwerte, einfach den Abstand der Punkte nacheinander. Danach wird in der Routine Make_System das Gleichungssystem aufgebaut, welches als Lösung das Krümmungsverhalten der Kurve an den einzelnen Punkten liefert, oder, um es etwas genauer auszudrücken, es wird die zweite Ableitung an den einzelnen Stellen berechnet. Glücklicherweise hat die Koeffizientenmatrix dieses linearen Gleichungssystems Tridiagonalgestalt, wodurch sich der Rechenaufwand verringert.

Bevor man die Koeffizienten des Interpolationspolynoms in Set_Splines errechnen kann, muß noch mit Solve_System die Lösung des Gleichungssystems ermittelt werden.

Zur Anwendung der beiden Routinen Make_Splines und Draw_Splines muß ein Array mit mindestens 4 Bildschirmpunkten (x/y) definiert werden. Außerdem müssen zwei Arrays für die errechneten x- und y-Splines zur Verfügung stehen. Als Eingangsparameter benötigt Make_Splines die Adresse des x/y-Arrays, die Anzahl der (vorhandenen) Bildschirmkoordinaten, die Adressen der beiden Splines-Arrays und die beiden Krümmungswerte. Zum Zeichnen der Kurve rufen Sie Draw_Splines mit den beiden Splines-Arrays und der Anzahl der Werte auf.

Wer mehr zur Theorie der Spline-Interpolation oder zu den Grafikfunktionen des VDI wissen möchte, findet in den beiden unten aufgeführten Büchern sicherlich wertvolle Hinweise.

Literatur:

[1] Numerische Mathematik, H.R. Schwarz, Teubner
[2] Atari St Profibuch, H.-D. Jankowski/J. F. Reschke/D. Rabich, Sybex

Aufruf:	Make_Splines(points, nr, spx, spy, kra, kre)
Funktion:	berechnet Splines zu den x/y-Koordinatenpaaren
Parameter:	points: Array mit x/y-Werten nr: Anzahl der Werte im point-Array spx, spy: Arrays für Splines-Werte kra, kre: Krümmung (2. Ableitung) der Kurve im Anfangs- und Endpunkt
Aufruf:	Draw_Splines(spx, spy, n)
Funktion:	zeichnet die Kurve
Parameter:	spx, spy: Arrays mit Splines-Werten n: Anzahl der Werte

Schönere Kreise

Einen echten Kreis auf einem quadratischen Raster, wie es bei Bildschirm- und Druckerausgabe vorhanden ist, zu zeichnen, ist unmöglich. Es besteht nur die Möglichkeit, die einzelnen Punkte des Kreisumfangs im Raster so zu setzen, daß die Abweichung von der idealen Linie möglichst gering ist.

Andreas Hollmann

Zur Lösung dieses Problems werden verschiedenen Algorithmen verwendet. Der Algorithmus, der vom VDI zum Kreiszeichnen verwendet wird, besitzt zwei Merkmale, die sofort ins Auge stechen: die Kreise werden unheimlich schnell und unheimlich häßlich gezeichnet.

Leider habe ich keine Informationen über den benutzten Algorithmus, aber, betrachtet man den Kreisbogen bei bestimmten Radien, so drängt sich die Vermutung auf, daß in Wirklichkeit Polygone gezeichnet werden. Das würde die hohe Geschwindigkeit erklären, denn ein paar Polygonecken sind schneller berechnet als jeder einzelne Punkt auf dem Kreisumfang.

Die durchgezogenen Linien der VDI-Darstellung sind schon schlimm genug, aber bei Verwendung verschiedener Linienmuster packt einen das Grauen.

Schönere Kreise erhält man durch das Setzen einzelnen Pixel (und keiner Linienzüge). Das Zeitaufwendigste dabei ist die Berechnung jedes einzelnen Kreispunktes. Der in der Routine verwendete Algorithmus basiert auf Horns Algorithmus, der zur Berechnung nur Addition, Subtraktion und binäre Schiebeoperationen verwendet (keine Multiplikation).

Der Routine circle werden die x- und y-Koordinaten des Kreismittelpunktes, der Radius und das Linienmuster (in einem 16-Bit-Word) übergeben. Beim Setzen der einzelnen Punkte wurde darauf geachtet, daß das VDI-.Clipping-Rechteck berücksichtigt wird und auf Bildschirmen mit anderer Auflösung keine Probleme auftreten.

Der Kreis wird in acht Oktanten unterteilt, deren Kreisbögen nacheinander gezeichnet werden. Vor dem Setzen eines Punktes wird geprüft, ob das Bit 0 im Linienmuster-Wort gesetzt ist. Danach wird das Linienmuster um ein Bit rotiert, man spart dadurch (zeit)aufwendige Bit-Zählerei.

Das Ergebnis läßt sich in der abgebildeten Hardcopy begutachten. Bei den Kreisen mit durchgezogenen Linien fällt auf, daß selbst Kreise von 2 Pixeln (kleinster Kreis) noch kreisähnliche Objekte ergeben, wogegen man beim VDI-'Kreis' einen Stern sieht.

Beim gepunkteten VDI-'Kreis' kleben teilweise einige Pixel in Gruppen zusammen, als Ausgleich dafür läßt VDI aber an anderen Stellen etwas größere Lücken ... Horns Algorithmus zeigt solche Effekte nicht und setzt die Punkte in gleichmäßigen Abständen

Literatur:
Computer Graphics and Image Processing
1979 Marek Doros
Incremental Circle Generator

Aufruf:	circle(x, y, r, type)
Funktion:	zeichnet Kreise nach Horns Algorithmus
Parameter:	x, y: Mittelpunktskoordinaten des Kreises
	r: Kreisradius
	type: Linientyp (16-Bit-Wort)

Überblendeffekte

Ein gutes Programm sollte auch grafisch ansprechend gestaltet sein. Wenn das Titelbild, von der Diskette geladen, ruckweise auf den Bildschirm stottert, macht das sicherlich keinen positiven Eindruck auf den Anwender; eine Möglichkeit zum eleganten Einblenden von Bildern muß her!

Andreas Hollmann

Die hier vorgestellten Routinen bieten Ihnen die Möglichkeit, Grafiken, unter Verwendung von Zufallszahlen, 'weich' überzublenden. Wie sich bei der Entwicklung der Routinen herausgestellt hat, dürfen die Zufallszahlen jedoch nicht völlig zufällig sein. Stellen Sie sich folgenden Fall vor: Sie wollen zwischen Bildern mit je 32 KB byteweise überblenden. Sie nehmen dazu Zufallszahlen zwischen 1 und 32000 und blenden in einer Schleife mit 32000 Durchläufen um. Wo das Problem liegt, fragen Sie. Nun, wer sagt Ihnen denn, daß alle 32000 Zahlen genau einmal vorkommen? Zufallszahlen haben leider die Angewohnheit, daß man eben nicht weiß, wann (und wie häufig) sie auftreten. Für die folgenden Routinen benötigen wir also 'Zufallszahlen', bei denen sichergestellt ist, daß aus dem gewählten Bereich, jede Zahl exakt einmal vorkommt. Mit Hilfe einer Liste, in die alle bereits vorgekommenen Zahlen eingetragen werden, könnte man das Problem zwar lösen, jedoch nicht in einer annehmbaren Zeitspanne. (Bei bitweiser Überblendung benötigen Sie Zahlen von 1 bis 256000!)

Die Assembler-Routinen liefern keine Rückgabewerte. Die benötigten Eingangsparameter können der folgenden Aufstellung entnommen werden:

Routine: Add1fade(Bild, Schirm, Add)
Funktion: pixelweises Überblenden durch Additionsalgorithmus
Parameter: Bild: Speicherstartadresse der Bilddaten
Schirm: Adresse des physikalischen Bildschirms
Add: Additionsparameter (zwischen 1 und 32000)

Routine: Add8fade(Bild, Schirm, Add)
Funktion: byteweises Überblenden durch Additionsalgorithmus
Parameter: Bild: Speicherstartadresse der Bilddaten
Schirm: Adresse des physikalischen Bildschirms
Add: Additionsparameter (zwischen 1 und 256000)

Routine: Rnd1fade(Bild, Schirm)
Funktion: pixelweises Überblenden durch Pseudozufallszahlen
Parameter: Bild: Speicherstartadresse der Bilddaten
Schirm: Adresse des physikalischen Bildschirms

Routine: Rnd8fade(Bild, Schirm)
Funktion: byteweises Überblenden durch Pseudozufallszahlen
Parameter: Bild: Speicherstartadresse der Bilddaten
Schirm: Adresse des physikalischen Bildschirms

Regelmäßiger Zufall

Die erste Lösungsmethode hat eigentlich recht wenig (um genau zu sein gar nichts!) mit Zufall zu tun: Bei jeder Überblendung wird als Startwert die Bildadresse 0 genom[Sie können die Routinen auch dahingehend erweitern, daß Sie eine beliebigen (zufälligen) Startwert nehmen]. Hierzu wird nun ein Wert zwischen 1 und 31999 addiert (Einschränkungen siehe unten). Bei einer Zieladresse größer als 32000 wird 32000 subtrahiert. Wenn wir nun eine Schleife mit 32000 Durchgängen programmieren, so wird jeder Bildpunkt (hier byteweise) genau einmal bearbeitet. Bei der Wahl des Additionswertes können Sie weitgehend frei entscheiden; allerdings darf die Konstante keine gerade Zahl sein und nicht mit 5 enden. Trotz dieser Beschränkungen ergeben sich 12800 mögliche Effekte (siehe Routine Add8fade). Bei einer pixel(bit)weisen Überblendung stehen Ihnen sogar 102400 Möglichkeiten zur Auswahl (siehe Add1fade).

Pseudozufall

Möchte man den Effekt doch lieber etwas zufälliger haben, kann man auf Pseudozufallszahlen zurückgreifen. Der Unterschied zwischen echten und Pseudozufallszahlen besteht darin, daß sich die Zahlenfolge beim 'unechten' Zufall nach einer bestimmten Periode wiederholt. Realisiert werden diese Zahlen mit rückgekoppelten Schieberegistern. Bei byte-

MAXON PASCAL

Integriertes System

MAXON Pascal bietet alles in einem. Compiler, Editor, Linker und Assembler stehen resident zur Verfügung.

- MAXON Pascal arbeitet vollständig im RAM. Kein Zugriff auf Platte/Diskette notwendig. Dadurch erreicht man traumhaft schnelle Turnaround-Zeiten.
- Interaktive Fehlererkennung bei Syntax- und Runtime-Fehlern. Der Compiler springt sofort zur fehlerhaften Stelle im Editor.
- zusätzlich ist ein Compiler als CommandLine-Version zum Einbinden in eigene Entwicklungsumgebung enthalten.

Geschwindigkeit

- Turboschneller Single-Pass-Compiler (20.000 Zeilen auf ST)
- Schneller und kompakter Programm-Code
- UNITS erlauben die modulare Zerlegung bestimmter Programmteile und schnellste Übersetzung auch bei großen Projekten.
- Code-Optimierung - der integrierte Linker bindet nur die benötigten Teile einer UNIT an das Programm.

DIE WELT HAT EINEN NEUEN PASCAL-COMPILER

Systemunterstützung

MAXON Pascal erlaubt den Zugriff auf sämtliche Funktionen des ST-Betriebsystems (VDI, AES, BIOS, XBIOS, GEMDOS), in standardisierter, C-kompatibler Form.

Kompatibilität

MAXON Pascal ist ein eigenständiges, aber auch weltoffenes Pascal-System für Atari.

- weitgehende Kompatibilität zu TurboPascal 5.0. Programme können ohne große Änderungen übernommen werden.
- GRAPH-UNIT unterstützt Standard PC-Grafik
- eine spezielle ST Pascal-UNIT stellt abweichende Befehle und Definitionen zur Verfügung. ST Pascal-Programme lassen sich dadurch leicht portieren.

INLINE-Assembler

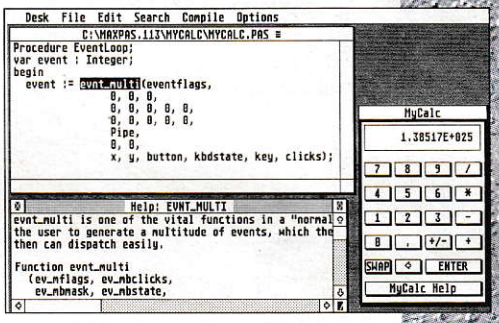
MAXON Pascal versteht auch direkten Assembler-Code. Somit lassen sich systemnahe oder extrem zeitkritische Programmteile in Assembler verfassen und samt Variablenübergabe direkt in den Pascal-Source einfügen.

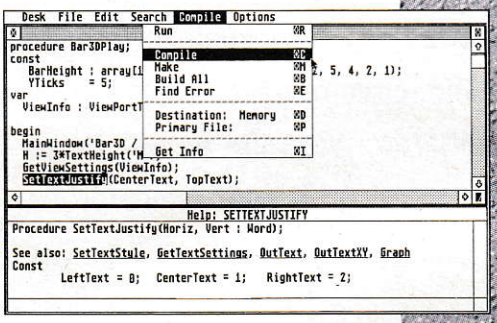
Hochpräzise Arithmetik

MAXON Pascal verfügt über schnelle mathematische Funktionen mit höchster Genauigkeit (18 Stellen, $\pm 1.1e^{\pm 4932}$ Stellen), sowie über die Unterstützung des 68881-Floating Point Prozessors.

OnLine-Help

- Auf Tastendruck liefert die integrierte Hilfefunktion Erklärungen zu dem angewählten Befehl.
- Zahlreiche Beispiele erläutern z.B. die Programmierung von GEM-Programmen in Pascal.
- Für CLI-Betrieb steht externes Help-Accessory zur Verfügung.





Neu in V. 1.5

- typisierte Konstanten: erlaubt die Typ-Zuweisung bei Konstanten bei gleichzeitiger Definition des Inhalts.
- ARRAYS > 32kByte: Array können nun beliebig groß werden.
- ABSOLUTE: Definition von Variablen an absoluter Speicheradresse.
- Optimierung der internen Speicherverwaltung (Word, Byte).
- Überarbeiteter Editor

Update DM 30.- gegen Einsendung der Originaldiskette (nur Vorauskasse möglich)

MAXON PASCAL 1.5

DM 259.-

unverbindliche Preisempfehlung

Turbopower für Atari ST/TT

MAXON Computer GmbH
Schwalbacher Str. 52 • 6236 Eschborn
Tel.: 06196/481811 • Fax: 06196/41885

MAXON computer gmbh

weiser Überblendung benötigt man ein 15 Bit breites Register. Dieses deckt den benötigten Wertebereich von 32000 ab (2^{15}=32768). Das Register erhält am Anfang den Startwert '1'. Nun werden die beiden untersten Bits (1 und 0) miteinander exklusiv-oder-verknüpft. Das Ergebnis (eine 0 oder 1) wird, nach Rechtsverschiebung des Registers, in die höchste Stelle (Bit 14) geschrieben (siehe Bild). Dies ist die neue Pseudozufallszahl. Die maximal erreichbare Periodenlänge ist 2n-1, hier also 32767, d.h. nach 32767 Durchläufen ist jede Zahl genau 1mal vorgekommen. Alle Zahlen ab 32000 werden ignoriert. Zum Überblenden werden also alle Zahlen zwischen 0 und 31999 verwendet (siehe Rnd8fade).

Auch bei dieser Methode ist eine pixelweise Überblendung möglich. Allerdings wird dann ein 18 Bit breites Register benötigt. Auch die Abgriffe finden an anderen Stellen (Bit 0 und 7) statt. Selbst bei Assembler-Programmierung kommt man bei dieser Routine (Rnd1fade) an die Grenze der Rechnerleistung eines 8MHz-68000ers (für eine Überblendung werden 5,7s benötigt).

Verwendung der Routinen

Aus Geschwindigkeitsgründen sind alle vier Routinen in Assembler programmiert. Sie benötigen als Eingabeparameter jeweils die Adresse des aktuellen (physikalischen) Bildschirmspeichers (XBIOS-Funktion 2) und die der Stelle, an der das Bild im Speicher beginnt. Die Additionsroutinen (Add1fade und Add8fade) bekommen als dritten Parameter den Additionswert.

Das Beispielsprogramm demonstriert Ihnen anschaulich die Überblendeffekte. Es benötigt dazu eine ASCII-Datei namens 'PIC_SHOW.INF' welche folgenden Aufbau haben muß: In der ersten Zeile steht die Sekundenanzahl, während der ein Bild jeweils zu sehen sein soll (Sie können die Routine dahingehend abändern, daß sie jedem Bild eine eigene Verweilzeit zuweisen). Zeile zwei enthält die Anzahl der Wiederholungen für die komplette Bildsequenz. In den darauffolgenden Zeilen werden die Pfade und Namen der zu zeigenden Bilder und der gewünschte Überblendeffekt (im Beispiel sind 10 ausgewählt) angegeben. Die letzte Zeile muß den Befehl 'end' enthalten. Ein Beispiel:

```
10
3
a:\bilder\bild1.pic
effekt 3
a:\bilder\bild2.pic
effekt 7
end
```

Fastzoom

Eine Zoom-Funktion sollte in keinem Grafikprogramm fehlen. Aber auch andere Anwendungen können von einer schnellen Zoom-Routine profitieren. Wie wäre es z.B. bei einem Titelbild mit Schrift- oder Bildvergrößerung (geringerer Speicherplatzbedarf auf Diskette)?

Gerald Schmieder

Die Routine benötigt als Eingangsparameter den Vergrößerungsfaktor, den (Bildschirm-)Ort des Originals, dessen Größe, den Zielort der Vergrößerung [jeweils für die horizontale (x) und Vertikale (y)] und den Schreibmodus.

Die Routine vergrößert das Original zeilenorientiert von hinten nach vorne. Die Richtung wurde gewählt, damit ein Objekt links oben am Bildschirm auch dorthin vergrößert werden kann. Beim umgekehrten Durchlaufen der Schleifen würde dies nicht funktionieren. Beim Vergrößern würde so ein mit der ersten Zeile gefüllter Bildausschnitt entstehen; bei Text sogar ein leerer Bildschirm: Die erste Zeile wird abgetastet. Diese ist bei Text leer. Bei einem Vergrößerungsfaktor von z.B. 5 würden nun 5 Leerzeilen am oberen Bildschirmrand erzeugt. Die nächste Zeile (normalerweise der oberste Rand der Buchstaben) wäre nun eine Leerzeile und würde 5 weitere Leerzeilen erzeugen. Das Ergebnis wäre eine leerer Bildschirm(ausschnitt).

Aufruf:	Zoom(Vx, Vy, Xpos, Ypos, X, Y, Xzoom, Yzoom, Modus)
Funktion:	variable Vergrößerung eines Bildschirmausschnitts
Parameter:	Vx, Vy: x- und y-Vergrößerungsfaktor Xpos, Ypos: Position des Originals (linkes oberes Eck) Xw, Yw: Größe des zu zoomenden Objekts (in Pixeln) Xzoom, Yzoom: Position des gezoomten Bildes (linkes oberes Eck) Modus: Schreibmodus

Kopieren und Verschieben variabler Bildausschnitte

Jedes anspruchsvolle Mal- und Grafikprogramm verfügt über eine Funktion, die es gestattet, Bildausschnitte von beliebiger Größe auf dem Bildschirm zu verschieben bzw. zu kopieren. Hierbei besteht der Unterschied zwischen 'Verschieben' und 'Kopieren' darin, daß bei der Funktion 'Verschieben' der Ausschnitt im wahrsten Sinne aus dem Bild 'herausgeschnitten' wird.

Hans-H. Ackermann

Bei der Funktion 'Kopieren' verbleibt der Originalausschnitt im Bild, die Kopie des Ausschnitts läßt sich an anderer Stelle beliebig positionieren. Das Verschieben von grafischen Einheiten stellt den wesentlichen Bestandteil des sogenannten Layouts etwa einer Druckvorlage dar. Die hier vorgestellten Routinen lassen Schere und Klebstoff - Hauptwerkzeuge beim Layout - endgültig überflüssig werden: Alle grafischen Einheiten, seien es Texte, Bilder oder auch Notensysteme, können mit der Maus auf dem Bildschirm druckfertig arrangiert werden.

Das Prinzip

Der zu verschiebende Bildausschnitt wird mit dem 'Gummifaden' eingerahmt. Dieser eingerahmte Bildschirmbereich wird in geeigneter Weise zwischengespeichert - gepuffert - und kann dann an beliebiger Stelle mit der Maus wieder auf dem Bildschirm plaziert werden. Bei der Funktion „Verschieben" wird der Bereich unter dem Rahmen gelöscht.

Die Realisierung

Die Realisierung dieses an sich einfachen Prinzips findet sich in der Routine Verschieben(Copy). Der einzige Eingangsparameter bestimmt, ob der Ausschnitt verschoben (Copy = 0) oder kopiert (Copy = 1) werden soll.

Die Hauptschleife ist nur notwendig, falls nach dem Aufruf der Routine die Möglichkeit bestehen soll, mehrere Verschiebungen unmittelbar nacheinander vornehmen zu können.

Ein Druck auf die linke Maustaste bestimmt nun den linken oberen Eckpunkt des zu verschiebenden/kopierenden Rechtecks. Danach wird ein Rahmen gezeichnet, dessen rechtes unteres Eck sich an der jeweils aktuellen Mauszeigerposition befindet. Nach Loslassen der linken Maustaste wird die innerhalb des Rahmens liegende Grafik zwischengespeichert. Soll der Ausschnitt nur verschoben werden, so wird nun der Bereich innerhalb des Auswahlrahmens gelöscht. Außerdem wird die Position des Mauszeigers an die linke obere Ecke gesetzt. Nun kann durch Bewegung der Maus der Ausschnitt auf dem Bildschirm an die neue Position verschoben werden. Er wird dort durch ein erneutes Anwählen der linken Maustaste 'festgeklebt'.

Anhand des Beispielprogramms kann die Funktionsweise der Routine demonstriert werden.

Aufruf:	Verschieben(Copy)
Funktion:	Mittels Maus kann ein rechteckiger Bildschirmbereich ausgewählt werden, der anschließend an eine neue Position verschoben (Copy = 0) oder kopiert (Copy = 1) werden kann.
Parameter:	Copy: Bestimmt Funktion der Routine. 0: Verschieben; 1: Kopieren

Lasso-Funktion

Ich stand vor dem Problem, aus einem Bild einen unregelmäßig geformten Ausschnitt herauszutrennen. Viele Malprogramme bieten zu diesem Zweck eine 'Lasso'-Funktion an. Da ich weder ein solches Programm noch eine Grafik-Bibliothek mit dieser Funktion mein eigen nenne, entschied ich mich, die entsprechende Routine selbst zu erstellen.

Ludwig Canisius

Meine Lasso-Routine schneidet einen mit der Maus definierten Grafikblock aus und legt ihn zwecks Weiterverarbeitung in einem Puffer im BITBLT-Format ab. Im Demoprogramm kann der gewonnene Ausschnitt an anderer Stelle im Bildschirm einkopiert werden.

Verwendung der Routine

Die Lasso-Prozedur wird vom Hauptprogramm aus mit Lasso(Undo,Modus) aktiviert. Der Eingabeparameter Undo entscheidet, ob ein zusätzlicher Speicher für eine UNDO-Funktion angelegt werden soll (Undo = 1) oder nicht (Undo = 0). Der Parameter Modus gibt die Art der Verknüpfung des Lasso-Inhaltes mit dem Hintergrund an.

Nach Aufruf wird zunächst mit einem Mausklick der Lasso-Startpunkt festgelegt. Der Zeiger zieht ab jetzt die Schlinge um die Grafik, bis zum zweiten Mal geklickt wird. Ergibt sich keine geschlossene Form, werden der Anfangs- und Endpunkt der Strecke miteinander verbunden. Der so ausgewählte Bildschirmausschnitt kann nun verschoben werden. Die linke Maustaste setzt den Block gemäß der mit Modus festgelegten Verknüpfungsart auf das Bild, die rechte beendet die Bearbeitung.

Prinzip der Lasso-Routine

Um einen unregelmäßig geformten Block ausschneiden zu können, muß man über eine Maske auf einem 2. Bildschirm verfügen, deren Form der des Blockes entspricht. Diese Maske wird einfach mit dem Originalbild UND-verknüpft, und der Ausschnitt steht bereit. Ist die Form ein Vieleck, reicht es, sie auszufüllen. Dieses Verfahren kann hier jedoch nicht angewendet werden, da der Benutzer mit dem Lasso jede beliebige Umrandung, also auch Überschneidungen, bestimmen können soll. Wird z.B. eine '8' als Randform gezeichnet, scheidet o.g. Methode aus, weil lediglich einer der Kreise der '8' mit einem Füllbefehl ausgefüllt werden kann. Außerdem würde bei dieser Methode die Umrandung selbst zum Ausschnitt gehören, was nicht wünschenswert ist. Aus diesem Grunde geht die Routine den umgekehrten Weg: man füllt den Teil des Bildes auf, der nicht vom Lasso umrahmt wird, und bekommt somit eine Maske zu jedweder Lasso-Form. Diese Maske muß, da sie invertiert ist, mit dem Bild NICHT UND-verknüpft werden. Ein Nachteil hierbei: um eine sichere Umrandung der Form durch den Füllbefehl zu gewährleisten, muß an den Bildrändern jeweils 1 Punkt freibleiben, damit die Füllroutine des gesamte Lasso umschließen kann.

Im Beispielprogramm wird die Routine einmal mit und einmal ohne UNDO-Funktion aufgerufen, wobei vorher ein Beispielbild mit gefüllten Kreisen gezeichnet wird.

Aufruf:	Lasso(Undo,Modus)
Funktion:	Mit der Maus kann ein unregelmäßiger Bildschirmbereich ausgeschnitten und an beliebiger Stelle eingefügt werden
Parameter:	Undo: Wenn Undo = 1, wird ein zusätzlicher Zwischenspeicher für eine UNDO-Funktion angelegt Modus: Verknüpfungsart des Lasso-Bereichs mit dem Bildschirmhintergrund

GFA-BASIC

hat einen starken Partner gefunden:

ergo!

*Die einzigartige Entwicklungsumgebung für **GFA-BASIC** ab Version 3.0*

Shell

bequemer Aufruf von Interpreter, Compiler etc. über Menüleiste, frei konfigurierbar, Hilfstexte zu Menüs und Dialogen, Compileroptionen über Dialogbox mit Erklärungen, Hotkeys.

Analyzer

Variablenanalyse schnell und übersichtlich, alle Informationen über Accessory im Interpreter(!) verfügbar, ausgefeilte Formatierroutine für kompakte Druckausgabe, dadurch keine ellenlangen Listen, Ausgabe individuell einstellbar, graphisches Baumdiagramm, Diagnosemodus findet Fehler und macht Verbesserungsvorschläge.

Präprozessor

Ausblenden/Einblenden von Programmteilen, dadurch Verwaltung verschiedener Versionen innerhalb eines einzigen Quelltextes möglich, symbolischen Konstanten ersetzen, Kill-Rem, Sourcecode verschlüsseln, Variablennamen ändern, Übertragung in PC-GFA-Basic.

BASIC-Online-Handbuch

Im Interpreter abrufbar: alle GFA-BASIC-Befehle mit Syntax und Erläuterungen, geordnet alphabetisch oder nach Sachgebieten, ASCII-Codes, Fehlercodes, Scancode-Tabelle, Füllmuster, Linienstile, trotzdem nur minimaler Speicherbedarf.

Dokumentations-Prozessor

Automatische Erstellung einer Programmdokumentation, Seitennumerierung und Inhaltsverzeichnis, aufwendig formatierter Programmtext mit frei wählbaren Schriftattributen für Befehle, Variablen, Kommentare, Zeilennummern, zu jeder Prozedur vollständige Kreuzverweisliste mit Seitenangaben.

ergo! *verkürzt Ihre Entwicklungszeiten*

ergo! *begleitet Sie von der Programmidee bis zur abschließenden Dokumentation*

ergo! *findet alte Fehler und verhindert neue!*

ergo! *verbannt Handbücher und Tabellen vom Schreibtisch*

ergo! *ist das unentbehrliche Hilfsmittel für Einsteiger, Aufsteiger und Profis!*

ergo! *sorgt für Durchblick!*

ergo! *vermittelt ein neues Programmiergefühl!*

ergo! *ist ergonomisches Programmieren!*

DM 148,--

unverbindlich empfohlener Verkaufspreis

Entwickelt von:

Columbus Soft

Vertrieb:

Heim Verlag

Heidelberger Landstraße 194
6100 Darmstadt-Eberstadt
Tel: (0 61 51) 5 60 57
Fax: (0 61 51) 5 60 59

In Österreich bei: Reinhard Temmel Ges.m.b.H. & Co KG, St. Julienstraße 4a, A-5020 Salzburg
In der Schweiz bei: DTZ Data Trade AG, Landstraße 1, CH-5415 Rieden-Baden

GFA-BASIC ist eingetragenes Warenzeichen der Firma GFA- Systemtechnik.

32 TAUSEND FARBEN

Die unglaubliche Grafikkarte rüstet auf
Mehr Leistung für weniger Geld

Mit 256 aus 16,7 Millionen Farben bis zu 1280 x 800 Pixeln und mit 16 Farben bis zu 1664 x 1200 Bildpunkten wurde das professionelle Arbeiten für viele zufriedene Kunden zum Erlebnis. Der Video-Mode-Generator zum Erstellen beliebiger, auch virtueller Auflösungen setzte Maßstäbe in der Monitoranpassung. Die schnellen VDI-Treiber verblüfften Fachleute, die Presse und die Konkurrenz.
Jetzt bringt das 32K-Erweiterungsmodul 32.768 Farben gleichzeitig auf den Bildschirm und ermöglicht damit zusätzliche Farbtreue in der elektronischen Bildverarbeitung.

CRAZY DOTS

Crazy Dots 256 Farben für
Mega ST oder Mega STE/TT
1298,-

Crazy Dots 32K für
Mega ST oder Mega STE/TT
Inklusive 32.728 Farb-Modul
1498,-

TKR

Stadtparkweg 2 • WD-2300 Kiel 1
☎(0431) 33 78 81 • FAX (0431) 3 59 84

Schweiz: EDV-Dienstleistungen ☎ (01) 784 89 47
Niederlande: Data Skip ☎ (018) 202 05 81

Schnelle UNFILL-Routine

In den meisten Fällen wird eine Routine, die schnell arbeiten soll, in Assembler programmiert. Auch bei der hier vorgestellten ist es so. Daß aber auch in Hochsprachen Geschwindigkeitssteigerung durch Verwendung von trickreichen Algorithmen erreicht werden können, zeigt die Entwicklungsgeschichte dieser Routine. Am Anfang stand ein BASIC-Programm, das zum Löschen eines rechteckigen Bildschirmbereiches 18 Minuten (1080 Sekunden) benötigt.

Sven Geier / Dirk Haun

Durch Anwendung des unten aufgeführten Algorithmus' ließ sich die Geschwindigkeit auf 11 Sekunken (100mal schneller!) verkürzen. Noch schneller (Faktor 10) funktioniert nun die Assembler-Routine.

Der UNFILL-Algorithmus

Der Bildschirmbereich, in dem alle gefüllten Flächen beseitigt werden sollen, wird Zeile für Zeile abgetastet, wobei in jeder Zeile jeder Punkt und sein Nachfolger daraufhin überprüft werden, ob sie die gleiche Farbe hatten oder nicht. Wenn nicht, befindet sich an dieser Stelle eine Kante, und der Punkt wird schwarz gefärbt; ansonsten sind die beiden Punkte innerhalb einer Fläche und werden weiß gefärbt. Diese Operation entspricht einer XOR-Verknüpfung der beiden Bildpunkte.

Wenn man sich nun nach einem Durchlauf das Ergebnis anschaut, stellt man fest, daß eine kleine Unschönheit auftritt: waagrechte Linien werden auf zwei Punkte reduziert, was dazu führt, daß geschlossene Formen oben und unten offen erscheinen (siehe Bild 1, Teil C). Abhilfe schafft hier nur ein nochmaliges Abtasten des ursprünglichen Bildes, aber diesmal senkrecht. Die beiden so entstandenen Bitmaps werden nun übereinandergelegt, also oder-verknüpft.

Funktionsweise der Routine

Nach den üblichen Initialisierungen (Stack einrichten; nicht benötigten Speicher freigeben), wird das Bild in den Puffer geladen (liegt im BSS-Segment). Anschließend wird die Bildschirmadresse ermittelt und das Originalbild dorthin kopiert.

In der ersten Schleife („exor1") wird die vertikale Abtastung direkt auf dem Bildschirm durchgeführt. Dabei werden immer 32 nebeneinander liegende Punkte auf einmal mit ihren Nachfolgern in y-Richtung EXOR-verknüpft.

Die zweite Schleife („exor2") führt die horizontale Abtastung an der Kopie des Originalbilds im Puffer durch. Dabei wird ein Langwort, also 32 aufeinanderfolgende Pixel, um ein Bit nach links geschoben; so hat man für zumindest 31 Pixel auf einen Schlag die Nachfolger in x-Richtung ermittelt. Was aber ist mit dem untersten Bit? Hier muß doch offensichtlich das oberste (nach üblicher Zählung das 31.) Bit des im Speicher nachfolgenden Langwortes eingefügt werden. Damit steht auch schon fest, daß die Abtastung mit dem letzten Langwort des Bildschirmspeichers beginnen muß. Und damit das beim Schieben herausfallende oberste Bit nicht verlorengeht, benutzt man den ROXL-Befehl des 68000. Durch mehrmaliges Hintereinanderausführen dieses Befehls wird jeweils das oberste Bit eines Langworts im x-Flag zwischengespeichert und bei der nächsten Ausführung als unterstes Bit in das nächste Langwort hineingeschoben (damit eignet sich dieser Befehl auch wunderbar für Laufschriften). Zurück zum Programm. Wir haben nun die Ergebnisse der beiden Abtastungen im Speicher stehen, jetzt müssen sie nur noch übereinandergelegt werden. Dies geschieht in der letzten Schleife („orloop").

Die Routine hat allerdings zwei 'Haken': Zum einen funktioniert die Vorgehensweise nur auf Monochrom-Bitmaps. Und zum anderen können mit der vorgestellten Routine nur Blökke „entfüllt" werden, deren x-Koordinaten ein Vielfaches von 32 sind.

Routine: unfill
Funktion: „entfüllen" aller gefüllten Bildschirmbereiche

Radieren

Dieser Beitrag beschreibt eine Routine, mit der eine wichtige Standardfunktion jedes Grafikprogramms realisiert werden kann: der 'Radiergummi' zur Korrektur von Fehlern, die bei der Erstellung von Grafiken zwangsläufig vorkommen.

Hans-H. Ackermann

Prinzipiell teilt sich die Routine in zwei Abschnitte: Im ersten wird die Dimension der Radierbox bestimmt. Danach kann sie an jeder beliebigen Position auf dem Bildschirm mit der Maus plaziert werden; sie markiert den zu löschenden Bildausschnitt. An dieser Stelle tritt die Löschbox (der innere Bereich der Radierbox) in Aktion und 'radiert' den Inhalt des gewählten Ausschnitts. Damit sich der Bildschirm nach eventuellen Fehllöschungen wieder restaurieren läßt, ist außerdem eine UNDO-Funktion vorgesehen.

Radierbox

Nach dem Aufruf der Prozedur Radieren erscheint am linken Bildschirmrand eine Box mit voreingestellter Größe. Diese Größe kann nach Prinzip des 'Gummibands' (Rubberbox) mit der Maus auf die gewünschten Maße gebracht werden. Die Dimensionierung wird durch Mausklick beendet. Nun kann die Box (Mauszeiger am linken oberen Eck) an jeder beliebigen Stelle auf dem Bildschirm positioniert werden.

Löschbox

Nachdem die gewünschte Stelle angefahren wurde, wird, nach einem Druck auf die linke Maustaste, die von der Radierbox markierte Fläche gelöscht. Zuvor wird der Inhalt für ein eventuelles UNDO zwischengespeichert. Nun kann entweder durch Drücken der UNDO-Taste der Bildschirmausschnitt wieder restauriert werden, oder Sie können die Box an eine weitere Stelle zum Radieren (linke Maustaste anklicken) schieben. Ist Ihr Radiervorgang beendet, kommen Sie durch Anwahl der rechten Maustaste in das Hauptprogramm zurück.

Aufruf: Radieren
Funktion: Mit der Maus können die Größe einer Radierbox festgelegt und anschließend Bildschirmausschnitte gelöscht werden (Restaurierung mit UNDO-Taste möglich).
Parameter: keine

ComBase

Standard
- ✓ Variables Datenbanksystem
- ✓ Schneller Zugriff auch auf große Datenmengen
- ✓ Parallelbetrieb von Datenbanken, Masken, Listen und Zusatzprogrammen (Multitasking)
- ✓ Multiuserbetrieb in Netzen (z.B. Bionet, ATARI Net, PamNet)
- ✓ Mehrfachsortierung auf 4 Ebenen
- ✓ Programmierbar in drei Stufen
- ✓ Leistungsfähige Wahl- und Rechendefinitionen
- ✓ Frei definierbare Masken mit grafischen Elementen

Der Unterschied
Schnelles, ausbaufähiges Multitasking Datenbanksystem ComBase in einer leicht zu bedienenden Fensterumgebung. Dabei kann man jederzeit von einer Aufgabe zur nächsten umschalten (sog. Multitasking). Man muß also nicht erst eine Maske vollständig ausfüllen und abspeichern, bevor man etwas anderes tun kann, z.B. schnell nach einer Telefonnummer suchen. So lassen sich spezielle Anwendungsprogramme einfach in das ComBase-System integrieren und sind jederzeit verfügbar.

Einfache Bedienung
Durch die grafische Bedienung erlernt man das Datenbanksystem in kurzer Zeit. Funktionen lassen sich auch mit Tastaturkommandos aufrufen.

Geschwindigkeit
ComBase ist schnell. Selbst bei größten Datenmengen wird eine hohe Geschwindigkeit erreicht. Dies geschieht z.B. durch die Verwendung von sogenannten Schablonendateien.

Daten
Datenbankkern: FlashAccess
Bis zu 40 Datenbanken (Netzwerk 400)
Max. Datensatzgröße 2 GigaByte
Max. 65536 Indizierungen pro Datei
Max. 2 Milliarden Datensätze pro Datei
Index-Cache (nur durch Speicher begrenzt)
Multiple Record-Locking

Programmierbar in 2 Stufen:
1. Durch Algorithmen, einer Programmierweise, mit dBASE ähnlichen Befehlen, die in das ComBase System homogen eingebunden wurde. Diese werden durch Funktionstasten in Masken oder automatisch gestartet (z.B. als Rechenfunktion)
   Ideal für alle herkömmlichen Datenbankanwendungen, Branchenprogramme, individuelle Lösungen,...
2. Nachladen von SPC Modula-2 Programmen, die das gesamte ComBase System mitbenutzen können und 'multitaskend' neben den Standardwerkzeugen laufen! Komplexe externe Spezialprogramme (Bildbearbeitung, Gerätesteuerung) die auf ComBase aufbauen.

Frei definierbare Masken (Init)
- Titel für Überschriften
- Felder vom Typ Text, Zahl, Datum, Zeit, Geld, Radio Button, Check-Box, Logisch und Extern (z.B. Bilder oder andere Objekte)
- Grafikelemente - Linien Rahmen, Füllung,
- Makro-Taste, um einen Text auf einen Button zu legen (z.B. »Sehr geehrter Herr« auf den Button [Herr])
- Frei belegbare Funktionselemente, die durch Maus- und Funktionstasten bedient werden können. Mit diesen Buttons werden auch evtl. definierte Algorithmen ausgelöst.

Alle Texte können in verschiedenen Fontgrößen (7-20 Punkt und, falls vorhanden auch GEM-Fonts) sowie in verschiedenen Farben dargestellt werden.
Alle Objekte sind frei plazierbar, Masken beliebig groß (Fenster), Definition von Reihenfolge und Sichtbarkeit, virtuelle Felder, geschlüsselte Mehrfachfelder, Verbundmasken, die aus Elementen mehrerer Dateimasken zusammengesetzt sind.

Datenbank Persönlich
Dateien, Verbunde, Algorithmen und alle persönlichen Einstellungen werden zu einem »Worksheet« zusammengefaßt. So kann für jeden Anwender ein persönliches Worksheet zusammengestellt werden (für Sekretärin, Mitarbeiter, Chef). Dies ist bei Multiuserbetrieb besonders wichtig.

Worksheets/Relationen
Jede Datei kann beliebig oft mit anderen Dateien in verschiedenen Zusammenhängen verwendet werden. Komfortable 'REL' Funktion erlaubt durch einfaches Ziehen von Verbindungen beliebige Verknüpfungen von Dateien. Exportrelationen erlauben das Übertragen von Daten zwischen Dateien (z.B. aus "Lager" in "Bestell")

Listen und Masken
Zu jeder Datei gehört mindestens ein Maskenfenster und ein Listenfenster, das die Daten editierbar in Listenform darstellen kann (BROWSE). Jede Datei hat ein eigenes Clipboard, um Datensätze zusammenzustellen.

Datensicherheit
Eine "Mirrordatei" wird auf Wunsch automatisch mitgeführt, um nach jeder Datenbankänderung ein Spiegelbild der aktuellen Datei auf einer anderen Partition zu haben.

Datenaustausch
Zwischen Dateien. Daten können aus der Adress- und Lagerdatei automatisch in eine Rechnungsdatei übertragen werden, um Rechnungsformulare auszufüllen. Diese Funktionen lassen sich auch mit Algorithmen programmieren.

Export
ASCII IM- und Export in allen denkbaren Formaten (Adimens, dBase). Formate auch selbst definierbar. Listen und Serienbriefexport an Textverarbeitung wie That's Write. Dort sind komplexe Gestaltungen oder Ausgabe z.B. an PostScript möglich.

Verkaufspreis 398,- DM*
*Unverbindlich empfohlener Verkaufspreis

CoCom

Ein erweiterter Desktop aus deutscher Entwicklung, der den bisherigen ST- und auch den TT-Desktop ersetzt und um viele sinnvolle Funktionen erweitert.

Der freundliche Desktop
Disketten- und Festplattenlaufwerke, Mülleimer, Drucker, Modem. Erweiterung der Fensterbedienung. Verschieben und Kopieren von Dateien. Mit und ohne Umbenennung. Gelöschte Dateien aus dem Papierkorb wieder retten oder auch endgültig löschen. Das alte ANZEIGEN/ /DRUCKEN/ABBRUCH wurde gegen komfortable Fenster mit variabler Buchstabengröße, Suchen, mit und ohne Zeilennummern,... auch mehrere Texte in verschiedenen Fenstern gleichzeitig. Bilder verschiedener Grafikformate werden automatisch erkannt und ebenfalls angezeigt. Das Drucker-Icon erlaubt Ausdruck mit verschiedenen Optionen, Rnder, Tabulatoren,... Farben für Desktop, Fenster und Icons können eingestellt werden.

Eigene Icons
Eine große Icon-Sammlung ist dabei (auch in Farbe). Diese kleinen Pictogramme kann man einzelnen, aber auch Gruppen von Dateien zuweisen. Vieles ist bereits voreingestellt. So findet man Icons für COMPO Software Produkte genauso, wie Symbole für viele andere Programme. Auch Dokumenttypen können Icons zugewiesen werden. DTP-Dokumente, Texte, Vektorgrafiken, Rastergrafiken, ...

Aktive Icons
Wählt man z.B. drei Texte an und schiebt diese auf das That's Write Icon, wird das Programm gestartet und lädt diese Texte.
Auch Datei- und Ordner-Icons können auf dem Desktop abgelegt werden. Dadurch entfällt unnötiges öffnen von Fenstern.

Icons verschiedener Größen
Da Icons fast beliebige Größe haben dürfen, gibt es neben den 'kleinen' auch große Icons für z.B. Großmonitore. Natürlich auch farbig. Die Icondatei läßt sich mit einem RCS oder auch einem Iconeditor bearbeiten.

Persönlicher Desktop
Ein Menü erlaubt das Laden und Sichern von Desktop-Konfigurationen, sodaß verschiedene Anwender am gleichen Rechner jeweils Ihren persönlichen Desktop haben können.

Tasten und Funktionstasten
Alle Funktionen (auch in den verschiebbaren Dialogboxen) können per Tastatur bedient werden. Die Funktionstasten können mehrfach mit Programmen und Funktionen! belegt werden.
Auch Fenster und Dateien können per Tastatur bedient werden. Dazu erscheint ein Datei-Cursor im Fenster. Mit Space kann man Dateien selektieren, mit Return starten oder Ordner öffnen, oder mit Insert Fenster wechseln. Selektierte Dateien bleiben dabei angewählt. Backspace schließt den Ordner, Delete das Fenster. Man entdeckt lauter Kleinigkeiten, die das Arbeiten sehr angenehm machen. Ideal z.B. für 'STACY' oder 'Book' unterwegs ohne Maus.

Script-Dateien
Eine Stärke von CoCom sind Scripte, ASCII-Dateien, die im Gegensatz zu herkömmlichen Batch-Dateien volle Kontrolle über den Desktop haben. Dialogboxen für Eingaben, Fenster für Ausgaben und Kommandos für alle Funktionen des Desktops grenzen an eine einfache Programmiersprache. Damit können Sie wiederkehrende Abläufe auf einen Tastendruck reduzieren.

Der Speicherplatz
Kein Problem, da je Programm definiert werden kann, ob CoCom im Speicher bleibt, oder ausgelagert wird.

Für engagierte Anwender
Per Tastendruck kann man auf einen UNIX angelehnten Commandointerpreter mit beachtlichen Befehls- und Funktionsumfang umschalten, der integriert ist. Eine ideale Entwicklungsumgebung. CoCom kennt XARG-übergabe und XACC-Protokoll von Accessories wie EasyBase oder That's Address.

Pull-Down-Menüs
Diese beschränken sich auf Voreinstellungen, sowie Anmelden von allen/einem Laufwerk und Werkzeugen, sowie Konfiguration der Icons. Alles Wichtige geschieht ber 'PopUp' an Ort und Stelle: Unnötige Mausbewegungen werden so drastisch reduziert.

Pop-Up Menüs
CoCom PopUp Menüs erkennen Ihre Umgebung. Das bringt übersicht und vereinfacht die Bedienung noch mehr. Die Menüs erkennen, wann Sie angefordert werden und bieten nur die passenden Optionen an. Auf Disketten zeigt das PopUp Optionen von der Anzeige des freien Platzes, über Diskcopy, Virenschutz bis Löschen und Formatieren (auch HD- und Fett-Option -voreinstellbar). Auf Festplattenicon dagegen zeigt das PopUp eine schnelle DateiSuchfunktion, Directory-Tree, Datensicherung, Platz-Statistik... Auch Fenster und Ordner haben ein eigenes PopUp mit Dateimaske (zeigt nur noch bestimmte Dateien), Sortieren und Darstellungsart (Icons oder Text, mit/ohne Datum, Länge, Attribute,...).

Lieferumfang und Hardware-Unterstützung
Neben Handbuch und Programm liegt eine Diskette bei mit einer sinnvollen Grundausstattung an Hilfsprogrammen wie Archivierer, Backup, Kopierprogramm,... Natürlich können auch eigene Programme eingebunden werden.
Alle ATARI ST/STACY/STE/TT mit Festplatte. Auflösungen ab 640x200 Punkte Monochrom und Farbe. Farbschirme, Großbildschirme, MegaScreen und OverScan werden unterstützt.

Firmen, Entwickler, EDV-Berater, Händler,
CoCom gibt es auch als OEM-Lizenz-Software durch den CoCom-Compiler erhalten Sie die Möglichkeit, kundenspezifische Versionen individuell zu erstellen. So z.B. für den Netzwerkeinsatz oder kundenspezifische PopUp Menüs,

Verkaufspreis 148,- DM*
*Unverbindlich empfohlene Verkaufspreis

Vertrieb in Deutschland: Heim Verlag, Heidelberger Landstr. 194, 6100 Darmstadt 13, Telefon 06151-56057, Fax 06151-56059.
Vertrieb in der Schweiz: DataTrade, Landstraße 1, 5415 Rieden/Baden, Telefon 056-821880, Fax 056-821884.
Info: COMPO Software GmbH, Ritzstraße 13, 5540 Prüm, Telefon 06551-6266, Fax 06551-6339.

Farbpalettenwechsel

Die in diesem Artikel vorgestellten Demoprogramme sollen zeigen, wie Sie in Ihren Programmen Farbpalettenänderungen benutzen können, um Zeichnungen zu animieren oder Farbeffekte auf dem Monitor darzustellen.

Christen Fihl

Wenn man auf dem Bildschirm zeichnet, ist es notwendig, die Farbe anzugeben, mit der gezeichnet werden soll. Die Farbnummer (0 bis 15) wird nicht direkt angezeigt, sondern zeigt auf den entsprechenden Inhalt einer Farbpalettenzelle. Die dort stehende Farbe erscheint dann auf dem Monitor. Um nun die Farbe eines Bildschirmpunktes zu ändern, ist es nicht notwendig, ihn neu zu zeichnen, sondern nur den Inhalt der Palettenzelle zu ändern. Allerdings werden dann auch alle anderen Pixel, deren Farbwert auf diese Zelle zeigt, in der neuen Farbe abgebildet!

Für Animationen ist es charakteristisch, daß Bilder auf dem Monitor schnell wechseln, ohne daß man sieht, wie sie aufgebaut werden. Dieser Effekt ist z.B. dadurch möglich, daß man die Farbpalette ändert. Natürlich dauert es immer noch genauso lange, ein Bild aufzubauen, aber der Effekt läßt sich für den Anwender des Programms unsichtbar programmieren. (Eine andere Möglichkeit bietet die Verwendung von mehreren Bildschirmen, von denen einer gezeigt wird, während auf den anderen gezeichnet wird; siehe Artikel über Hintergrundprogrammierung im Grundlagenteil dieses Sonderhefts.)

Um unsichtbare Linien zu ziehen, muß die Farbe der Linie mit der des Hintergrunds übereinstimmen. Deshalb ist es notwendig, die Anzahl der Farben auf zwei oder vier zu beschränken. Im Beispiel-Animationsprogramm werden vier Farben benutzt, was bedeutet, daß zwei Figuren animiert werden können (4*4 = 16 Farben). Durch die Benutzung von lediglich 2 Farben lassen sich vier Figuren animieren (2*2*2*2).

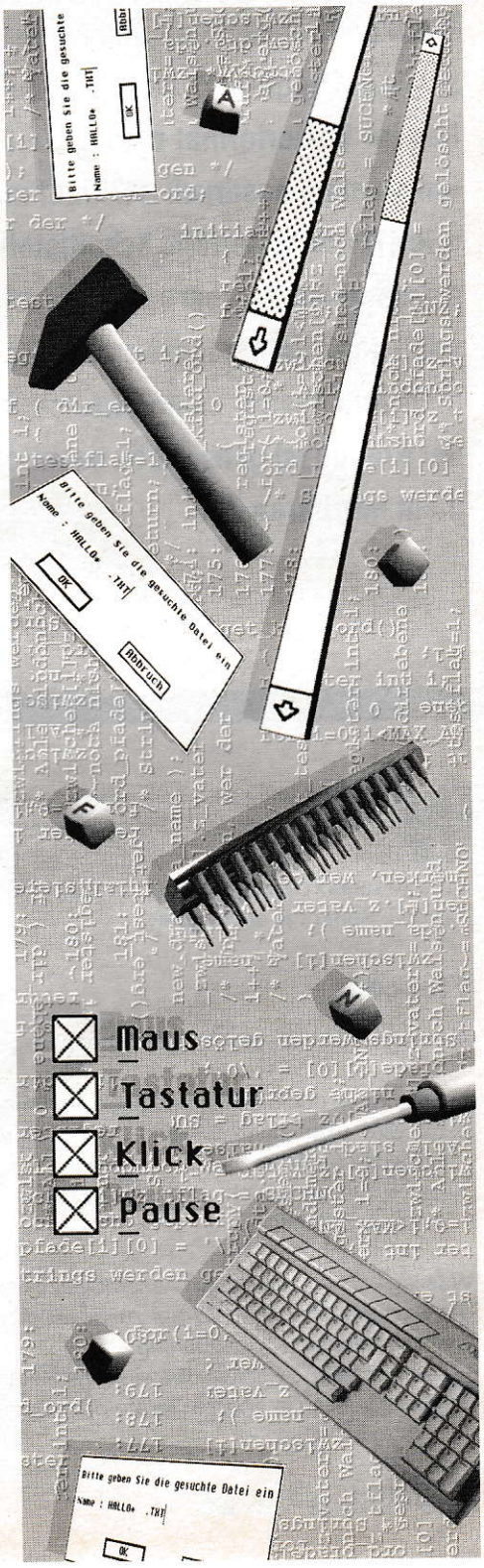
Nehmen wir an, daß zwei Figuren auf dem Bildschirm zur Darstellung kommen. Wir wollen die Bilder jeweils einzeln anzeigen. Das erste Bild wird in den Farben 0, 1, 2 und 3 dargestellt, das andere in den Farben 0, 4, 8, 12. Beim Aufbau des Bildes muß der Schreibmodus entweder auf OR oder auf XOR gestellt werden. Dadurch verhindert man, daß das zweite Bild das erste zerstört.

Durch das Tauschen der Farbpaletten (hierbei bezeichnet F1 die erste Farbe, F2 die zweite und so weiter) [F1, F2, F3, F4, F1, F2, F3, F4, F1, F2, F3, F4, F1, F2, F3, F4], [F1, F1, F1, F1, F2, F2, F2, F2, F3, F3, F3, F3, F4, F4, F4, F4] und [F1, F2, F3, F4, F3, F2, F3, F4, F3, F2, F3, F4, F1, F2, F3, F4] können beide Bilder separat angezeigt werden. Die erste Palette stellt nur Bild 1 dar. Hierzu muß für alle Farbbitkombinationen, die den selben Farbwert des Bildes enthalten auch die gleiche Farbe gesetzt werden. Dadurch ergaben sich die Kombinationen —00 (0, 4, 8, 12), —01 (1, 5, 9, 13), —10 (2, 6, 10, 14) und —11 (3, 7, 11, 15). Bild 2 wird durch die zweite Farbpalette angezeigt [Farben: 00— (0, 1, 2, 3), 01— (4, 5, 6, 7), 10— (8, 9, 10, 11) und 11— (12, 13, 14 15)]. Diese Werte kamen zustande durch das Prüfen der Bitpattern, die mit 4 Bits möglich sind. Das erste Bild benutzt dabei die Bits 1 und 0 (XX), während das andere die Bits 3 und 2 (YY) verwendet. Die möglichen Kombinationen sind dabei YY00, YY01, YY10, YY11 (Farben des Pixels des ersten Bildes, bei beliebiger Farbe (des gleichen Bildpunktes) des zweiten Bildes) und 00XX, 01XX, 10XX, 11XX.

Das Animationsprogramm schaltet zwischen zwei Bildern hin und her. Das Palettenprogramm erstellt einige psychedelische Grafiken. Beide Programme setzen natürlich einen Farbmonitor voraus.

Systemerweiterungen

Im Grundlagenteil finden Sie eine Reihe von Artikeln, die sich mit der Programmierung mittels Betriebssystemfunktionen beschäftigen. Ein Schwerpunkt liegt dort in der GEM-Programmierung. Auf den folgenden Seiten finden Sie 7 Beiträge, die sich damit befassen, wie man mehr aus dem Betriebssystem herausholen bzw. wie man es ergänzen kann. So stellen wir Ihnen eine Library zur Dialogboxbehandlung, die neben den Standard- auch Dialoge mit beweglichen und Pop-Up-Boxen durchführen kann, vor. Desweiteren besprechen wir Routinen zu den Themen effiziente Speicherverwaltung und weiteren interessanten Programmierbereichen.



Ausgabeumlenkung via BIOS

Wem geht es nicht ab und an so: Man hat ein schönes Programm, das ebenso schöne Grafiken im Vektorformat erzeugt und daher ewig langsam im Ausdruck ist? Oder man programmiert für seine serielle Schnittstelle eine neue, extrem schnelle Übertragungsroutine, die aber nicht ganz richtig funktioniert. Leider braucht man zum Testen einen zweiten Rechner, und der Freund ist ganz schön sauer ob der für ihn nicht nutzbaren Zeit.

Jan Starzynski

Wie schön wäre es, könnte man den Drucker oder die serielle Schnittstelle durch die Tastatur oder eine Datei oder ... ersetzen. Der erfahrene ST-COMPUTER-Leser wird sagen: Dafür gibts doch die GEM-DOS-Funktion $46 *Fforce()*. Stimmt ja auch, aber...

Fforce()

An dieser Funktion störten mich schon lange drei Mängel:

1. Es können nur GEMDOS-Funktionen umgelenkt werden.
2. Es wird nicht nach Ein-und Ausgabefunktionen unterschieden (außer CON:).
3. Es können auch nicht alle verfügbaren Geräte umgelenkt werden (z.B. MIDI).

Was lag näher, als sich den Assembler zu nehmen und sich der Sache anzunehmen? Aber zunächst einmal zu den bekannten Sachen, sprich *Fforce()*.

Diese GEMDOS-Funktion bewirkt wie bekannt das Umlenken von Geräten in andere Geräte oder Dateien. Ein kurzes Beispiel in C:

```
#define CON_IN  0
#define CON_OUT 1
#define AUX     2
#define PRN     3
/* womit wir auch schon
   alle behandelbaren
   Geräte erwähnt hätten */

main()
{
 int han;
 han=Fcreate("test.dat",0);
 Fforce(PRN,han);
 Cprnout('a');
 Fclose(han);
 return 0;
}
```

Es passiert im Programm nichts weiter, als daß die Datei TEST.DAT geöffnet und der Drucker dorthin umgelenkt wird. Wenn nun ein Zeichen auf den Drucker ausgegeben wird, landet es nicht auf diesem, sondern in unserer Datei, und man kann in der Datei nach Beenden des Programmes den Buchstaben 'a' bewundern. So weit, so gut. Jetzt weiß ich also, wie ich in meinem eigenen Programm den Drucker „verbiege". Das hilft mir aber beim Grafikprogramm nicht weiter. Schließlich habe ich das nicht selbst geschrieben und der Hersteller wird wohl auch kaum den Quelltext verschikken. Die Hilfe ist einfach: Diese „Geräteverbiegung" kann weitervererbt werden. Ersetzt man also das

```
Cprnout('a')
```

im Beispiel durch z.B.

```
Pexec("graphik.prg",command,"",0)
```

so wird GRAPHIK.PRG gestartet und die Grafik, die sonst auf dem Drucker gelandet wäre, liegt dann in unserer Datei, wo sie ja gut aufgehoben ist. Nachdem unser Programm beendet wurde, ist wieder alles in bester Ordnung, das Betriebssystem sorgt selbst dafür, daß der Drucker druckt und nicht mehr „dateit". Und wie drucke ich dann das ganze aus? Ganz einfach: vom Desktop aus die Datei zweimal anklicken und

die Frage *ansehen* | *drucken* | *Abbruch* mit „drucken" beantworten. Dann geht die Post ab, und der Drucker wirft die schönste Grafik aufs Papier. Damit hätten wir also die einfache Lösung kurz nochmal abgeschnitten. Aber wie gesagt, das funktioniert sofort nicht mehr, wenn unser Grafikprogramm die Grafik mittels BIOS ausgibt. Dann müssen wir nach wie vor eine halbe Stunde Rechenzeit in Kauf nehmen. Und für andere Anwendungen bestehen immer noch die oben genannten Mängel. Und hier setzt mein BIOS.PRG an.

Jetzt komme ich

Was eigentlich das Programm bewirken soll, ist ja nun klar. Und weil ich *Fforce()* nicht für so schlecht halte, sollte die Wirkung auch ähnlich sein. Ich habe also extra für dieses kleine Programm eine neue BIOS-Funktion implementiert, die bei mir *Bforce()* heißt und die Dateiumlenkung auf BIOS-Ebene bewirkt und im Gegensatz zu *Fforce()* auch noch zwischen Ein- und Ausgabe unterscheidet. Folglich gibt es drei Parameter:

1. der umzulenkende Kanal
2. wohin der Kanal umzulenken ist
3. ob Ein- oder Ausgabe gemeint ist

Um das Kapitel der Benutzung gleich ganz abzuhaken, nun noch das Schema des Aufrufs der Funktion in C und Assembler:

```
#define Bforce(a,b,c) bios(132,a,b,c)
Bforce(device,newdevice, in_out);
move.w in_out,-(sp)
move.w newdev,-(sp)
move.w device,-(sp)
move.w #132,-(sp)
trap   #13
addq.l #8,sp
```

Die Parameter haben dabei folgende Bedeutung:

in_out: 0 - Eingabe umlenken
1 - Ausgabe umlenken

newdev: BIOS- bzw. Datei-Handle des Umlenkungszieles

device: BIOS-Handle des umzuleitenden Kanales. Ist *newdev:* größer oder gleich sechs, handelt es sich um ein Datei-Handle, sonst um eines der folgenden BIOS-Handles.

0 Drucker
1 serielle Schnittstelle
2 Konsole
3 MIDI
4 IKBD
5 RAW-Konsole

Datei-Handles sind automatisch größer als sechs, darüber braucht man sich also keine Sorgen zu machen. Wen jetzt die technischen Feinheiten nicht interessieren (sog. Nutzer), der kann aufhören zu lesen. Allerdings sollte er sich die Warnungen am Ende des Textes noch durchlesen.

Programmierung

Dem geehrten Leser stehen natürlich noch alle anderen Wege offen. Aber so funktioniert es in etwa: Mit einer Funktion namens *install* klinke ich mich in den BIOS-Vektor (XBRA-Verfahren; siehe Artikel in diesem Heft) und fange erstmal alle BIOS-Aufrufe ab. Danach werden alle mich interessierenden Funktionen ausgefiltert, als da wären

Name	Nummer
1. Bconstat	0
2. Bconin	2
3. Bconout	3
4. Bcostat	8
5. Bforce	132

Das sind alle BIOS-Funktionen, die mit Ein-und Ausgabe auf Geräte zu tun haben. 1. und 4. sind Erkundigungsfunktionen, die den Status von Geräten feststellen, 2. und 3. sind die Ein- und Ausgabefunktionen, und 5. schließlich ist hier gerade im Entstehen. Zuerst beschäftigen wir uns mit 5.

Wird diese Funktion aufgerufen, wird anhand von *device* in die Ein- oder Ausgabetabelle der Wert von *newdev* eingetragen. *device* wird dabei als Offset in der Tabelle verwendet.

Wird nun allerdings 1., 2., 3. oder 4. erkannt, liegt als Funktionsparameter auch immer die *device*-Nummer auf dem Stack. Diese wird vom neuen BIOS-Handler als Offset in die Ein- oder Ausgabetabelle verwendet und die dort stehende Nummer anstelle des alten Wertes auf den Stack gelegt. Kurz gesagt, wird die Funktionsnummer ausgetauscht. Danach wird der BIOS-Aufruf weitergeleitet an den ursprünglichen BIOS-Handler, der dann ganz unbekümmert mit dem untergemogelten Gerät weitermacht, als wäre nichts passiert. Das Programm, das den Aufruf tätigt, bekommt auch nichts von der kleinen Manipulation mit.

So wie bis hier dargestellt, funktioniert das ganze aber nur für Umleitungen innerhalb der sechs BIOS-Kanäle. In Dateien kann man damit aber noch nichts schreiben. Da aber Datei-Handles gut zu erkennen sind (>=6), wird in diesem Fall wie folgt vorgegangen:

Umleitung in Dateien

Für Ein- oder Ausgabefunktionen wird die in der Tabelle gefundene neue Gerätenummer als Datei-Handle erkannt und zur weiteren Bearbeitung das GEMDOS genutzt. Der zur Ausgabe vorgesehene Wert wird also in einen Ein-Byte-Puffer übertragen und mittels *Fwrite()* in die ausgewählte Datei ausgegeben. Bei der Eingabe wird der einzulesende Wert mittels *Fread()* in den Puffer eingelesen und danach in D0 übertragen, wo das BIOS alle Funktionswerte zurückgibt. Die Erkundigungsfunktionen sind in diesem Fall nur Dummy-Funktionen, da sie immer wahr zurückgeben. Es gibt also hier einen „Rückschritt" vom BIOS zum GEMDOS, der allerdings nicht ganz problemlos funktioniert, da das GEMDOS auf das BIOS zurückgreift. Wenn nämlich ein BIOS-Aufruf des GEMDOS erfolgt, und dieses wiederum das GEMDOS konsultiert, werden die internen Speicher des GEMDOS gnadenlos überschrieben, und man kriegt das große Grübeln, wo plötzlich die ganzen Bomben herkommen. Aber man kann sich ja

helfen. Die Lösung ist zwar nicht so ganz elegant, funktioniert aber. Es wird nämlich auch noch der GEMDOS-Vektor überwacht, und sobald eine Funktion [außer *Pexec()*] angesprungen wird, ignoriert unsere BIOS-Routine alle Umleitungen in Dateien, so daß das GEMDOS ungestört arbeiten kann. Dadurch ist zwar die Umleitung nicht mehr ganz so mächtig, aber dafür sicherer.

Das war's im Prinzip auch schon. Als Zugabe ist noch ein kleines Programm dabei, das die BIOS-Routine resident im Speicher ablegt. Und zum Schluß nun noch die versprochenen Warnungen.

Achtung!

Als wichtigstes und allgemeinstes: am BIOS führt (fast) kein Weg vorbei. Eine Umlenkung von Kanälen wirkt sich also überall aus, selbst bei solchen Sachen wie der Fileselectbox! Also Vorsicht bei der Anwendung, insbesondere bei Umlenkung der Standardeingabe. Des weiteren ist zu beachten, daß nicht alle BIOS-Funktionen für alle Geräte implementiert sind. Die hier vorgestellte Routine kümmert sich um solche Feinheiten nicht, aber die Original-BIOS-Fehlermeldungen kommen zurück. Das gilt nicht bei der Umlenkung in Dateien. Man sollte also immer sehen, daß die Dateiarbeit reibungsfrei ablaufen kann. Des weiteren macht das Betriebssystem die Veränderung der Kanäle nicht selbständig wieder rückgängig, sondern der Nutzer muß das selbst machen [mit *Bforce (device,device,in_out)*]. Sonst bleibt alles bestehen bis zum nächsten Reset. Die Funktionen *install()* und *exstall()*, sollte sie jemand in einem eigenen Programm nutzen wollen, müssen im Supervisormodus aufgerufen werden.

Dauerhaftes MALLOC für Accessories

Diejenigen, die schon einmal Accessories programmiert haben, mußten sich, sobald diese etwas komplexer wurden, bestimmt schon mit dem Problem herumschlagen, daß man die GEMDOS-Funktion MALLOC zur Speicherreservierung nicht so einfach anwenden kann wie in 'normalen' Programmen.

Lutz Preßler

Man muß nämlich beachten, daß beim Beenden eines Hauptprogramms automatisch auch alle mit *MALLOC()* reservierten Speicherbereiche des Accessories freigegeben werden, wenn diese während der Laufzeit des Hauptprogramms reserviert wurden. Daraus folgt, daß spätestens beim Eintreffen der *AC_CLOSE*-Message alle reservierten Speicherbereiche wieder zurückgegeben werden müssen. Wenn man nun aber z.B. eine Adreßverwaltung als Accessory schreiben wollte, hieße das, daß deren Funktion stark eingeschränkt würde, da die Adressen bei jedem Programmende wieder gelöscht werden müßten. Und nicht nur dann: Die *AC_CLOSE*-Message wird nicht nur beim Beenden eines Programms gesendet, sondern auch beim Starten eines solchen. Da ich keinen Weg kenne, festzustellen, ob nun ein Programm wirklich beendet wurde, muß man zwangsläufig auch beim Starten eines Programms den Speicher freigeben, obwohl das eigentlich gar nicht nötig wäre.

Speicherverwaltung

Das Ganze ist also wirklich nicht sehr befriedigend. Man fragt sich, ob es da nicht irgendeine Möglichkeit gibt, das Freigeben der Speicherbereiche zu verhindern. Dazu muß man sich erst einmal im klaren sein, warum es überhaupt zur Freigabe kommt, was das Betriebssystem also macht. Die Wurzel allen Übels ist, daß Accessories zwar eigene AES-Applikationen sind, aber leider unter GEMDOS keine eigenen Prozesse. Sie haben deshalb zum Beispiel auch keine vollständig gefüllte Basepage (die man aber trotzdem verwenden kann, s.u.). Das Accessory ist für GEMDOS also der gleiche Prozeß wie das gerade aktive Hauptprogramm. Nun ist es so, daß GEMDOS intern bei jedem allokierten Speicherbereich vermerkt, zu welchem Prozeß er gehört, indem er einen Zeiger auf dessen PD (Basepage) abspeichert. Beim Ende des Hauptprogramms durch *Pterm/Pterm0* werden nun seinerseits alle Speicherbereiche freigegeben, die diesem PD zugeordnet sind. (Für genauere Informationen verweise ich auf [1].) Dies ist im allgemeinen ja auch ganz sinnvoll, da bei normalen Programmen nach dem Ende der Speicher nicht mehr benötigt wird. Für residente Treiber usw. steht dann *Ptermres()* zur Verfügung, wo kein Speicher freigegeben wird. Durch den 'Designfehler' im TOS werden nun aber auch die Speicherbereiche der Accessories freigegeben, und das ist ja gerade unser Problem. Wie kann man es lösen oder umgehen? Dazu muß man wissen, daß es eine GEMDOS-Variable namens *act_pd* gibt, die immer die Adresse des PD des gerade aktiven Prozesses enthält (sprich einen Zeiger auf die Basepage des Hauptprogrammes). Wenn per *Pexec()* ein Programm gestartet wird, zeigt *act_pd* danach auf dieses Programm, nach Programmende wieder auf den 'Elternprozeß'. Bei *MALLOC()* und *Pterm()* wird nun *act_pd* benutzt, um die Zugehörigkeit der Speicherblöcke festzustellen. Haben Sie schon einen Lösungsvorschlag? Sehr gut!

Speicher sichern

Ja, die Idee ist wirklich, *act_pd* vor einem Accessory-*MALLOC*- (bzw. *Mfree*-) Aufruf umzusetzen (und zwar auf die Basepage des Accessories) und danach sofort wieder zurück. Die Folge ist, daß bei einem späteren *Pterm()* die vom

Aufruf:	accmalloc(amount)
Funktion:	reserviert Speicher am Ende des freien Bereichs (für Accessories!)
Parameter:	amount: benötigter Speicher
Rückgabewert:	Zeiger auf Startadresse des reservierten Speichers; bei aufgetretenem Fehler 0
Aufruf:	accmfree(ptr)
Funktion:	gibt mit accmalloc() reservierten Speicher wieder frei
Parameter:	ptr: Anfangsadresse des freizugebenden Speicherbereichs [Rückgabewert von accmalloc)]
Rückgabewert:	0: alles ok; -40: Speicherblock wurde nicht mit (acc)malloc() reserviert

Accessory reservierten Speicherbereiche nicht mehr freigegeben werden, da sie ja nicht mehr dem Hauptprogramm zugeordnet sind. Ein paar 'Details' sind aber trotzdem noch zu besprechen, bevor Sie diesen Weg ausprobieren können. Vor allem habe ich bis jetzt immer nur von von *act_pd* gesprochen, aber nicht gesagt, wo diese Speicherzelle liegt. Leider ist es nämlich nicht ganz so einfach wie bei den meisten anderen Systemvariablen, die schon seit Urzeiten (in diesem Fall 1985) von ATARI garantiert wurden. *act_pd* war bis zur ROM-TOS-Version 1.0 undokumentiert, lag allerdings immer an der gleichen Adresse. Diese änderte sich beim 'Blitter-TOS' 1.2 zwar, doch hatte ATARI eingesehen, daß *act_pd* eigentlich doch ganz nützlich sein könnte, und erweiterte den Systemheader u.a. dahingehend, daß ein Zeiger auf *act_pd* enthalten ist. Man muß also anhand der TOS-Version (ebenfalls im Systemheader) unterscheiden, ob ein 'altes' TOS vorhanden ist, und die Adresse von *act_pd* entsprechend bestimmen. An dieser Stelle muß ich nun auch einräumen, daß das hier beschriebene Verfahren zwar recht sauber ist und auch auf allen TOS-Versionen funktioniert, aber nicht ausschließlich nur dokumentierte Eigenschaften benutzt. Das größte potentielle Problem ist, daß *act_pd* von ATARI nur zum Auslesen freigegeben wurde, der Schreibzugriff ist eigentlich nicht erlaubt. Da *act_pd* aber ja nur für sehr kurze Zeit (ein paar GEMDOS-Aufrufe) geändert wird, könnte das höchstens bei zukünftigen Multitasking-Versionen des GEMDOS/TOS zu Problemen führen. Eine andere Einschränkung ist, daß nirgendwo dokumentiert ist, daß für die Zuordnung der Speicherblöcke zu den Prozessen *act_pd* benutzt wird. Eine andere Möglichkeit kann ich mir allerdings fast gar nicht vorstellen. Wie gesagt, diese Einwände habe ich nur der Korrektheit wegen aufgeführt.

Speichermangel?

Viel wichtiger ist dagegen folgendes: Wenn Sie das Verfahren so anwenden würden, wie ich es bis jetzt beschrieben habe, würden Sie sich evtl. bald über scheinbar unerklärlichen Speicherplatzmangel wundern, und zwar ironischerweise gerade bei Gebrauch von sauber geschriebenen Programmen. Was ich meine? Stellen Sie sich folgende Situation vor: Sie starten ein Programm, das ganz korrekt seinen überschüssigen Speicher wieder zurückgibt. Dort rufen Sie nun ihr Accessory auf, das nach obigem Verfahren 'dauerhaft' Speicher reserviert. Dann beenden Sie das Hauptprogramm. Der Accessory-Speicher ist immer noch unverändert erhalten. Also alles in Butter? Leider nicht ganz. Fragen Sie nun einmal den 'freien' Speicher ab (*MALLOC(-1)*). Er wird übermäßig stark geschrumpft sein. Was ist passiert? Das Programm hatte seinen überflüssigen Speicher zurückgegeben. Das Accessory hat sich nun den Anfang dieses Speicherblocks 'dauerhaft' reserviert. Wenn das Programm dann beendet wird, wird dieser ja nicht mehr freigegeben und damit der große freie Speicherblock in zwei Teile geteilt. Davon wird dann für das nächste Programm nur noch die größere Hälfte benutzt. Diese Situation ist natürlich inakzeptabel, was kann man also machen? Man müßte dafür sorgen, daß das Accessory nicht den Anfang sondern das Ende des freien Bereichs zugeteilt bekommt. Dann würde beim Programmende nur soviel Speicher fehlen, wie wirklich gebraucht wird. Dies ist mit einem weiteren Trick auch möglich: Es wird einfach der gesamte freie Speicher abzüglich der gewünschten Größe reserviert. Danach wird der eigentliche Speicherblock reserviert und im Anschluß daran der erste 'Hilfsblock' wieder freigegeben. Dieses Vorgehen erzielt den gewünschten Effekt, ist legal und hat noch zu keinen Problemen geführt.

Ich hoffe, daß diese Routinen für einige nützlich sind und vielleicht die Erstellung von umfangreicheren Accessories erleichtern. Eventuell habe ich hiermit ja sogar den Anstoß zu ganz neuen Projekten gegeben, was mich sehr freuen würde.

Verwendete Literatur:

[1] ST-Computer Sonderheft Nr.2 (1987): TOS intern (Alex Esser, S. 35 ff.)
[2] Reschke/Jankowski/Rabich: ATARI ST Profibuch Sybex, Düsseldorf

Effiziente Speicherverwaltungen

Haben Sie schon einmal versucht, ungefähr 191158 Objekte gleicher Größe vom Betriebssystem anzufordern? Nein? Macht nichts. Irgendwann werden Sie auch einmal dazu kommen. Aber wenn das doch der Fall war, sollten Sie diesen Artikel nicht nur überfliegen. Dieses Problem tritt meist nur bei Programmen auf, die mit dynamischen Datenstrukturen „rechnen" müssen.

Klaus Elsbernd

Speicherverwaltung ist ein Problem, daß man meist dem Betriebssystem oder der Programmiersprachen-Library überläßt. Diese gehen damit recht stiefmütterlich um. Aus diversen Artikeln u.a. in dieser Zeitschrift [1, 2] und Büchern wissen wir alle von der mangelhaften Speicherverwaltung mittels der sogenannten Memory-Deskriptoren (MDs) des GEMDOS. Also griffen die Hochsprachen-Compiler zu unterstützenden Maßnahmen in Form eigenener, vorgeschalteter Routinen, die die Anzahl der *malloc*-Aufrufe an das Betriebssystem reduzieren. Aber auch das ist nicht immer der Weisheit letzter Schluß.

Das Problem

Es wird eine große Zahl von gleich großen Speicherbereichen benötigt. Die genaue Zahl läßt sich nicht im voraus bestimmen. Sie ist von Programmaufruf zu Programmaufruf verschieden. Ebenso ist die Größe dieser Bereiche erst zur Laufzeit feststellbar. Geschwindigkeit hat das Primat über Speicherverbrauch (auch ein 1040er hat eine Menge Speicher). Dieses Problem tritt dann auf, wenn man dynamische Datenstrukturen aufbaut, deren Speicherbereiche über eine Funktion vom System (hier und im folgenden ist damit sowohl die GEMDOS-Speicherverwaltung als auch die in der Programmiersprache integrierte gemeint) angefordert werden sollen. In Pascal wird dafür die Funktion *new(*record)* verwendet.

Die Lösung...

... ist eigentlich recht einfach und wurde z.B. in einfacherer Form in [3] benutzt. (Gute Lösungen sind immer einfach.) Wenn es auch in der Problembeschreibung so klingt, als ob hier der Speicher verschwendet würde, so ist dies, wie wir weiter unten sehen werden, nicht der Fall. In der Tat arbeitet der Algorithmus speichersparender als andere, wenn nur eine genügend große Zahl von Records benötigt wird.

Im Prinzip besorgen wir uns von der Speicherverwaltung [also über *malloc()*] einen zusammenhängenden Speicherbereich (Block), der eine gewisse größere Menge unserer benötigten Strukturen umfaßt. Diese können wir als ein einfaches Array verwalten, indem wir bei Bedarf das nächste freie Element zur Verfügung stellen.

Ist der Block voll, wird ein neuer vom System angefordert, und die Zuteilung erfolgt nun aus diesem. Die Verwaltung des gerade aktuellen Blocks erledigen wir über eine Tabelle. (Eine Verkettung als sequentielle Liste durch Zeiger am Anfang des Blockes ist leicht zu implementieren.) Wir erhalten also ein zweistufiges System, über das uns Speicher zugeteilt wird. Das Ziel ist eine möglichst flexible Verwaltung, die nicht durch irgendwelche Konstanten beschränkt wird. Damit ist sowohl eine Anpassung an die eigenen Programme als auch an die dynamischen Anforderungen innerhalb eines Programmlaufes möglich. Insbesondere ist die Größe des benötigten Records flexibel. Gleiches gilt für Art und Umfang verschieden großer Datenstrukturen.

Als Informationen werden folglich benötigt:
- die Adresse der Tabelle der Blöcke *memtab*. In ihr werden die Blöcke in die Verwaltung eingeklinkt.
- der Index innerhalb der Tabelle der Blöcke *memtabidx*. Dieser zeigt auf den Block, aus dem die Zuteilung erfolgt.
- die maximale Anzahl der Blöcke *memtabsize*, die über *malloc* vom System geordert werden sollen. Diese Begrenzung ist eigentlich nicht nötig, aber sie ermöglicht eine rechtzeitige garbage-collection (Müllbeseitigung), wenn Objekte auch wieder freigegeben werden müssen. Sonst wird

Aufruf:	searchmemcntrl(size)
Funktion:	sucht nach Speicherverwaltungsstruktur oder legt diese neu an
Parameter:	size: Größe des benötigten Speichers
Rückgabewert:	Adresse auf Speicherverwaltungsstruktur
Aufruf:	freshmem(mem)
Funktion:	reserviert Speicher
Parameter:	mem: Adresse auf Speicherverwaltungsstruktur [Rückgabewert von searchmemcntrl()]
Rückgabewert:	Zeiger auf Speicherblock; bei Fehler NULL
Aufruf:	freemem(size, ptr)
Funktion:	Speicherblock freigeben
Parameter:	size:Größe des Speicherblocks ptr:Zeiger auf Speicherblock
Rückgabewert:	keiner

solange Speicher vom System angefordert, bis die Library-Routine abwinkt. Das ist insofern fatal, als es auch andere Routinen gibt, die Speicher benötigen. Auch diese könnten dann nicht mehr bedient werden. Wird vorher abgebrochen, haben sie jedoch noch eine Chance.

- die Adresse des augenblicklichen Blockes *memblockptr*. Dadurch wird die Zugriffsgeschwindigkeit auf einen neuen Record beschleunigt.
- der dazugehörige aktuelle Index *memblockidx* innerhalb des aktuellen Blockes.
- diverse statistische Informationen, wie die Anzahl der belegten Elemente aller Blöcke *memstatuentries*, die Zahl der noch freien Elemente *memstatfentries* nach einer garbage-collection, und die Zahl der zur Verfügung stehenden Elemente *memstatentries*. Letztere unterscheiden sich erst dann von *memstatuentries*, wenn das System keinen weiteren Speicher zur Verfügung stellt und man den Müll (garbage) einsammeln und verwenden muß.
- da wir verschiedene solcher Speicherorganisationen für unterschiedlich große Datenstrukturen verwalten wollen, verketten wir die Kontrollinformationen miteinander und benötigen deshalb einen Zeiger auf das nächste Speicherverwaltungselement *memnext*.

Der Algorithmus

Wenn nun ein Programm eine bestimmte Datenstruktur benötigt, die sehr häufig vorkommt, ruft es das Unterprogramm *freshmem()* auf. Es erwartet die Adresse einer oben beschriebenen Speicherverwaltungsstruktur. Um diese zu erhalten, wird in einer verketteten Liste mit Hilfe der Funktion *searchmemcntrl()* nach einer passenden Struktur gesucht. Wer will, kann bei bekannten Record-Größen diese auch fest „verdrahten" und schon passende Variablen anlegen.

searchmemcntrl() durchsucht die Liste der Speicherverwaltungsblöcke sequentiell nach dem zugehörigen Verwaltungsblock. (Das Suchverfahren bricht ab, wenn *size >= mem ->memsize* ist.) Wird keine solche gefunden, d.h. keine Verwaltungsstruktur verwaltet Speicherbereiche der Größe *sizeof(struct myrec)*, wird eine solche eingerichtet.

Diese Aufgabe erledigt *newmemcntrl()*. Die Routine ermittelt aufgrund der Größe der gewünschten Datenstruktur die Blockgröße als ein ganzzahliges Vielfaches in der Nähe von „willkürlich" 8 kB. Weiterhin werden die Verwaltungsinformationen in der Speicherstruktur auf Anfangswerte gesetzt. Nachdem *searchmemcntrl()* die neue Speicherstruktur eingekettet hat, wird diese zurückgegeben.

Zurück zu *freshmem()*. Diese Funktion soll ein Objekt einer Datenstruktur liefern. Dazu haben wir ihr ja die zugehörige Speicherverwaltung übergeben. Das Unterprogramm unterscheidet zwei Phasen:

- **Phase 1:** Es gibt noch Speicher vom System, oder die maximale Anzahl von Blöcken ist noch nicht erreicht.
- **Phase 2:** Es gibt keinen Speicher vom System mehr, oder die maximale Anzahl der vom Programm zu benutzenden Blöcke (falls es eine solche Grenze geben soll) ist erreicht. Dann werden nur noch Records aus einer verketteten Liste zur Verfügung gestellt. Diese muß dann aus den nicht mehr verwertbaren Elementen aufgebaut sein.

freshmem() überprüft zuerst, ob wir uns noch in Phase 1 befinden. Das ist der Fall, wenn das System noch Speicherplatz für die Blöcke zur Verfügung stellen konnte. Anfangs ist dies (fast) immer so. Dann wird überprüft, ob es noch freie Records innerhalb eines Blockes gibt (mem->memblockidx++ < mem->memblocksize). Ist dies nicht der Fall, wird ein neuer Block vom System angefordert und an der Stelle *mem ->memtabidx* in der Tabelle *mem->memtab* eingetragen. Der Aufruf von *memset()* löscht den Block. Zurückgeliefert wird nun der erste Record innerhalb dieses Blockes.

Es kann aber passieren, daß entweder keine neuen Blöcke mehr angefordert werden dürfen (dies ist bei entsprechender Dimensionierung der *memtab* mittels des Makros *CELLTABSIZE* ziemlich unwahrscheinlich), oder daß es keinen freien Speicherplatz mehr dafür gibt. In beiden Fällen wechseln wir

zur Phase 2. Von nun an können wir nur noch Datenstrukturen zur Verfügung stellen, die von unserem Anwendungsprogramm freiwillig zurückgegeben worden sind. Dies tat es (hoffentlich) mittels des Unterprogramms *freemem()*. Es kann auch eine sogenannte garbage-collection *collectgarb()* aufgerufen werden, die das Aufsammeln des entsprechenden Speichermülls besorgt. Nun wird immer nur von der Freispeicherliste das 1. Element zurückgegeben. Hier nun der Vollständigkeit halber eine Einschränkung des Verfahrens (die eigentlich keine ist). Die minimale Speichergröße für Records ist die Größe eines Zeigers, also im allgemeinen 4 Bytes.

Eine einfache Modifikation des Unterprogramms *freshmem()* erlaubt die sofortige Nutzung zurückgegebener Records. Aus der Phase 2 kehren wir niemals in die Phase 1 zurück.

Was bringt's

Diese Frage hat zwei Seiten. Zum einen:

- Wie steht es mit dem Speicherverbrauch? Die Verwaltung des Systems muß dem Verbrauch der Blockverwaltung gegenübergestellt werden. Aus [1] wissen wir, daß die benötigten Records in TURBO-C mittels 8 Bytes verkettet werden. Für „n" solcher Datenstrukturen benötigen wir also „8n" Bytes. Hinzu kommen, und das werden wir vernachlässigen, 16 Bytes für initiale Verwaltungseinträge je vom System benötigten Block.

Das hier vorgestellte System benötigt eine Tabelle der zu verwaltendenen Blöcke, also CELLTABSIZE*sizeof(long * *) == 4*CELLTABSIZE Bytes, sowie eine Verwaltungsstruktur [sizeof(struct memcntrl) == 46 Bytes] und zusätzlich sizeof(sysmem) == 4 Bytes. Weiterhin sind die vom System benötigten Verwaltungs-Bytes zu berechnen. Diese schätzen wir mit jeweils 8 Bytes je Block nach unten ab. Wann lohnt sich also diese Verwaltung? Genau dann, wenn

```
8*n > 4*CELLTABISZE + 46 + 4 + 8*(trunc(n/anzblk)+1)
```

mit

```
anzblk = max (BLOCKSIZE / sizeof(struct myrecord),1)
```

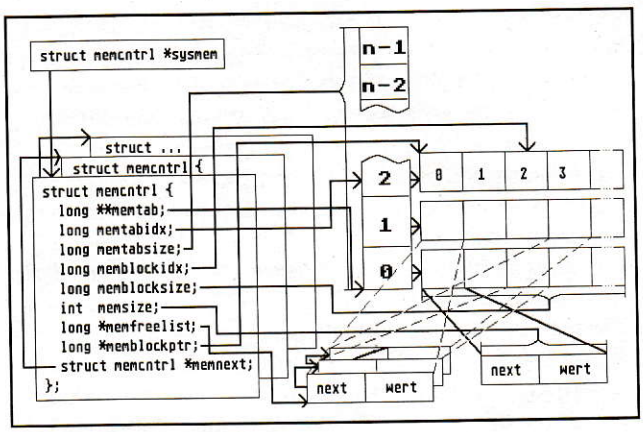
:=> mit

```
sizeof(struct myrecord) < BLOCKSIZE
8*(n - trunc((n * sizeof (struct myrecord))/
  BLOCKSIZE)) >4* CELLTABISZE + 50 + 8
```

CELLTABSIZE kann aufgrund der normalerweise maximalen Ausbaustufe von 4MB für den Atari maximal 4MB/8kB = 512 Bytes sein.

```
:=>
n - trunc((n * sizeof (struct myrecord))/
    BLOCKSIZE) > 256 + 50/8 + 1
```

macht ungefähr



Komplette Speicherverwaltung

Größe der Datenbereiche	> n
4-22	263
24-54	264
56-84	265
86-...	256

Tabelle 1 für BLOCKSIZE = 8kB ohne Berücksichtigung des zusätzlichen Programmcodes

Größe der Datenbereiche	> n
4-10	263
12-26	264
28-42	265
44-56	256
58-...	267

Tabelle 2 für BLOCKSIZE = 4kB ohne Berücksichtigung des zusätzlichen Programmcodes

Größe der Datenbereiche	> n
4-18	327
20-42	328
44-68	329
70-...	330

Tabelle 3 für BLOCKSIZE = 8kB mit Berücksichtigung des zusätzlichen Programmcodes

```
n - n * sizeof(struct myrecord)/BLOCKSIZE > 256
 + 50/8 + 1 = 263.25

n*(1 - sizeof(struct myrecord)/BLOCKSIZE) > 263.25

n > 263.25 / (1 - sizeof (struct myrecord)/ BLOCKSIZE)
```

In den Tabellen 1 und 2 kann man für verschiedene Größen eigener Datenstrukturen sehen, ab wann sich diese Speicherverwaltung lohnt, differenziert nach Größe der jeweiligen Blöcke. Wie man sieht, lohnt es sich bereits schon nach wenigen Datensätzen gleicher Größe.

Eigentlich müssen wir bei unseren Überlegungen auch noch den Speicherverbrauch der programmierten Routinen

berücksichtigen. Diese schlagen unter TURBO-C mit ca. 512 Bytes zu Buche. Daraus ergibt sich ein Korrekturfaktor von 512/8 = 64 für die rechte Seite der Ungleichung. (Daten hierzu finden sich in Tabelle 3.)

- Ein anderer Aspekt ist die Geschwindigkeit des ablaufenden Programms. Diese hat sich in meinen Fall mehr als verdoppelt. Im Durchschnitt besteht die Allokation eines neuen Records aus dem zumeist nicht benötigten Zugriff auf die entsprechende Verwaltungsstruktur und einem Array-Zugriff.

Die Adresse wird einfach durch Shift-Operationen und einfache Addition des Indexes ermittelt. Die Zeit für einen Unterprogrammaufruf und die Abfrage der Phase kommen hinzu. Insgesamt ist dieses Verfahren wesentlich schneller als das Durchsuchen einer Freispeicherliste mittels eines womöglich dem Problem nicht adäquaten Verfahrens (First/Best-Fit ...). Hinzu kommt, daß bestimmte Verwaltungsalgorithmen die freigegebenen Speicher recht stiefmütterlich behandeln.

Abschlußbemerkung

Wie man wieder mal sieht, kommt es bei der effizienten Programmierung von Problemen weniger auf eine hochoptimierte Befehlsfolge und „geniale" Registerallokation oder eingestreuten ASM-Anweisungen als auf einen intelligenteren Algorithmus an. Die hier vorgestellten Routinen sollten leicht in Programme eingebunden werden können. Anwenderschnittstellen sind nur die Unterprogramme *freshmem()*, *searchmemcntrl()* und *freemem()*. Der Programmcode hält sich in Grenzen und ist zudem noch portabel. Für alle Programme, die mit einer größeren Anzahl von gleichen Datenstrukturen kämpfen, ist dies im Vergleich zu den Systemroutinen eine bessere Lösung. Wer allerdings extrem optimieren möchte, kann dieses Verfahren mit dem in [1] vorgestellten kombinieren (oder sogar mittels Assembler-Anweisungen implementieren).

Literatur:

[1] Hans-Jürgen Richstein,
Memory Manager - Leistungsfähige Speicherverwaltung in Turbo-C,
ST-Computer 1990, Nr. 11
[2] A. Esser: TOS Intern,
ST-Computer Sonderheft Nr. 2
[3] Sharam Hekmatpour,
Lisp - A portable implementation. Prentice Hall 1989

Neue form_do-Routine

Die form_do-Routine des AES ist beim Arbeiten mit Dialogboxen ein praktisches Hilfsmittel, um die Kommunikation des ATARI ST mit dem Benutzer zu erleichtern. *form_do()* übernimmt die komplette Verwaltung der Objekte innerhalb einer Dialogbox. Was aber, wenn man während des Dialogs zusätzliche Aktionen (z.B. eine eigene Tastaturabfrage) durchführen will?

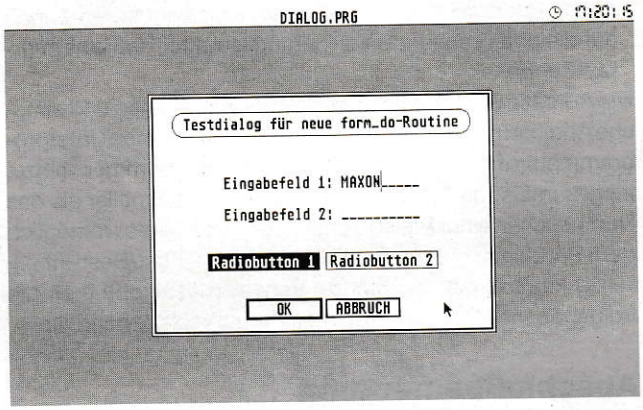
Uwe SeimetLutz Preßler

Oder wenn man das Aussehen von "Standardobjekten" verändern will? Hier kommt man mit der eingebauten *form_do*-Routine nicht weiter, sondern muß sich eine eigene Eingabe programmieren.

Bei der Programmierung von Dialogen nimmt uns das GEM mit seiner *form_do*-Routine einen Großteil der Arbeit ab. Ist die Dialogbox einmal per Resource Construction Set erstellt, wird nach dem Zeichnen dieses Objekts [mit Hilfe von *objc_draw()*] einfach die *form_do*-Routine des AES aufgerufen, und der Dialog mit dem Benutzer läuft automatisch ab. Das Hauptprogramm erhält erst wieder die Kontrolle, wenn die Dialogbox über ein EXIT- oder TOUCHEXIT-Objekt verlassen worden ist.

Vielleicht haben Sie sich aber schon die Frage gestellt, wie es manche Programme (z.B. das DISKUS-Diskutility oder der TEMPUS-Text-Editor) realisieren, die Buttons innerhalb einer Dialogbox auch über die Tastatur zu erreichen. Dies ist nur über eine neue Eingaberoutine möglich, die die vorhandene *form_do*-Routine ersetzt und neue Funktionen ermöglicht.

Was muß eine solche Eingaberoutine leisten? Zunächst müssen Eingaben oder Änderungen, die über die Tastatur oder die Maus hervorgerufen werden, auf dem Bildschirm dargestellt werden. Wird also eine Taste gedrückt, muß das der Taste entsprechende alphanumerische Zeichen in das aktuelle Eingabefeld der Dialogbox übertragen werden. Anschließend muß der Eingabe-Cursor auf die nächste Zeichenposition gesetzt werden. Handelt es sich bei der gedrückten Taste um die Tabulator- oder eine Cursor-Taste, muß ein Wechsel der Eingabeposition oder des -feldes erfolgen. Weiterhin muß sich der Programmierer darum kümmern, daß der Cursor innerhalb von Eingabefeldern an der richtigen Stelle dargestellt wird.



Schließlich müssen auch Mausaktionen überwacht werden. Wird ein EXIT- oder TOUCHEXIT-Button angeklickt, muß der Dialog beendet werden. Handelt es sich um einen Radio-Button, muß dies insofern berücksichtigt werden, als nicht nur der Status des angeklickten Buttons geändert werden muß, sondern auch andere Radio-Buttons einen neuen Objektstatus erhalten müssen.

Sie sehen, es gibt viel zu beachten. Glücklicherweise besitzt das AES genügend Unterroutinen, um bei der Verwendung einer eigenen Dialogroutine die Überwachung der oben aufgeführten Eingaben zu erleichtern. Insbesondere spreche ich damit die Funktionen *form_keybd()*, *form_button()* und *objc_edit()* an. In Verbindung mit diesen AES-Routinen läßt sich eine Eingaberoutine formulieren, die einige neue Möglichkeiten zur Verwaltung von Dialogen bietet und vom äußerem Erscheinungsbild her wie eine "normale" Dialogbox wirkt.

Damit die folgenden Ausführungen nicht graue Theorie bleiben, soll anhand des Beispielprogamms DIALOG aufgezeigt werden, wie die Programmierung einer eigenen *form_do*-Routine vonstatten gehen kann. DIALOG stellt eine Dialogbox auf dem Bildschirm dar, die aus zwei Eingabefeldern, zwei Radio-Buttons sowie einem "OK"- und einem "ABBRUCH"-Button besteht. Die Buttons können wie üblich über die Maus bedient werden, und auch sonst bestehen auf den ersten Blick keine Unterschiede zu anderen Dialogboxen.

Wie sieht es auf den zweiten Blick aus? Nun, Radio-Button 1 kann nicht nur über die Maus, sondern auch über die Tastenkombination [ALTERNATE] [1] betätigt werden, Radio-Button 2 ist über [ALTERNATE][2] ansprechbar. Darüber hinaus kann der "ABBRUCH"-Button über die [Undo]-Taste erreicht werden.

Das ausführlich kommentierte Assembler-Listing zu DIALOG.PRG möchte ich nun als Grundlage nehmen, um die Programmierung einer eigenen Dialogroutine zu erläutern. Wie bereits erwähnt, bilden die AES-Routinen *form_keybd()* und *form_button()* das Kernstück des Programms. *form_-*

FEST - UND WECHSELPLATTEN

C:\

24 ms, 800 Kb/s

NEU 88 MB

WECHSELPLATTE FÜR ATARI ST, ATARI MEGA STE, ATARI TT, APPLE, NEXT, IBM, KOMPLETT INCL. KABEL UND MEDIUM, AUCH IM MEGA–ST GEHÄUSE

AB 1598,– DM

TEST ST–COMPUTER 9/91
"IN ALLEN ANSPRÜCHEN PROFESSIONELL"

KOMBISTATIONEN:
88 MB WECHSEL + FESTPLATTE

MHDS–88–105 WECHSELPL. + QUANTUM 105 MB	2498,–
MHDS–88–210 WECHSELPL. + IMPRIMIS 210MB	3198,–
MHDS–88–545 WECHSELPL. MIT CONNER 3,5" 12 MS, 1300 KB/S SUPER!	4698,–

KOMPL. ANSCHLUSSFERTIG INCL. KABELN UND MEDIUM, AUCH MIT 44 MB WECHSELPLATTE LIEFERBAR! mit ICD-Hostadapter

A:\ MD 88

88 MB Wechselplatte Syquest

D:\

Lieferprogramm auch im Mega-ST-Gehäuse

WEITERE KOMBISTATIONEN:
44 MB SYQUEST–WECHSELPLATTE + FESTPLATTE

MHDS–44–52 44 MB + QUANTUM 52 MB,1200KB/S, 17ms	1948,–
MHDS–44–105 44MB + QUANTUM 105 MB,1200KB/S, 17ms	2245,–
MHDS–44–80 44 MB + SEAGATE ST 296N, 84 MB,28ms	1945,–
MHDS–44–81 44 MB + SEAGATE ST 1096N, 83 MB,22ms	2045,–
MHDS–44–210 44 MB + IMPRIMIS ST 1239N, 210MB,15ms	2848,–
MHDS–44–545 (auch im 35"–Format!) 44 MB + CONNER (GANZ NEU),545MB,12 ms	4498,–

WIR LIEFERN AUCHFESTPLATTEN BIS 2 GIGABYTE!!

Mit ICD Advantage/(Plus) Hostadapter, DMA in/out gepuffert, SCSI-Bus herausgeführt, SCSI-Adresse einstellbar (0—7), Max. 256 Partitionen möglich, Läuft unter allen ATARI-Betriebssystemen von TOS 1.0 bis 3.06, Auch mit UNIX, MS-DOS, Spectre, OS—9 uvm., Vollkommen ATARI AHDI 4,xx-kompatibel, Autoboot von allen Partitionen — Autopark

ATARI TT/ST/Mega ST a.A.
ATARI LASERDRUCKER SLM 605 2098,–

Nur mit den bewährten ICD-Hostadaptern haben Sie 1. Qualität. Vertrauen Sie auf diese Technik.

Irrtümer vorbehalten

E:\

Lieferprogramm auch im Mega-ST-Gehäuse

FESTPLATTEN:

MHD 52:	52 MB QUANTUM, 17 MS	AB	898,–
MHD 105:	105MB QUANTUM, 17 MS	AB	1198,–
64 KB CACHE, 1200 KB/S, SUPERLEISE			
MHD–50:	52 MB, Seagate		845,–
MHD–80:	84 MB,SEAGATE ST296N		998,–
MHD–81:	83 MB, ST1096N,24MS		1048,–
IMPRIMIS UND CONNER,–FESTPLATTEN 12MS,1200KB/S			
MHD–210:	210 MB, ST 1239N, 12MS		1798,–
MHD–545:	545 MB, CONNER, 12MS		3498,–
MHD 240:(Quantum LPS 240 MB, 12 MS)			1998,–

WECHSELPLATTEN:

MHDS–44:	SYQUEST 44 MB 28 MS	AB	1145,–

INCL MEDIUM
ALLE GERÄTE KOMPLETT ANSCHL.FERTIG AUCH IM KLEINEN TT–FESTPLATTENGEHÄUSE LIEFERBAR

SCANNER

Trade iT SCAN 256 MIT EBV–SOFTWARE	SONDERPREIS
Trade iT COLORSCAN 300: A4–SCANNER MIT GAMMA–KORREKTUR, 16.7 MIO FARBEN, 256 GS, 300DPI	a.A
HP–DESKJET 500 INCL PATRONE	a.A

WEITERES LIEFERPROGRAMM:

ST–4 FLOPPY–STATION 1.44 MB TEAC 235–PS2	KOMPL.	198,–
ST–7 FLOPPY 5,25", TEAC FD 55GFR		249,–
44 MB MEDIUM EXTRA		155,–
88 MB MEDIUM EXTRA		259,–
Wir haben noch viel mehr:		a.A

Weiteres auf Anfrage

M. Fischer Computer Systeme
Goethestr. 7 6101 Fr. Crumbach
Tel. 06164 - 4601
Fax 06164 - 3748

keybd() ermöglicht es, in einer Dialogbox Tastatureingaben zu simulieren, mit *form_button()* kann die Betätigung eines Mausknopfes simuliert werden. Damit man die Möglichkeit hat, während des eigentlichen Dialogs zusätzliche Aktionen vorzunehmen, geschieht die Abfrage von Tastatur und Maus nun nicht mehr über die eingebaute *form_do*-Routine des AES, die ja Manipulationen während des Dialogs nicht zuläßt, sondern über die *evnt_multi*-Routine. Allgemein können über *evnt_multi()* diverse Benutzeraktionen überwacht werden. Um zu spezifizieren, worauf *evnt_ multi()* konkret achten soll, wird ein Parameterwort übergeben, in dem je nach zu überwachender Aktion entsprechende Bits gesetzt sind. Nach diesem AES-Aufruf überwacht GEM für uns so lange Tastatur und Maus, bis eine Aktion (Tastendruck oder Betätigung der Maustaste) seitens des Benutzers erfolgt. Erst dann wird die Programmkontrolle wieder an das Hauptprogramm zurückgegeben. Ist dies geschehen, so finden sich in einem eigens für den *evnt_ multi*-Aufruf errichteten Array (in unserem Fall *ev_buff*) Angaben über die Art der Aktion. Wie bereits beim Aufruf von *evnt_multi()* werden diese Daten wieder durch einzelne Bits repräsentiert.

DIALOG.PRG verzweigt nun, je nachdem, um welche Aktion es sich gehandelt hat, in eine Routine zur Behandlung von Mausaktionen oder kümmert sich um die Taste, die vom Anwender gedrückt wurde. Beschäftigen wir uns zunächst mit dem Fall, daß *evnt_multi()* einen Tastendruck gemeldet hat. ASCII- und Scancode der gedrückten Taste werden uns im *ev_buff*-Array mitgeteilt. Liefert *evnt_multi()* einen gültigen ASCII-Code zurück, handelt es sich um eine "normale" alphanumerische Eingabe, und das Zeichen kann direkt in die Dialogbox geschrieben werden. Ist der ASCII-Code jedoch Null, liegt also nur der Scancode der betätigten Taste vor, geht es nun darum, festzustellen, ob ein Button existiert, der durch diese Taste bedient werden kann.

Um Buttons über die Tastatur bedienen zu können, werden bei DIALOG.PRG Tastenkombinationen zusammen mit der [ALTERNATE]-Taste verwendet, da in diesem Fall von *evnt_multi()* nur der Scancode der Taste, jedoch kein ASCII-Code zurückgeliefert wird. Natürlich muß man an geeigneter Stelle eine Information unterbringen, welcher Button durch welche Taste bedient werden kann. Es ist sehr praktisch, diese Zusatzinformation innnerhalb der Objektdaten der Dialogbox unterzubringen. Wie ein Objekt organisiert ist, läßt sich in Tabelle 1 sehen [1]. Im Wort für den Objekttyp ist nur das Low-Byte belegt. Somit steht das High-Byte für eigene Anwendungen zur Verfügung. DIALOG.PRG macht sich dies zunutze, indem es bei Buttons, die über die Tastatur bedient werden können, im High-Byte des Objekttyps den Scancode der Taste erwartet, durch die der betroffene Button angesprochen werden kann.) Das Programm durchsucht nun einfach den gesamten Objektbaum auf ein Objekt, bei dem als erweiterter Objekttyp der Scancode der Taste eingetragen ist, die gedrückt wurde. Findet sich ein solches Objekt nicht, wird zum Schleifenanfang zurückgekehrt und der Dialog fortgesetzt. Ist ein Objekt vorhanden, das über die gedrückte Taste angewählt werden kann, wird die Objektnummer berechnet und über *evnt_button()* ein Mausklick auf dieses Objekt simuliert. Der Effekt eines Tastendrucks in Verbindung mit der [ALTERNATE]-Taste ist somit der gleiche wie beim Anklicken eines Buttons mit der Maus.

Wurde ein alphanumerisches Zeichen (also ein Tastendruck ohne Kombination mit der [ALTERNATE]-Taste) eingegeben, wird mit den ASCII- und Scancodes für dieses Zeichen zunächt die *form_keybd*-Routine aufgerufen, die in der Lage ist, diesen Tastendruck für unsere Dialogbox aufzuarbeiten. Handelte es sich bei der gedrückten Taste um eine Cursor- oder die Tabulatortaste, liefert uns *form_keybd()* die Nummer des Eingabefeldes zurück, in der wir nun den Text-Cursor positionieren müssen. Zusätzlich bekommen wir in diesem Fall eine Null als Rückgabewert, was bedeutet, daß keine Taste betätigt wurde, um deren ASCII-Darstellung wir uns kümmern müßten.

Erhalten wir keine Null zurück, sondern einen positiven Wert, so ist das Eingabefeld zwar das gleiche geblieben, aber dafür müssen wir an der aktuellen Cursor-Position ein ASCII-Zeichen eintragen. Dies geschieht mit Hilfe von *objc_edit()*. Diese Funktion erlaubt es, Texteingaben in einem Formular vorzunehmen. Nach Aufruf dieser Routine mit dem ASCII-Code der gedrückten Taste wird das Zeichen in der Dialogbox dargestellt. Anschließend liefert uns *objc_edit()* die neue Position des Text-Cursors zurück, der ja jetzt um eine Stelle nach rechts gerückt ist.

Falls *evnt_multi()* abgebrochen wurde, weil ein Maus-Button betätigt wurde, ist zunächst zu prüfen, ob sich der Maus-Cursor überhaupt innerhalb der Dialogbox befindet. Hierzu können wir die *objc_find*-Funktion heranziehen. Die Koordinaten des Mauszeigers, die *objc_find()* übergeben werden müssen, werden uns von *evnt_multi($* in *ev_buff* zur Verfügung gestellt. Liefert *objc_ find()* einen Wert größer als Null zurück, befand sich der Mauszeiger innerhalb der Dialogbox. Andernfalls wird als Fehlersignal ein Glockenton ausgegeben, so wie es auch die "normale" *form_do*-Routine macht.

Wurde ein Objekt innerhalb der Dialogbox angeklickt, wird AES dieser Umstand mittels *form_button()* mitgeteilt. Das GEM übernimmt den Rest der Arbeit, kümmert sich also z.B. darum, ob es sich beim angeklickten Objekt um einen Radio-Button handelt. In diesem Fall wird nicht nur der selektierte Button invertiert, sondern die restlichen Radio-Buttons werden deselektiert. Handelte es sich beim selektierten Objekt um einen Exit-Button, liefert *form_button()* einen Wert von Null zurück. Ist dies der Fall, wird der Dialog beendet.

Wurde ein Objekt ausgewählt, das keinen Exit-Status besitzt, muß zunächst geprüft werden, ob es überhaupt selektiert werden darf und es sich nicht z.B. um einen String handelt. Stellt das angeklickte Objekt ein Eingabefeld dar, muß der Text-Cursor auf dieses Feld plaziert werden. Hier hilft die *objc_ edit*-Routine weiter. Zunächst wird der Text-Cursor an der alten Stelle entfernt und anschließend an der neuen Eingabeposition dargestellt.

Nun noch zur Darstellung eigener Objekttypen, wie im Fall von DIALOG dem String, der von einer abgerundeten Box umgeben ist. Will man die Zeichenroutinen des AES umgehen und das Aussehen von Objekten weitgehend selber bestimmen, muß man mit benutzerdefinierten Objekten (G_USERDEF bzw. G_ PROGDEF) arbeiten. Bei Objekten dieser Art zeigt *ob_ spec* (s.o.) auf eine USER BLK-Struktur (s. Tabelle 2). Trifft das AES beim Erstellen eines Dialogs auf diesen Objekttyp, wird die Objektdarstellung nicht vom AES übernommen, sondern stattdessen eine Routine aufgerufen, deren Adresse vom Programmierer in der obigen Objekt-

struktur festgelegt werden kann. DIALOG.PRG nutzt nun VDI-Routinen (nur solche dürfen beim Zeichnen eigener Objekte aufgerufen werden!) dazu, Text und Rahmen des ersten Strings der Dialogbox in einem eigenen Format, also mit abgerundeten Ecken, darzustellen. Natürlich sind an dieser Stelle die verschiedensten Manipulationen denkbar.

Eine Frage ist jedoch noch offen geblieben: Wie bringe ich im High-Byte des Objekttyps eigene Informationen unter? Hier kommt es auf das verwendete Resource Construction Set an. Die Methode, den Objekttyp für eigene Daten zu nutzen, wird z.B. durch das Resource Construction Set von Kuma unterstützt. Hier hat der Anwender die Möglichkeit, für jedes Objekt einen sogenannten "erweiterten Objekttyp" (extended object type) anzugeben. Die Zahl, die man an dieser Stelle einträgt, findet sich im High-Byte des Objekttyps wieder.

Soweit meine Erläuterungen zur Programmierung einer eigenen form_do-Routine. Weitere Informationen können dem kommentierten Programm-Listing entnommen werden. Der Einbau der vorgestellten Routine in eigene Programme gestaltet sich recht einfach, da im Grunde genommen nur der Aufruf der *form_do*-Routine des AES durch die neue Routine ersetzt werden muß.

```
typedef struct
{
  int        ob_next;      /* -> nächstes Objekt */
  int        ob_head;      /* -> erstes Kind */
  int        ob_tail;      /* -> letztes Kind */
  unsigned int ob_type;    /* Objekttyp */
  unsigned int ob_flags;   /* Objektflags */
  unsigned int ob_state;   /* Objektstatus */
  char       *ob_spec;     /* Zeiger auf ergänz. Struktur */
  int        ob_x;         /* x-Position (rel. zum
                              Parent-Objekt) */
  int        ob_y;         /* y-Position (rel. zum
                              Parent-Objekt) */
  int        ob_width;     /* Breite */
  int        ob_heigt;     /* Höhe */
} OBJECT;
```

Tabelle 1: Organisation eines Objektes

```
typedef struct
{
  int  (*ub_code) ();      /* Zeiger auf die eigene Funktion */
  long ub_parm;            /* ein optionaler Parameter */
} USERBLK;
```

Tabelle 2: Die USERBLK-Struktur

Bewegliche Dialogboxen

Wäre es nicht eine tolle Sache, wenn man sich seine Dialogboxen so auf dem Bildschirm anordnen könnte, daß man optimal damit arbeiten kann? Keine Hin- und Herfahrerei mit der Maus mehr! Daneben gibt es aber noch andere, ungeahnte Möglichkeiten. Denn oft wird auch ein gerade wichtiger Desktop-Teil von einer unvermittelt auftauchenden Dialogbox überdeckt, die auch noch unverschämterweise nach gerade den verdeckten Informationen fragt. Dann kann man wieder auf *ABBRUCH* drücken (wenn's geht), nachsehen, was darunter steht, dieses auswendig lernen oder aufschreiben und daraufhin die ganze Prozedur von neuem starten.

Matthias Baldauf

Mit der hier vorgestellten *MOVEDIAL-Library* können Dialogboxen erstellt werden, die frei auf dem Bildschirm verschiebbar sind oder wahlweise an der aktuellen Mausposition erscheinen (auf neuhochdeutsch Pop-Up-Menüs genannt). Für Programmierer werden einfach zu handhabende Dialogbox-Routinen bereitgestellt, die komplett den Aufruf, die Darstellung und Verwaltung folgender drei Dialogboxtypen übernehmen:

- **Standard-Dialogboxen:**
  diese erscheinen wie gewohnt in der Bildschirmmitte und sind nicht verschiebbar.

- **Movedial-Boxen:**
  erscheinen beim ersten Aufruf ebenfalls in der Bildschirmmitte, können dann aber vom Anwender beliebig auf dem Desktop verschoben („gedragged") werden. Bei einem erneuten Aufruf erscheinen sie dann wieder an der letzten Position.

- **Pop-Up-Dialogboxen:**
  diese tauchen immer dort auf, wo sich gerade der Mauszeiger befindet.

Die *MOVEDIAL-Library* kann von interessierten und fachkundigen Anwendern noch erweitert werden, worauf am Ende des Artikels noch spezieller eingegangen wird.

Theorie

Objekte sind die Grundelemente des ganzen AES-Systems. Die Drop-Down-Menüs, Icons, Fenster, Alert-Boxen und auch die hier behandelten Dialogboxen bestehen aus nichts anderem als Objekten. Diese sind nichts anderes als Datenstrukturen, die alle grafischen Objekte beschreiben. Alle Objekte sind in einem Objektbaum zusammengefaßt. In diesem Baum gibt es Nachbarn (*ob_next*) und Kinder (*ob_head* und *ob_tail*). Kinder können wieder Kinder haben, diese wieder Kinder usw. Nachbarn in dem Objektbaum sind z.B. die verschiedenen Dialogboxen eines Programms. Kinder innerhalb einer Dialogbox stellen die Texte, Eingabefelder und Buttons dar, die selbst wieder Objekte sind. Damit man sich das alles ein wenig besser vorstellen kann, soll es an einem Beispiel verdeutlicht werden. In Bild 1 ist eine Dialogbox zu sehen, in der zwei Kästen mit Buttons enthalten sind (abgesehen vom *OK*-Button). Die beiden Kästen stellen Nachbarn (innerhalb der Dialogbox) dar, mit den Auswahl-

Aufruf:	do_dial(dialbox)
Funktion:	führt Standard-Dialog durch
Parameter:	dialbox: Zeiger auf Dialogbaum
Rückgabewert:	angewähltes Objekt
Aufruf:	do_movedial(dialbox, dragger)
Funktion:	führt Dialog mit beweglicher Dialogbox durch
Parameter:	dialbox: Zeiger auf Dialogbaum dragger: Nummer des Dragger-Objekts
Rückgabewert:	angewähltes Objekt
Aufruf:	do_popup(dialbox)
Funktion:	führt Pop-Up-Dialog (Box erscheint an Mausposition) durch
Parameter:	dialbox: Zeiger auf Dialogbaum
Rückgabewert:	angewähltes Objekt
Als Ergänzung innerhalb der MOVEDIAL-Library enthalten:	
Aufruf:	gem_init()
Funktion:	GEM-Initialisierung
Parameter:	keine
Rückgabewert:	VDI-Handle [Rückgabewert von *graf_handle()*]
Aufruf:	gem_exit(handle)
Funktion:	GEM-Abmeldung
Parameter:	handle: VDI-handle [Rückgabewert von *gem_init()*]
Rückgabewert:	keiner →

knöpfen als Kinder. Das Schöne an der Sache ist nun, daß durch Angabe der Position (*ob_x* und *ob_y*) eines der Nachbar-Objekte innerhalb des Vater-Objekts (gemeint ist das übergeordnete Objekt, hier der Dialogbox-Rahmen) verschoben werden kann und gleichzeitig alle Kinder-Objekte (hier die Buttons) mit verschoben werden. Das liegt daran, daß alle Positionsangaben relative Positionen zum Vater-Objekt darstellen. Bei Verschiebung des Vater-Objekts wird logischerweise das Kind-Objekt mit verschoben.

Genau dieser Umstand (die relativen Positionsangaben) läßt sich nun zum Verschieben der Dialogbox nutzen. Normalerweise wird mit einem *form_center()*-Aufruf vor dem Zeichnen des Dialogs eine Zentrierung der Dialogbox auf dem Desktop vorgenommen. Dieser *form_center()*-Aufruf berechnet aus der Desktop- und der Dialogboxgröße ein Koordinatenpaar (*ob_x* und *ob_y*) und trägt dies in die Objektstruktur ein. Unterläßt man das Zentrieren, wird der Objektbaum (also die Dialogbox) an der in der Objektstruktur angegebenen Position (*ob_x* und *ob_y*) relativ zur Desktop-Position ausgegeben. Legt man in jeder Dialogbox einen speziellen *Drag*-Button an (ähnlich der Move-Leiste bei Fenstern, die dem Verschieben dient), kann man nach Druck auf diesen Button (linke Maustaste festhalten!) den Dialog verschieben. Dazu müssen Sie den Drag-Button als *EXIT*-Button deklarieren, worauf nach dessen Anwahl der Dialog verlassen wird. Jetzt kann die Wegdifferenz, die mit gedrücktem Mausknopf zurückgelegt wurde (die Wegdifferenz der Maus wohlgemerkt), erfaßt und zur alten Objektposition addiert werden. Dadurch setzen Sie die Dialogbox auf die neue gewünschte Position. Da sich dieser Vorgang recht kompliziert anhört, soll er durch Bild 2 bildhaft dargestellt werden.

Nach diesen Vorbemerkungen sollen nun die Besonderheiten der einzelnen Dialogboxarten der *MOVEDIAL-Library* erläutert werden. Wie der Leser sicher schon geargwöhnt hat, bauen die einzelnen Dialogboxroutinen auf einem gemeinsamen Unterprogrammstamm auf. Diese Routinen dienen zum Vorbereiten und Zeichnen der Dialogbox sowie zum Restaurieren des Bildschirms und benützen ausschließlich GEM-Routinen. Das ist der einzig richtige Weg, um auflösungsunabhängige Programme zu entwickeln, die dann auf jeder vom GEM unterstützten Grafikkarte lauffähig sind. Die genannten Routinen sind im einzelnen weiter unten beschrieben.

Standard

Als erstes soll der *Standard-Dialog* erwähnt werden. Eigentlich gibt es hierzu nichts mehr Besonderes zu sagen, da weiter oben schon alles Wichtige über die hier verwandte *form_center()*-Prozedur geschrieben wurde. Durch diese Routine wird die Dialogbox auf dem Desktop mittig zentriert.

Movedial

Über die *Movedial-Boxen* gibt es dagegen einiges mehr zu sagen. Nachdem nach obengenanntem Verfahren (Bild 2) der Verschiebungsvektor (Differenzwerte in x- und y-Richtung) ermittelt ist, kann die Dialogbox an der alten Position gelöscht und an der neuen wieder ausgegeben werden.

Interne Routinen der MOVEDIAL-Library [werden von den drei Dialogroutinen *do_dial()*, *do_movedial()* und *do_popup()* verwendet]:

Aufruf: vor_dial(dialbox, dialtype)
Funktion: Vorbereitungen zur Dialogboxdarstellung:
- Position der Box berechnen (mittig oder an Mausposition)
- Koordinaten in Objektstruktur eintragen
- Bildschirmhintergrund mittels *form_dial()* reservieren

Parameter: dialbox: Zeiger auf Dialogbaum
dialtype: Dialogtyp:
0: Standard-Dialogbox
1: Movedial-Boxen
2: Pop-Up-Dialogbox
Rückgabewert: keiner

Aufruf: draw_dial(dialbox, dialtype)
Funktion: zeichnet Dialogbox mittels *objc_draw()*; vorher unbedingt *vor_dial()* aufrufen!
Parameter: dialbox: Zeiger auf Dialogbaum
dialtype: Dialogtyp [siehe *vor_dial()*]
Rückgabewert: keiner

Aufruf: movedial(dialbox, dialtype)
Funktion: verschiebt Dialogbox auf dem Desktop
Parameter: dialbox: Zeiger auf Dialogbaum
dialtype: Dialogtyp [siehe *vor_dial()*]
Rückgabewert: keiner

Aufruf: nach_dial(dialbox, dialtype)
Funktion: gibt Hintergrund mittels *form_dial()* frei
Parameter: dialbox: Zeiger auf Dialogbaum
dialtype: Dialogtyp [siehe *vor_dial()*]
Rückgabewert: keiner

Warum aber müssen die Differenzwerte ermittelt werden und nicht eine absolute Position? Nun, als Grundlage für die neue Position der Dialogbox wird die aktuelle Mausposition nach Loslassen der linken Maustaste genommen. Würde die Dialogbox nun an diese Position gesetzt werden, wäre die linke obere Ecke genau unter dem Mauszeiger. Das war aber nicht beabsichtigt. Man hatte die Dialogbox ja irgendwo innerhalb des Drag-Buttons geschnappt und verschoben; dann soll sie gefälligst auch wieder so gezeichnet werden, daß der *Drag*-Button genauso unter der Maus zum Liegen kommt wie vor der Verschiebung. Genau das erreicht man durch Addition des Verschiebungsvektors auf die alte Dialogboxposition (siehe Bild 2). Eine Bereichsüberschreitung, also ein Verlassen des Desktop-Bereichs beim Verschieben der Dialogbox, ist nicht möglich, da die den Verschieberahmen zeichnende AES-Betriebssystemroutine *graf_dragbox()* die Maximalkoordinaten mit übergeben bekommt.

Pop-Up

Die Position einer *Pop-Up-Dialogbox* wird aus der aktuellen Mausposition berechnet, d.h. eine solche Box erscheint

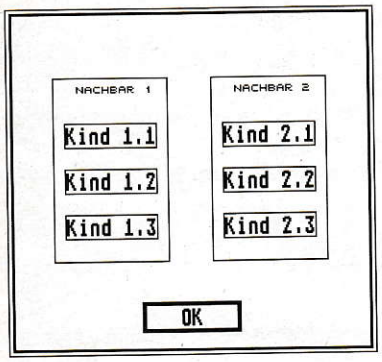


Bild 1

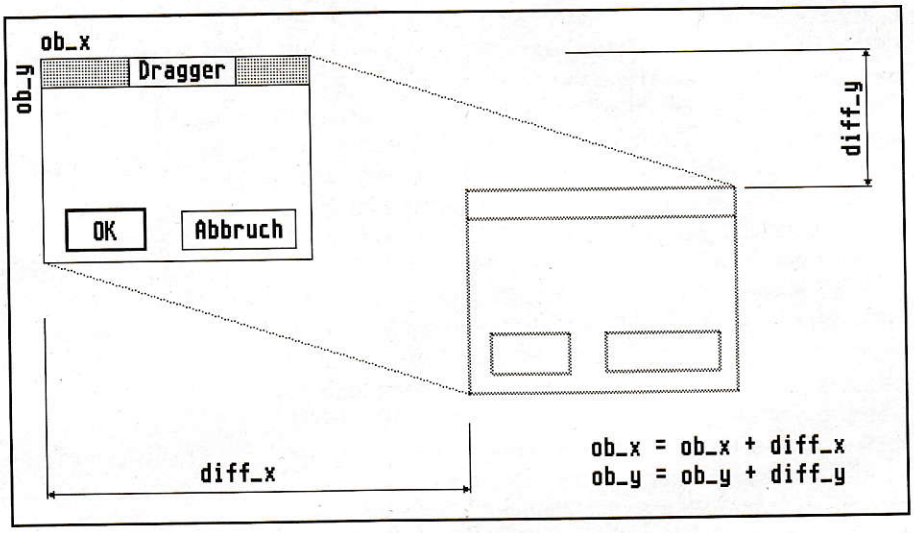


Bild 2

immer zentriert um die momentane Mausposition. Dabei gilt es natürlich noch die Fälle einer hier möglichen Bereichsüberschreitung abzufangen, denn die Dialogbox soll ja immer innerhalb des Desktops auftauchen, damit alle Elemente auch mit der Maus erreichbar sind.

MOVEDIAL-Library

Nach den theoretischen Grundlagen sollen die einzelnen Routinen der *MOVEDIAL-Library* erläutert werden. In der Prozedur *vor_dial()* werden die Vorbereitungen zur Darstellung der Dialogbox getroffen. Übergeben wird ein Zeiger auf den Dialogbaum und der Dialogtyp. Es werden dabei folgenden Typen (*dialtype*) unterschieden:

1 - Movedial-Boxen
2 - Pop-Up-Dialogbox
andere - Standard-Dialogboxen

In dieser Prozedur [*vor_dial()*] werden nach den oben besprochenen Methoden die Zeichenkoordinaten der Dialogbox (*ob_x* und *ob_y*) relativ zum Desktop ermittelt. Danach kann mit dem *form_dial()*-Aufruf der notwendige Bildschirmhintergrund reserviert werden. Mit Hilfe der Unterroutine *draw_dial()* wird dann der Dialogbaum auf dem Bildschirm ausgegeben. Der Aufruf *objc_draw()* zeichnet den Objektbaum nur, verwaltet ihn aber nicht. Die Verwaltung wird in den weiter unten beschriebenen Routinen *do_dial()*, *do_movedial()* und *do_Pop-Up()* vorgenommen. Nachdem eine Dialogbox wieder verlassen wurde, muß der Bildschirmhintergrund auch wieder freigegeben werden. Das erfolgt durch die Prozedur *nach_dial()*.

Als nächstes kann die interessanteste Routine *movedial()* unter die Lupe genommen werden. Hier wird die Dialogbox (nur, wenn es sich um eine MOVEDIAL-Box handelt) von der alten auf eine vom Benutzer gewünschte neue Position verschoben. Nachdem die Maus die Kontrolle übernommen hat [*wind_update()*] und die Form einer Hand (*FLAT_HAND*) bekommen hat, wird die Größe des Desktops ermittelt [mittels *wind_get()*]. Jetzt kann die Dialogbox innerhalb der Desktop-Grenzen mit der Maus verschoben werden. Die Dialogbox wird dabei als gepunkteter Rahmen dargestellt. Zu diesem Zweck stellt das AES schon eine Routine *graf_dragbox()* zur Verfügung, die alle Bereichsüberschreitungen abfängt. Diese Routine behält solange die Kontrolle, wie die linke Maustaste gedrückt bleibt. Nach Loslassen dieser Taste wird der Dialogbaum an der ursprünglichen Stelle gelöscht. Jetzt kann die neue Position berechnet werden. An dieser neuen Position werden wieder die Vorbereitungen zum Zeichnen der Box getroffen (also Hintergrundspeicher reserviert). Hierauf wird die Maus auf den Pfeil (*ARROW*) umgeschaltet und die Mauskontrolle wieder abgegeben [*wind_update()*].

Standard-Dialog

Nachdem jetzt alle zur Dialogabwicklung notwendigen Routinen vorgestellt wurden, müssen diese nur noch in koordinierter Reihenfolge aufgerufen werden. Zur Abwicklung eines Standard-Dialogs dient die Routine *do_dial()*, die nach Beendigung eines Dialogs den Index des Objekts zurückliefert, welches zum Verlassen der Dialogbox benutzt wurde. Übergeben wird der Zeiger auf die Objektstruktur des gewünschten Objektbaums. Beim Abarbeiten von *do_dial()* wird als erstes die Routine *vor_dial()* aufgerufen, die ja bekannterweise (siehe oben) die Vorbereitungen zur Dialogboxdarstellung trifft. Jetzt kann durch *draw_dial()* der Dialogbaum gezeichnet werden. Als *dialtype* [der zweite Parameter der *..._dial()*-Aufrufe] muß immer eine Zahl ungleich 1 und 2 (hier 0) angegeben werden. Nachdem der Dialog jetzt auf dem Bildschirm steht, kann das AES die Verwaltung übernehmen [*form_do()*]. Nach Beendigung des Dialogs wird das eventuell selektierte Exit-Objekt wieder deselektiert und der Dialog vom Bildschirm entfernt [*nach_dial()*]. Genauso wie die vorgenannte Routine *do_dial()* ist auch die Prozedur für die Pop-Up-Dialoge *do_Pop-Up()* aufgebaut. Einzig der andere Dialogtyp (*dialtype=2*) wird verwandt. Etwas aufwendiger ist die für verschiebbare Dialogboxen zuständige Routine *do_movedial()* gestaltet. Nach der bekannten Vorbereitung wird solange in einer Schleife (do..while) verblieben, bis ein Exit-Objekt ungleich dem *Drag*-Balken (mit dem Index *DRAGGER*) zum Verlassen der Dialogbox angeklickt wurde. Inner-

halb der Schleife wird der Dialog gezeichnet [*draw_dial()*] und, falls das *DRAGGER*-Objekt gewählt wurde, durch Aufruf von *move_dial()* verschoben.

Die vorgestellten *do_...()*-Routinen sind so natürlich nur für Boxen mit einem Exit-Button zu gebrauchen, die nicht weiter vom Programm verwaltet werden müssen. Das bedeutet, daß sie bei Verwendung von programmverwalteten Dialogen (z.B. Hoch- und Herunterzählen eines Wertes durch zwei Buttons) noch erweitert werden müssen. Eine solche Erweiterung durch einen *do{...}while !ende*-Block ist im Beispielprogramm dargestellt.

Beispielprogramm

Im Beispielprogramm soll die Routine *do_Pop-Up_menue()* etwas genauer betrachtet werden. Da dort die Hauptdialogbox (ein Pop-Up-Dialog) des Beispielprogramms verwaltet wird, werden als erstes die Startadressen aller benutzten Dialogboxen (Objektbäume) ermittelt [durch *rsrc_gaddr()*]. Danach wird ein do..while-Konstrukt solange durchlaufen, bis der *QUIT*-Button betätigt wird. Innerhalb der Schleife werden als erstes die Vorbereitungen zum Zeichnen des Hauptdialogs getroffen. Danach wird dieser gezeichnet. Benutzerreaktionen werden wie bekannt durch die AES-Routine *form_do()* abgewickelt, die dann den Index des Objekts zurückliefert, das zum Verlassen des Dialogs geführt hat. Jetzt kann der Dialog wieder vom Bildschirm entfernt werden. Wurde ein Objekt ungleich dem *QUIT*-Button betätigt, so werden die entsprechenden Dialogboxen aufgerufen (innerhalb des *switch()*-Konstrukts). Hier können selbstverständlich auch eigene Unterprogramme aufgerufen werden. Würde man aber auf diese Weise das oben angeschnittene Problem des in-/dekrementieren eines Wertes über Pfeilbuttons erledigen, wäre dies mehr als unelegant, da jedesmal die komplette Box neu gezeichnet werden müßte.

Erweiterungsmöglichkeiten

Zum Abschluß möchte ich auch noch die Programmierer unter den Lesern zum Verfeinern der vorgestellten Library anregen. Unter dem Motto „schöner, größer, besser, mehr" läßt sich noch einiges an den Routinen machen.

Als erstes ist hierbei an eine bessere Restaurierung des Hintergrunds zu denken. Zwar wird nach Beendigung des Dialogs die Dialogbox wieder vom Bildschirm entfernt, aber leider bleiben Reste in Fenstern zurück. Diese Rückstände bleiben solange erhalten, bis die Fensterrestaurierungsroutine des Programms aufgerufen werden kann. Ein solcher Aufruf erfolgt normalerweise innerhalb einer *event_multi()*-Prozedur. Wird also eine Dialogbox verlassen, bekommt das AES eine *REDRAW-Message*, was das eigene Programm dazu veranlaßt, die entsprechenden Fenster zu restaurieren. Dies funktioniert so natürlich nicht bei den *MOVEDIAL*-Boxen, da ja auch nach dem Verschieben innerhalb der *do_movedial()*-Prozedur verblieben wird. Wird also eine *MOVEDIAL*-Box auf dem Schirm verschoben, bleiben innerhalb von Fenstern immer Reste zurück. Dem kann man abhelfen (und dies ist der erste Erweiterungsvorschlag), wenn man vor dem Zeichnen [durch *draw_dial()*] den dadurch belegten Bilduntergrund retten würde (in einen Pufferspeicher) und ihn nach Aufruf von *nach_dial()* wieder dorthin zurückkopieren würde. Die benötigten Ausmaße werden ja auch zum Reservieren des Hintergrunds in *vor_dial()* und *nach_dial()* ermittelt. Bleibt nur noch, auf die Größe des Pufferspeichers hinzuweisen. Die benötigte Größe sollte jeweils aus der aktuellen Auflösung berechnet werden, da bei späteren Rechnerversionen (z.B. TT etc.) die Dialogboxen eventuell größer als die heutige Standardauflösung von 640x400 werden könnten (siehe dazu Artikel in diesem Heft).

Eine schöne Sache wäre auch das Verschieben der kompletten Box statt nur eines Rahmens. Dazu kann die vorgenannte Routine benutzt werden. Nur muß hier auf eine Veränderung der Mausposition gewartet und jedesmal der Bildschirm restauriert und die Box neu gezeichnet werden. Damit das Zeichnen schneller vonstatten geht, wird die Box beim Verschieben innerhalb des Bildschirms kopiert, also nicht durch *draw_dial()* neu gezeichnet. Um das ganze flimmerfrei erledigen zu können, ist folgender Ablauf denkbar:

A Bildschirmhintergrund komplett retten
B Dialogbox ein erstes Mal zeichnen [durch *draw_dial()*]
C beim Verschieben zuerst die Box an die neue Position kopieren und nur die benötigten Teile des Originaluntergrunds wiederherstellen (also die Differenz zwischen der neuen und der alten Position)

Leider kann diese flimmerfreie Methode schon einiges an Speicherplatz für den Bildpuffer kosten (man denke nur an Großmonitore mit 1024x768 oder mehr Pixeln Auflösung).

Ein letzter Tip betrifft das Wiederherstellen einer einmal gemachten Positionierung von MOVEDIAL-Boxen auf dem Bildschirm. Da nach jedem Neustart eines Programms mit MOVEDIAL-Boxen diese insgesamt neu positioniert werden müssen, wäre eine Methode vorteilhaft, die alle Dialogbox-Positionen wieder herstellen kann. Dazu müßten auf Wunsch alle Positionen in einer Datei abgelegt und bei Programmstart wieder eingelesen werden können (ähnlich *DESKTOP.INF*). Dabei muß allerdings jeweils noch eine Auflösungsanpassung vorgenommen werden. Wenn man sich vorstellt, daß beim Abspeichern der Positionen ein Monitor mit einer Auflösung von 1024x768 Pixeln verwandt wurde und einige Boxen am unteren Bildschirmrand positioniert waren, wird klar, daß ein Wiederherstellen dieser Dialogboxpositionen bei einer 640x400-Pixel-Auflösung schiefgehen muß. Man könnte die Auflösung, die beim Abspeichern eingestellt war, mit in die Info-Datei ablegen, um so die Dialogboxpositionen auf die aktuelle Auflösung bei erneutem Programmstart rückrechnen zu können.

Universelle Dialogbox

Warum viele Dialogboxen für verschiedene Zwecke konstruieren, wenn eine einzige denselben Zweck erfüllt? Edit-Felder variabler Länge machen es möglich. Dabei benötigt man weniger Speicherplatz und muß sich mit weniger Feldnamen und Adressen belasten.

Aufruf:	var_edit(tree, index, n, pt, tx, val)	
Funktion:	richtet Edit-Felder und Boxtitel ein	
Parameter:	tree:	Zeiger auf (Dialogbox-)Objekt
	index:	Index des zu verändernden Objekts
	n:	Länge des Textes bzw. des Edit-Feldes
	pt:	Zeiger aus Maskentext
	tx:	Zeiger auf Text
	val:	Zeiger auf Validitätszeichen 9: nur Ziffern X: alle Einträge erlaubt
Rückgabewert:	Adresse des Textes	

Aufruf:	hndl_dial(tree, cur, x, y, w, h)	
Funktion:	Dialogboxbehandlung	
Parameter:	tree:	Zeiger auf (Dialogbox-)Objekt
	cur:	Objektindex des Edit-Feldes, auf dem der Cursor erscheinen soll
	x, y, w, h:	x- und y-Koordinate, Breite und Höhe der Dialogbox
Rückgabewert:	angeklicktes Objekt	

Wolfgang Heine

Stellen Sie sich ein Programm vor, in dem Sie dem Benutzer die Möglichkeit bieten wollen, über die Breite und Art der Edit-Felder einer Dialogbox frei entscheiden zu können (z.B. Adimens Init). Denken Sie an ein Programm, in dem der Benutzer in unterschiedlichen Dialogboxen verschiedene Einträge ganz bestimmter Art und Länge machen soll; oder wollen Sie ein Programm erstellen, das Edit-Felder mit mehr als 40 Buchstaben benötigt, dann kann Ihnen die hier vorgestellte Routine einige Anregungen bieten.

Wir wollen das Problem durch Konstruktion eines einzigen Objektbaums lösen, den man durch Modifikation während des Programmlaufs den jeweiligen Bedürfnissen anpaßt.

Vorarbeiten

Zum Studium der Objekt- und TEDINFO-Struktur verweise ich auf die in ST Computer bereits erschienenen Artikel sowie auf das Handbuch Ihrer Programmiersprache.

Für eine variable Dialogbox benötigen Sie eine Anzahl von Zeichenketten (im Beispielprogramm 14 mit einer Länge von 80 Zeichen). Für diese müssen Sie Speicher reservieren bzw. eine Struktur schaffen. Des weiteren werden in der TEDINFO-Struktur in den letzten beiden Einträgen (Länge der Textzeichenkette und des Maskentextes) Platzhalter eingesetzt (hier -2), die im späteren Programm jeweils auf die aktuellen Werte gesetzt werden müssen.

Natürlich benötigen wir auch eine Objektstruktur für die Dialogbox. Hierin sind die Breiten der Edit-Felder und somit der ganzen Box sowie die x-Koordinaten aller Objekte mit Platzhaltern zu versehen. Vergessen Sie nicht, das letzte Objekt mit LASTOB zu versehen. Bei der Steuerung des Schreib-Cursors für die Edit-Felder mittels der Pfeiltasten kommt es sonst zu einem Totalabsturz.

Zur jeweiligen Darstellung einer Dialogbox müssen die Platzhalter durch aktuelle Werte ersetzt werden. Wenn ein Objekt n Buchstaben enthält, werden dazu 8*n Bildpunkte benötigt. Links und rechts sollten Sie einen Rand lassen (im Programm je 30 Punkte). Die Routine *var_edit()* wird danach zum Einrichten der variablen Felder aufgerufen. Dabei erhält Sie bei jedem Aufruf die Adresse des (Dialog-)Baums, den Objektindex des zu ändernden Eintrags, die Anzahl der Buchstaben, den Maskentext und ein Validitätszeichen ('9' bedeutet 'nur Ziffern; 'X': jeder Eintrag ist erlaubt).

Zur Bearbeitung der Dialogbox steht Ihnen die Routine *hndl_dial()* zur Verfügung. Diese zeichnet die Dialogbox zentriert und setzt den Cursor auf ein freibestimmbares Edit-Feld. Als Rückgabewert wird die Nummer des angeklickten Objekts geliefert.

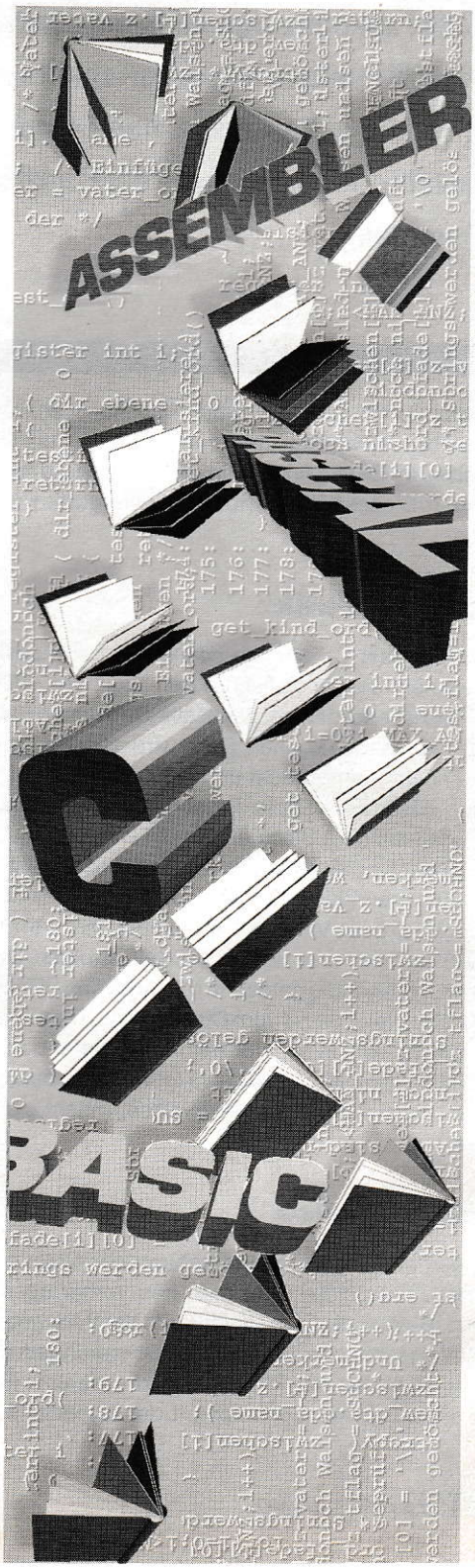
Anregung

In ähnlicher Form lassen sich alle Arten von Objekten verändern, verschieben (siehe 'Bewegliche Dialogboxen' in diesem Heft), verstecken oder mehrfach nutzen. Versuchen Sie doch, zur Übung eine Box mit einem Titel- und einem einzigen Edit-Feld zu entwerfen, in das Sie nacheinander bei jedem Aufruf etwas anderes eingeben können. Versuchen Sie es vielleicht mit einem Paßwort, Ihrer Konto- und Ihrer Telefonnummer.

Achten Sie dabei auf den Maskentext und die Anzahl der einzugebenden Zeichen sowie auf ihre Gültigkeit. Mit ein wenig Übung können Sie bald komplizierte Dialogboxen zusammensetzen.

Spracherweiterungen

Heutzutage sind höhere Programmiersprachen in den meisten Fällen in komplette Entwicklungsumgebungen eingebunden und mit einer Reihe von Zusatzfunktionen versehen. Diese dienen der Erweiterung des jeweiligen Sprachenstandards und der leichteren Programmierung des Rechners (z.B. GEM-Bibliothek). Manchmal sucht man aber gerade eine, für sein Programm benötigte Erweiterung, umsonst. Dann können Routinen oder, für einen größeren Themenkomplex, eine ganze Bibliothek sehr von nutzen sein. Man findet diese in den einzelnen Computerzeitschriften oder kann Sie als Spracherweiterungen kaufen (siehe Übersicht im Utilitie-Teil). Auf den folgenden Seiten finden Sie 8 Erweiterungen (von der Speicherverwaltung über dynamische lokale Variablen bis hin zu Mengen) für die Programmiersprachen C, PASCAL und BASIC.



Dynamische, lokale Variablen in C

In C muß der Platzbedarf von lokalen (automatischen) Variablen bereits zur Übersetzungszeit bekannt sein. In diesem Artikel wird eine einfache Funktion vorgestellt, mit der zur Laufzeit Platz für weitere lokale Variablen geschaffen wird, die genauso wie vom Compiler bereitgestellte beim Verlassen der Funktion 'zerstört' werden.

Roman Hodek

Diese Funktion nenne ich in Anlehnung an [1] *alloca()*, und sie ist wie folgt deklariert: void *alloca(unsigned int size). Im Listing finden Sie die Funktion als Assembler-Programm. Dieses können Sie beim Linken Ihres Programms anfügen.

Stack-Akrobatik

Am Anfang jeder C-Funktion befindet sich ein *LINK A6,#x*. Dieser Befehl sichert zunächst den Wert von *A6* auf den Stack. Das ist erforderlich, damit alle Funktionen das gleiche Register verwenden können. Dann wird der Stackpointer (*SP* bzw. *A7*) nach *A6* gebracht und der Wert *x* (er muß negativ sein) zum *SP* addiert. Effekt dieser komplizierten Angelegenheit ist, daß auf dem Stack ein 'Loch' entsteht. Dieses Loch wird als Platz für die lokalen Variablen benutzt. Zusätzlich hat man in *A6* einen Zeiger auf das obere Ende des Lochs, der zum Zugriff auf die Variablen verwendet wird. Am Ende der Funktion muß dann der Befehl *UNLK A6* erfolgen. Damit wird das ganze rückgängig gemacht, d.h. *A6* in den *SP* übertragen und der alte Wert von *A6* vom Stack geholt. Danach sieht der Stack wieder so aus wie vor dem LINK-Befehl. Und genau diese Tatsache, daß *UNLK* keine Information benötigt, wieviel Platz durch *LINK* reserviert wurde, macht sich *alloca()* zunutze.

Funktionsweise

Man kann sich die Vorgänge anhand der Grafiken im Bild vorstellen. Ganz links sieht man den vorher geschilderten Zustand. Beim Aufruf von *alloca()* wird der Parameter *size* auf den Stack geschoben, ebenso die Rücksprungadresse. *alloca()* holt als erstes die Rücksprungadresse vom Stack und merkt sie sich in *A0*. Jetzt kommt der eigentliche Sinn der

Aufruf:	alloca(size)
Funktion:	Reserviert auf Stack Platz für lokale Variablen
Parameter:	size: Größe des zu reservierenden Speichers
Rückgabewert:	Zeiger auf reservierten (Stack-)Speicherbereich

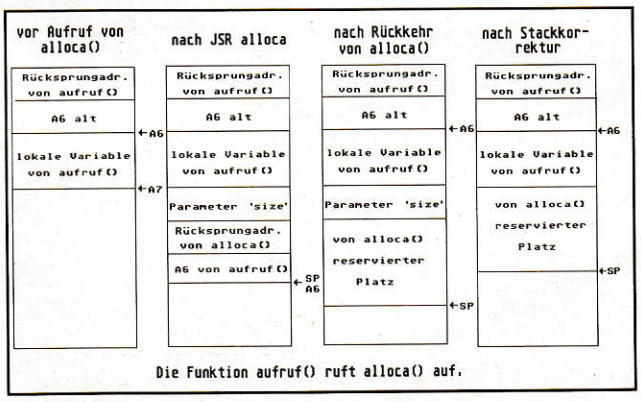
Sache: Vom *SP* wird der Wert *size* abgezogen, genauso wie es *LINK* tut. Es hat auch denselben Effekt: auf dem Stack entsteht wiederum ein Loch. Nur ist die Größe des Lochs nicht im Programm festgeschrieben, sondern wird von *size* angegeben. Damit hätten wir unseren dynamisch allozierten Speicher. Nun muß *alloca()* noch einen Zeiger auf diesen Speicher zurückgeben, und das ist *SP+2*. Dieser Wert wird in *D0* geladen und mit *JMP(A0)* zurückgesprungen. Die Rücksprungadresse haben wir ja vom Stack heruntergeholt, also ist *RTS* nicht möglich.

Warum ist eigentlich der Zeiger SP+2? Die aufrufende Funktion muß nach der Rückkehr noch eine Stack-Korrektur durchführen, d.h. den Platz für die Parameter wieder freigeben. In unserem Fall sind das 2 Bytes. Nach dem *JSR alloca* setzt der Compiler also noch *ADDQ.L #2,SP* (oder so ähnlich). Daher beginnt der allozierte Speicher nicht bei SP, sondern bei SP+2. Er wird aber dadurch nicht kürzer, da der Platz, den der Parameter (*size*) eingenommen hat, jetzt zum allozierten Speicher gehört.

Nun die schlechte Nachricht ...

So schön wie *alloca()* funktioniert, im Leben gibt es leider keine Vorteile ohne Nachteile. Und der Nachteil von *alloca()* ist, daß eine Funktion, die dynamische lokale Variablen nutzt, keine Registervariablen benutzen darf.

Die Erklärung ist einfach: Werden Register verwendet, müssen deren alte Inhalte irgendwohin gerettet werden, man will ja nicht die Registervariablen einer aufrufenden Funktion



Inhalt des Stacks beim und nach dem Aufruf von alloca()

zerstören. Der Compiler macht das, indem er die in Frage kommenden Register mit einer MOVEM-Anweisung auf den Stack schiebt und vor dem Rücksprung wieder zurückholt. Und das Ablegen geschieht unglücklicherweise genau unter den lokalen Variablen. *alloca()* würde seinen Speicher jetzt unterhalb der Register anlegen, so daß beim Funktionsende nicht die alten Registerwerte zurückgeholt würden, sondern der Inhalt des dynamischen Speichers, was natürlich Unsinn ergäbe.

Es wäre zwar eine Änderung an *alloca()* denkbar, die die geretteten Registerwerte nach unten verschieben und auf diese Weise Platz reservieren würdet. Doch leider ist die Länge des Registerbereiches nicht bekannt, nur die Länge Register + lokale Variablen (*A6-SP*). Man müßte also, wenn, die beiden zusammen verschieben, um neuen Platz auf dem Stack zu gewinnen. Bitte denken Sie selbst alle Möglichkeiten durch (ich habe mir lange genug den Kopf zerbrochen).

```
 1: ; alloca() reserviert Platz auf dem Stack
 2: ; Die aufrufende Funktion darf keine
 3: ; Registervariablen verwenden !
 4: ; (c) 1992 MAXON Computer
 5:
 6: alloca:  move.l  (a7)+,a0   ; Rücksprungadr.
 7:
 8:          move.w  (a7),d0
 9:          suba.l  d0,a7
10:
11:          move.l  a7,d0
12:          addq.l  #2,d0
13:
14:          jmp     (a0)
```

Die alloca()-Funktion

Wenn nicht eine der beiden Teillängen bekannt ist, kommt entweder Unsinn heraus, oder die vorher vorhandenen Inhalte der lokalen Variablen gehen verloren. Und das ist nun wirklich nicht der Sinn von dynamischem Speicher.

GDOS-Zeichensätze in MAXON-Pascal

MAXON-Pascal bietet durch die Units GEMVdi und GEMAes vollen Zugriff auf alle VDI- und AES-Routinen. So ist es auch kein Problem, das GDOS zu integrieren, um so eine große Hardware-Kompatibilität zu erreichen. Als kleine Anregung möchte ich zeigen, wie man GDOS-Bildschirmzeichensätze in eigene Programme einbindet.

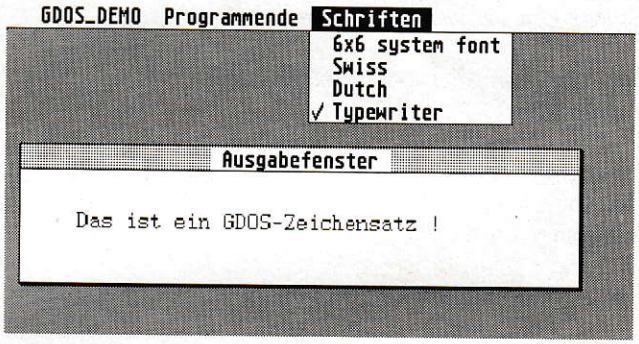


Bild 1: Die geladenen GDOS-Zeichensätze können für die Schriftausgabe im Fenster gewählt werden. Der ausgewählte Schrifttyp wird dann im Menü mit einem Haken gekennzeichnet.

Wolfgang Sattler

Das GDOS (Graphics Device Operating System) ist ein Programm, das die Verwaltung von verschiedenen Gerätetreibern (Bildschirm, Drucker, Metafile etc.) mit den dazugehörigen Zeichensätzen übernimmt. Über das GDOS ist somit eine komfortable Kommunikation zwischen eigenen Programmen und der Hardware möglich. Bei einem Wechsel der Hardware muß nur das GDOS angepaßt werden, und somit ist das eigene Programm ohne Anpassung auf einer Vielfalt von Hardware-Zusammenstellungen (z.B TT, Großbildschirm, Beschleunigungskarten etc.) lauffähig. Trotz dieser Möglichkeit hoher Kompatibilität wird das GDOS von den meisten Programmierern immer noch stiefmütterlich behandelt, obwohl es in den meisten Programmiersprachen sehr leicht über die VDI-Aufrufe genutzt werden kann. Diese Unbeliebheit folgt wahrscheinlich aus seiner geringen Geschwindigkeit. Aber durch die Entwicklung hin zu Beschleunigungskarten und Programmen (NVDI, Quick ST etc.) bzw. zu schnelleren Rechnern (TT und Mega STE) wird dieses Manko immer mehr aufgehoben. Ich möchte zur Demonstration des GDOS ein kleines Programm in MAXON-Pascal vorstellen, das die GDOS-Bildschirmzeichensätze nutzt.

Im folgenden möchte ich auf das Zusammenspiel zwischen GDOS und VDI eingehen und die von mir benutzten VDI-Routinen genau beschreiben:

Das GDOS lädt beim Starten die Datei *ASSIGN.SYS*, in der die zu ladenden Gerätetreiber und Zeichensätze aufgeführt sind. Nähere Information über Aufbau und Bedeutung dieser Datei findet man in [2]. Die in dieser Datei angegebenen Zeichensätze für den Bildschirm wollen wir nun mit unserem Programm einladen und zur Textausgabe im Fenster einsetzen. Zum Laden benutzt man die Funktion

```
anzahl:=vst_load_fonts(vdi_handle, select);
anzahl,vdi_handle,select: INTEGER;
```

die alle verfügbaren Zeichensätze in den Speicher lädt und uns die Anzahl der geladenen Zeichensätze übergibt. Man übergibt ihr als Parameter das *Vdi_handle* (die Kennung, die man beim Öffnen der virtuellen Workstation erhalten hat) und den Wert *select = 0*. Zur Auswahl zwischen den einzelnen Zeichensätzen dient uns die Prozedur

```
vst_font(vdi_handle,font);
vdi_handle,font: INTEGER;
```

wobei *font* den Index des jeweiligen Zeichensatzes darstellt. Allerdings haben wir beim Laden noch keinerlei Informationen über die Indizes oder die Namen unserer geladenen Fonts erhalten. Hier hilft die Funktion *vqt_name*. Dieser Routine übergibt man als Parameter *elementnum*, als wievielter der Zeichensatz geladen wurde (hierbei zählt der Systemzeichensatz als Nummer 1) und erhält Name und Index zurück.

```
index:= vqt_name(vdi_handle, elementnum,name);

index,vdi_handle,elementnum: INTEGER;
name: STRING80;
```

Jetzt kann man die Zeichensätze leicht mittels *vst_font* auswählen. Am Programmende werden die Zeichensätze mittels *vst_unload_fonts* aus dem Speicher gelöscht:

```
vst_unload_fonts(vdi_handle, select);
```

In *select* übergeben wir wieder den Wert 0. Weitere Informationen über diese Routinen findet man in [1] und [2]. Im englischsprachigen Handbuch sind die Parameter leider teilweise falsch beschrieben.

VDI-Aufrufe, die das GDOS benötigen, wie z.B. *vst_load_fonts*, verabschieden sich bei nicht installiertem GDOS in der Regel mit ein paar Bomben. Leider gibt es keine VDI-Routine, die das Vorhandensein von GDOS überprüft. Es gibt aber eine von Atari beschriebene Routine, die überprüft, ob GDOS installiert ist.[1] Die Funktion liefert den Wert 0, falls GDOS nicht installiert ist. Ich habe diese Routine mit Namem *vq_gdos* in Assembler mitaufgeführt. Man muß die Routine nur mittels Inline-Anweisungen einbinden.

Menübehandlung

Um im Programm zwischen den einzelnen Zeichensätzen wählen zu können, möchte ich den Schriften Menüeinträge zuordnen. Dazu erstelle ich zuerst mit Hilfe eines Resource Construction Sets einen Menübaum mit drei Titeln: *GDOS_Demo*, *Programmende* und *Schriften*. Im Menü *Schriften* benenne ich die Box, in der sich die Einträge befinden, als *MENU_BOX*. Dann kann ich später über die Object-Struktur auf die Größe der Box zugreifen. Ich füge nun 10 Einträge mit der Länge von 18 Zeichen und dem Text *unbelegt* ein. In diese trage ich während des Programmablaufs die Zeichensatznamen mittels *menu_text* ein. (Deshalb unbedingt auf die Länge von 18 Zeichen achten). Jetzt gebe ich dem ersten Eintrag den Namen *SCHRIFT1*. Zum Schluß brauche ich nur noch dem Quit- und dem Infoeintrag einen Namen zu geben, (siehe auch Listing ZEICHEN.I) und man kann mit dem Programmieren loslegen.

Jetzt ein paar Worte zum gesamten Programmaufbau: Nach einer Überprüfung der GDOS-Installation und erfolgreicher GEM-Initialisierung laden wir die Zeichensätze und erfragen deren Namen und Index. Die Namen tragen wir ins Menü Schriften ein. Wir verstecken nun noch restliche Menüeinträge und passen die Größe der Menübox an. Der System-Font wird als aktueller Zeichensatz im Menü abgehakt dargestellt. Damit ist die Routine *Menü_Vorbereitung* abgeschlossen. Nun öffnen wir ein Ausgabefenster, in dem wir einen Text ausgeben, damit wir die Zeichensätze auch sehen können. Jetzt kann man bequem mit Hilfe des Menüs zwischen den Schrifttypen umschalten.

Hat man in aller Ruhe alle Schriften bewundert, kann man das Programm beenden. Jetzt wird das Fenster geschlossen und gelöscht. Dann löschen wir noch das Menü und die Zeichensätze aus dem Speicher und melden uns bei GEM ab.

```
1: ; Assembler-Routine zur Überprüfung der GDOS-
     Installation
2:
3: VQ_GDOS:  move.l (a7)+,a0   ;Rücksprungadresse
                                 merken
4:           move.w #-2,d0
5:           trap   #2
6:           cmp.w  #-2,d0
7:           sne    d0
8:           ext.w  d0
9:           move.l d0,(sp)
10:          jmp    (a0)       ; zurück zum
                                  Hauptprogramm
```

Routine zur Überprüfung der GDOS-Installation

Memory Manager

Wer ausgiebig Gebrauch von den Lieblingskindern der Informatik - Datenstrukturen wie Records und daraus entstehenden Listen, Bäumen, Schlangen usw. - macht, ist auf die dynamische Zuteilung von Speicher durch das den Speicher-Heap verwaltende GEMDOS angewiesen, muß also zur Laufzeit des Programms immer wieder anfragen, ob denn weitere Bytes für ein neues Listenelement o.ä. zu bekommen seien.

Hans-Jürgen Richstein

Ausgelegt ist diese Speichervergabe aber eher für einige größere Blöcke, wie sie beim Laden und Starten eines Programmes erforderlich werden. Natürlich darf man auch z.B. lediglich 12 Bytes für ein kleines Listenelement anfordern; versucht man dies jedoch mehr als etwa 300mal, so ist das Betriebssystem schon bald an den Grenzen seines internen Verwaltungsspeichers, wo es die 16 Byte großen sogenannten Memory-Deskriptoren (MDs) für die vergebenen Miniblöcke ablegt [3][4].

In diesem Fall hört man oft den gutgemeinten Rat, sich einen großen Speicherbereich vom Betriebssystem zu holen und diesen dann selbst zu verwalten. Aber wie?

Auf die Finger geschaut

Verschiedene Hochsprachen-Compiler stellen eigene Routinen zur Speicherverwaltung zur Verfügung, die das Betriebssystem insbesondere bei der Anforderung kleiner Blöcke für Strukturvariablen (Records) entlasten sollen.

Eine Untersuchung beispielsweise der von Turbo-C 2.0 zur Verfügung gestellten Routinen *malloc*, *free* usw. zeigt jedoch , daß auch hier nicht der Weisheit letzter Schluß realisiert wurde. Es handelt sich um eine Modifikation der in [1] vorgestellten Speicherverwaltung.

Angeforderte Blöcke ab 4 KByte werden direkt vom Betriebssystem geholt und bei der Freigabe auch direkt an dieses zurückgegeben. Zusätzlich werden aber auch verkettete sogenannte *Memory Cluster* mit einer Größe von 8 KByte selbst verwaltet, aus denen heraus dann Blöcke von weniger als 4 KByte Länge vergeben werden.

Kommt es nun also zu einer solchen Anforderung eines kleinen Blocks, wird zunächst geprüft, ob bereits *Memory Cluster* vorhanden sind, und wenn ja, ob in einem der Cluster noch freier Speicher der angeforderten Größe vorhanden ist. Ansonsten wird zunächst ein weiterer 8 KByte (genau 8208 Bytes, also 8 KByte plus zwei mal 8 Bytes für initiale Verwaltungseinträge) großer Speicherblock vom Betriebssystem angefordert und in die Cluster-Verkettung eingefügt. Dann wird Speicher in der gewünschten Größe von dem Cluster abgeteilt und mit einem 8 Byte großen Datenkopf versehen,

Aufruf: init_memory_manager(start_adress, size)
Funktion: initialisiert interne Verwaltungsstrukturen
Parameter: start_adress: Zeiger auf den verwaltenden Speicherbereich
size: Größe des zu verwaltenden Speicherbereichs
Rückgabewert: 0: alles ok; -1: Fehler

Aufruf: mm_malloc(desired_size)
Funktion: gibt Speicherbereich (aus zu verwaltendem) zur Benutzung frei
Parameter: desired_size: gewünschte Speichergröße
Rückgabewert: void-Zeiger auf reservierten Speicherbereich; NULL im Fehlerfall

Aufruf: mm_calloc(number_of_items, size_of_items)
Funktion: reserviert Speicherbereich [wie malloc()]; als Parameter wird jedoch nicht die gesamt benötigte Größe, sondern die Anzahl der Elemente und die Größe eines einzelnen Elemets übergeben
Parameter: number_of_items: Anzahl der Feldelemente
size_of_items: (Byte-)Größe eines Elements
Rückgabewert: void-Zeiger auf reservierten Speicherbereich; NULL im Fehlerfall

Aufruf: mm_realloc(block, new_size)
Funktion: nachträgliche Änderung der Größe eines angeforderten Speicherblocks. Wird ein neuer Block benötigt, kopiert die Routine die Werte aus dem alten Block in den neuen
Parameter: block: Zeiger auf Speicherblock [lieferte mm_malloc() oder mm_calloc()]
new_size: neue Größe des Speicherblocks
Rückgabewert: Zeiger auf neuen Speicherblock; NULL, wenn Größenänderung nicht möglich

der zur späteren Identifizierung bei der Rückgabe dient. Wurden alle Blöcke aus einem Cluster wieder freigegeben, wird dieser wieder komplett an das GEMDOS zurückgegeben.

Letztendlich stößt man natürlich auch hier irgendwann auf die o.g. Grenzen, denn die Cluster müssen ja ebenfalls aus dem Hintergrund vom Betriebssystem herangeschafft werden. Das Problem ist allerdings nicht sehr gravierend, denn durch die Pufferung des Speichers in den Clustern lassen sich immerhin etwa 2.5 MByte Speicher auch für beliebig viele kleine Speicheranforderungen verwalten.

Selbsthilfe

Trotzdem gab es durch eine konkrete Problemstellung Gründe, eine eigene Speicherverwaltung auszutüfteln. Ein Programm erforderte die Aufteilung des zur Verfügung stehenden Speichers in zwei Teile: zum einen in einen Puffer für Dateien, der möglichst groß sein sollte, und zum anderen einen Verwaltungsspeicher, der dementsprechend auf ein Mindestmaß beschränkt bleiben mußte. Der erstgenannte Puffer mußte eine unveränderliche Größe haben, so daß der verbleibende Speicher alle dynamisch erzeugten Datenstrukturen von sehr unterschiedlicher Größe (16 Bytes bis 32 KBytes) aufzunehmen hatte.

Läßt man dem GEMDOS nun aber z.B. 100 KBytes übrig, um diese dann über klassische (C-) *mallocs* anzufordern und so als Verwaltungsspeicher zu nutzen, wird dieser Bereich nicht optimal genutzt. Einige hundert Speicheranforderungen für sehr kleine Datenstrukturen und diverse benötigte Blöcke mit Größen von 1 bis 32 KByte fragmentieren den Speicher recht schnell, da beispielsweise der Übergang von einem zum nächsten Speicher-Cluster nicht genutzt werden kann, selbst wenn diese im Speicher direkt nebeneinander liegen. So können am Ende der Cluster freie Bereiche übrig bleiben, die wegen einer geringfügig größeren Anforderung vorerst nicht genutzt werden können. Stattdessen wird ein neuer Cluster vom Betriebssystem geholt. Weiterhin kann freier Speicher am Anfang und Ende eines solchen Clusters nicht mit dahinter- oder davorliegendem GEMDOS-Freispeicher zu einem größeren Block verschmelzen, solange er wegen geringfügiger Belegung unter der Fuchtel der zusätzlichen Speicherverwaltung steht.

Ungünstig wird die Speicherauslastung dieses Verwaltungsbereiches auch von der Vielzahl sehr kleiner Datenstrukturen beeinflußt, da für jede Zuteilung neben der angeforderten Menge auch noch 8 Bytes für die Verwaltung anfallen, die ebenfalls in dem Cluster als *Header* vor dem vergebenen Block abgelegt werden. So wächst der effektive Speicherbedarf der zahlreichen kleineren Strukturvariablen mit Längen von 16-32 Bytes um 25-50% über die eigentliche Datenmenge hinaus an.

Zur Sache

Diese Problematik taucht im Prinzip immer dann auf, wenn man nicht in Megabytes schwelgen kann, sondern den Speicherbedarf minimieren möchte. Dies ist zum Beispiel bei Accessories nötig, die ja mit so wenig Speicher wie möglich auskommen sollen, zumindest während sie inaktiv sind.

Aufruf:	mm_free(block)
Funktion:	gibt reservierten Speicher (nach Plausibilitätsprüfung) wieder frei
Parameter:	block: Zeiger auf Speicherblock
Rückgabewert:	0: alles ok; -1: Fehler
Aufruf:	mm_coreleft()
Funktion:	Ermittlung der Größe des größten noch freien Speicherbereichs
Parameter:	keine
Rückgabewert:	Größe des Speicherbereichs
Aufruf:	mm_total_coreleft()
Funktion:	Ermittlung des gesamten noch nutzbaren Netto-Speichers
Parameter:	keine
Rückgabewert:	gesamt vorhandener freier Speicher
Aufruf:	mm_number_of_fragmenst()
Funktion:	ermittelt Anzahl der Bereiche, in die der freie Speicher [siehe mm_total_coreleft()] unterteilt ist
Parameter:	keine
Rückgabewert:	Anzahl der noch freien Speicherbereiche

Die hier vorgestellte Speicherverwaltung wird einmalig zu Programmbeginn mit einem festen Speicherbereich initialisiert, aus dem sie fortan dynamisch Speicherblöcke vergeben kann. Nach außen hin geschieht dies mit denselben Funktionen, wie sie auch in C verwandt werden. Lediglich ihre Aufrufsyntax ist etwas verändert, um sie unterscheiden zu können.

Der zu verwaltende Speicherbereich kann über *Malloc* anfangs geholt werden, kann aber auch ein im Programmcode reserviertes Feld sein. Dies ist insbesondere bei Accessories wichtig, die Daten über ihre aktive Phase hinaus behalten sollen. Mittels *Malloc* allozierter Speicher wird bei Accessories ja bei Terminierung des Elternprozesses mit freigegeben [3].

Die freien Speicherblöcke erhalten jeweils eine *MEM_BLOCK*-Struktur (siehe Abb. 1) als Datenkopf, der die Information über die Größe des Blocks enthält, und einen Zeiger auf den im Speicher folgenden Block.

Die Hauptverbesserung gegenüber den in [1] und [2] vorgeschlagenen Speicherverwaltungen ist die Beschränkung auf ein Minimum an Verwaltungsdaten für die vergebenen Blökke. Ganz ohne eine Information über die Größe des vergebenen Blockes kommt man natürlich nicht aus, denn bei der Freigabe kann man ja aus der Startadresse noch nicht auf die Länge schließen.

Bei der o.g. Problemstellung waren ausschließlich Blöcke von weniger als 64 KByte zu vergeben, was wohl auch sonst der weitaus häufigste Fall zur Laufzeit eines Programms sein dürfte. Für diese Blöcke reicht es aber aus, ihre Größe in einem vorzeichenlosen Integer - also in nur 2 Byte - zu speichern. Lediglich für Blöcke mit einer Größe von mehr als 64 KByte wird ein *long* erforderlich. Nun kann man aus der Anfangsadresse eines freigegebenen Blocks ebenfalls nicht

ermitteln, ob der Block mit einem *long* oder einem *int* als Datenkopf versehen ist. Deswegen muß man dem größeren Blocks insgesamt 6 Bytes Datenkopf gönnen (Abb. 1).

Bei Rückgabe eines Blocks schaut man sich zunächst die zwei Bytes vor dem übergebenen Zeiger an. Steht dort eine Null, handelt es sich um einen großen Block, dessen Größe nun weitere vier Bytes vorher ausgelesen werden kann. In den anderen Fällen sind die zwei Bytes vor dem Zeiger die als vorzeichenlose Ganzzahl abgelegte Größe. Diese zwei verschiedenen *Header* sind in den Datenstrukturen *LARGE_BLOCK* und *SMALL_BLOCK* (siehe Abb. 1) ausgedrückt.

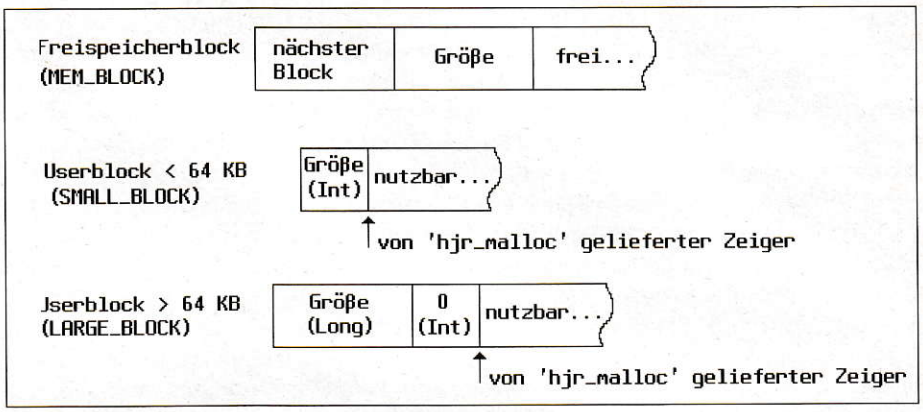


Abb. 1: Die Datenstrukturen

Ansonsten ist keine weitere Verwaltung der Blöcke in irgendwelchen zusätzlichen Listen erforderlich. Dadurch vergrößert sich der Anteil des effektiv nutzbaren Freispeichers ganz erheblich; insbesondere bei kleinen Anforderungen wird weit weniger Speicher vergeudet.

Und so geht's

Die vorgestellten Routinen werden genauso benutzt, wie die Speicherverwaltung des Betriebssystems oder die der C-Compiler. Zusätzlich muß allerdings am Anfang eines Programms der zu verwaltende Speicherbereich initialisiert werden. Diesen Bereich kann man auf beliebige Weise beschaffen, sei es durch ein zur Compile-Zeit erzeugtes statisches Array oder durch Anforderung beim GEMDOS.

Ein Grundgerüst für C-Programme, das dieses Vorgehen demonstriert, zeigt das Listing. *MEM_MNGE.H* enthält die Prototypen für die global zugänglichen Funktionen und ist per *#include* einzubinden. Die eigentliche Speicherverwaltung wird lediglich übersetzt, also ohne Projektdatei compiliert. Das entstehende *MEM_MNGE.O* muß dann zu der jeweiligen Applikation dazugelinkt werden, wobei automatisch nur die tatsächlich verwendeten Routinen eingebunden werden.

Die folgende Erläuterung der relevanten Routinen erfolgt mit formlosen Parametern. Die jeweilige genaue Definition entnehmen Sie bitte jeweils dem Listing.

init_memory_manager (Startadresse,Größe)
Durch den anfänglichen Aufruf dieser Funktion werden die internen Verwaltungsstrukturen initialisiert. Man übergibt lediglich die Startadresse und die Größe des zu verwaltenden Bereichs.

Im wesentlichen wird dem Block eine initiale *MEM_BLOCK*-Struktur als Datenkopf verpaßt und ein zusätzlicher Dummy-*MEM_BLOCK* angelegt, der einen Null-Speicherbereich bezeichnet und als Listenkopf für die Freispeicherverkettung dient und nie vergeben wird. Weiterhin werden Beginn und Ende des Bereichs in Variablen abgelegt, so daß später zurückgegebene Blöcke auf ihre Lage innerhalb dieses Bereichs geprüft werden können. Wenn alles geklappt hat, gibt es zur Belohnung eine 0 zurück, ansonsten eine -1.

hjr_malloc(Größe)
Es wird ein Speicherbereich der gewünschten Größe vom Freispeicher abgeteilt. Falls dies erfolgreich möglich war, erhält man einen typenlosen Zeiger auf den zugewiesenen Bereich, ansonsten einen Null-Zeiger. Dieser Bereich darf auf jeden Fall nur bis zu der angeforderten Größe benutzt werden. Schreibt man über diesen Bereich hinaus, zerstört man wichtige Verwaltungsinformationen des dahinterliegenden Blocks. Die Reaktionen darauf reichen von Bomben (dahinter lag Freispeicher) bis zu rätselhaftem Speicherschwund (dahinter lag vergebener Speicher, der nun nicht mehr freigegeben werden kann).

Zunächst wird die tatsächlich erforderliche Speichergröße ermittelt, d.h. zu der gewünschten Größe addieren sich je nach Erfordernis zwei oder sechs Bytes für den *SMALL_BLOCK*- bzw. *LARGE_BLOCK*-Kopf. Danach wird die Freispeicherverkettung nach dem ersten Block durchsucht, der größer oder gleich der benötigten Größe ist. Dieses ist das sogenannte *First-Fit*-Verfahren. Man könnte auch den Block suchen, der die am besten passende Größe hat, also nach dem *Best-Fit*-Algorithmus. Tatsächlich ergeben sich aber in der Praxis keine Vorteile, die den größeren Suchaufwand rechtfertigen würden. Wie Versuche gezeigt haben, ist eine bessere Auslastung des Speichers, wie man sie intuitiv vermuten würde, nicht gegeben [2].

War die Suche nach einem Block erfolgreich, wird dieser auf die erforderliche Größe gestutzt, sofern der dann verbleibende Speicher groß genug ist, um eine *MEM_BLOCK*-Struktur aufzunehmen. Diese ist ja für die Verwaltung des freien Speichers in jedem Block erforderlich. Reicht der Restspeicher nicht dafür aus, wird er einfach mit vergeben, so daß der Anwender dann einen geringfügig größeren Block erhält, ansonsten wird der zu große Block in zwei Blöcke aufgeteilt, von denen der erste genau die gewünschte Größe hat. Diese Aufgaben übernimmt die Funktion *split-block*.

Schließlich wird der so gefundene und eventuell verkleinerte Speicherblock für die Vergabe vorbereitet, d.h. mit dem richtigen Datenkopf versehen. Die Funktion *make_user_block* übernimmt diesen Teil der Arbeit. Sie liefert dann auch gleich den typenlosen Zeiger auf den zwei oder sechs Bytes weiter hinten liegenden Bereich, der dem Anwender dann zur Nutzung übergeben wird (siehe Abb. 1).

hjr_calloc(Anzahl,Größe)
Diese Funktion stellt lediglich eine Erweiterung der *hjr_malloc*-Routine dar, die im wesentlichen die Reservierung von Speicher für Datenfelder (Arrays) erleichtert.

Man übergibt der Funktion die Anzahl der Elemente und die Größe eines einzelnen Elementes. Es werden dann Anzahl mal Größe Bytes angefordert und - falls vorhanden - anschließend mit Null initialisiert. Der Wert der Funktion ist ebenfalls ein typenloser oder ein Nullzeiger.

hjr_realloc(Block,Größe)
Diese Funktion erlaubt es, die Größe eines bereits angeforderten Blocks nachträglich zu verändern. Sie ist im Gegensatz zu den meisten Speicherverwaltungen wesentlich hartnäckiger bei der Durchsetzung der Interessen des Anwenders. Die Funktion liefert einen neuen typenlosen Zeiger auf den Bereich zurück, der der neuen Größenanforderung genügt, oder einen Nullzeiger, falls es keine Möglichkeit gab, einen entsprechend großen Block zu organisieren. Die bisherigen Daten bleiben auf jeden Fall immer erhalten, außer natürlich bei einer Verkleinerung des Bereiches, wo der hintere Teil des Bereiches abgeschnitten wird.

Der neue Zeiger muß aber keineswegs mit dem übergebenen Bereich übereinstimmen, auch nicht bei einer Verkleinerung, wenn diese über die 64-KByte-Grenze hinweg geschieht. Vielmehr wird in der Regel ein neuer Bereich zugewiesen, in den die bisherigen Daten herüberkopiert wurden. Daher ist nach einem *hjr_realloc* darauf zu achten, daß Zugriffe auf den Bereich nur relativ zu diesem neuen Beginn gemacht werden. Bekommt man einen Nullzeiger zurück, ist der alte Bereich trotzdem noch aktiv, er ließ sich halt nur nicht wunschgemäß vergrößern. Man muß also daran denken, ihn noch freizugeben, wenn man ihn so nicht mehr brauchen kann.

Nach Übergabe des bisherigen Blockzeigers und der gewünschten neuen Gesamtgröße wird zunächst geprüft, ob der Block vergrößert oder verkleinert werden soll. Eine Verkleinerung stellt natürlich überhaupt kein Problem dar und wird sofort erledigt, wenn der gewonnene Speicher groß genug ist, um eine *MEM_BLOCK*-Struktur aufzunehmen. Die Funktion *shrink_block* präpariert dazu den frei werdenden Speicher so, als wäre er ursprünglich alloziert worden. Dann wird er mit der serienmäßigen *hjr_free*-Routine (s.u.) freigegeben. Falls der Block über die magische 64-KByte-Grenze hinweg verkleinert wird, verkleinert sich natürlich auch der Datenkopf um vier Bytes. Dann wird auch der gesamte zugewiesene Bereich um vier Bytes verschoben, so daß auch nach der Verkleinerung noch alle Daten an demselben Ort sind - allerdings nur relativ zu dem zurückgelieferten Zeiger.

Soll der Block vergrößert werden, sind einige weitere Anstrengungen erforderlich. Zunächst wird der bequemste Fall geprüft, nämlich ob hinter dem zu vergrößernden Bereich direkt ausreichend Freispeicher zur Verfügung steht. Wenn dem so ist, wird dort die benötigte Portion abgeschnitten und an den übergebenen Bereich angehängt. Wenn dadurch nicht die 64-KByte-Grenze überschritten wurde, ist alles paletti, und es wird genau der Blockzeiger zurückgeliefert, der auch übergeben wurde. Daten müssen überhaupt nicht verschoben werden.

In allen anderen Fällen kommen wir nicht so leicht davon. Das kleinste Problem tritt auf, wenn trotz der Möglichkeit zur obigen direkten Speichervergrößerung der Block nun größer als 64 KBytes wird und wir ihn nun mit einem *LARGE_BLOCK-Header* versehen müssen. Dann müssen alle bisherigen Daten des Anwenders noch um vier Bytes nach oben verschoben werden.

Reicht der Speicher hinter dem Block nicht aus, wird dasselbe noch mit dem Freispeicher versucht, der sich möglicherweise direkt vor dem übergebenen Block befindet. Ist auch hier Fehlanzeige, zeigt sich der *Memory-Manager* geduldig und versucht, ob nicht Vorgänger- und Nachfolgerspeicher gemeinsam den Speicherhunger des Anwenders befriedigen können. Nach Verschieben der Daten und Freigabe von eventuellem Restspeicher wären dann alle zufrieden.

Zeichnet sich jedoch immer noch keine glückliche Lösung ab, müssen wir den *Brute-Force*-Weg gehen und einfach

frech per *hjr_malloc* nach Speicher in der gewünschten Größe fragen. Sollte dies wenigstens erfolgreich sein, wird der alte Block komplett in diesen Bereich kopiert und dann an die Speicherverwaltung zurückgegeben. Der Anwender erhält den Zeiger auf den neuen Block.

In allen anderen Fällen müssen wir durch einen Nullzeiger Unfähigkeit zur Erfüllung der Anforderung signalisieren.

hjr_free(Block)
Mit dieser Funktion gibt man vormals allozierten Speicher zurück, so daß er anschließend zur erneuten Vergabe zur Verfügung steht.

Zunächst wird untersucht, ob es sich bei dem übergebenen Block um einen *LARGE_BLOCK* oder einen *SMALL_BLOCK* handelt. Aus dieser Information gewinnt man dann den tatsächlichen Beginn und die Größe des Blocks. Diese Daten werden dann der Funktion *insert_into_list* übergeben, wo sie zunächst im Rahmen der Möglichkeiten einer Plausibilitätskontrolle unterzogen werden. Mit dem Block stimmt sicherlich etwas nicht, wenn er mitten in einem Vorgängerbereich beginnt oder in nachfolgenden Freispeicher hineinreicht. Suspekt wäre auch, wenn die Größe kleiner als eine *MEM_BLOCK*-Struktur wäre, oder wenn die Grenzen des Blocks außerhalb des eigentlich verwalteten Bereichs lägen. In so einem Fall wird der Block zwecks Schadenbegrenzung schlicht ignoriert, was durch einen Rückgabewert von -1 angezeigt wird.

Zeigen sich keine Probleme mit den Daten, wird untersucht, ob er eventuell mit davor- und/oder dahinterliegendem Speicher zu einem größeren zusammenhängenden Block verschmelzen kann. Dies läßt sich einfacher erledigen, als vielleicht zunächst vermutet werden könnte. Eine Null gibt es für erfolgreiche Rücknahme des Blocks.

hjr_coreleft()
Diese Funktion durchsucht einfach die Freispeicherverkettung nach dem größten freien Block und teilt dem Anwender dann dessen effektive Größe mit, d.h. die reale Größe abzüglich eines erforderlichen Datenkopfs Marke *LARGE_BLOCK* oder *SMALL_BLOCK*.

hjr_total_coreleft()
Diese Funktion geht über den üblichen Standard hinaus und teilt einem den gesamten noch nutzbaren Netto-Speicher mit.

hjr_number_of_fragments()
...ist als Ergänzung zur Funktion *hjr_total_coreleft* zu sehen. Sie gibt Auskunft über die Fragmentierung des Freispeichers, d.h. in wieviele voneinander isolierte Bereiche der verwaltete Speicher zerstückelt ist. Diese beiden letzten Routinen kann man während der Programmentwicklung zu Effizienzanalysen der Speicherverwendung nutzen. Man kann sich dann stichprobenartig „x Bytes frei in y Fragmenten" o.ä. ausgeben lassen.

...zu guter Letzt

Die Speicherverwaltung ist in dieser Form sehr effizient, sowohl bezüglich der Verarbeitungsgeschwindigkeit und der Größe des einzubindenden Codes als auch der Nutzung des zur Verfügung gestellten Speicherbereichs.

Trotzdem gilt es, die angeforderten Bereiche mit größter Sorgfalt zu verwenden, denn sie teilen sich den Speicherblock mit den vitalen Verwaltungsdaten der Freispeicherverkettung. Man darf also auf keinen Fall über einen angeforderten Bereich hinaus schreiben. Dies ist aber im Prinzip kein wirklicher Nachteil, denn schließlich wäre dann die anfordernde Applikation irgendwo fehlerhaft. Während hier die Speicherverwaltung selbst darunter zu leiden hätte, wären es bei einer Verwaltung durch eine zentrale Liste eben andere Daten der Applikation, die hinter dem wuchernden Bereich abgelegt wären und nun Schaden nähmen.

Mengen in C

In C ist im Gegensatz zu Pascal oder Modula-2 kein Mengentyp vorhanden. Mengen spielen jedoch eine Rolle in der Cluster-Analyse, der kombinatorischen Optimierung und sind Grundlage der formalen Begriffsanalyse, mit der ich mich seit einigen Jahren befasse. Grund genug, eine eigene Bibliothek zum Rechnen mit Mengen in C zu schreiben.

Klaus Rindfrey

Wie können Mengen im Rechner dargestellt werden? Ein Element kann in einer Menge vorhanden sein oder nicht. Das bedeuten, daßeine Menge durch ein boolesches Array dargestellt werden kann, wobei jedem Element ein Array-Eintrag zugeordnet wird. In C bietet es sich an, die bitweisen Operatoren zu benutzen, man benötigt dann für jedes Element nur ein Bit, so daß in einem Wort 16 Elemente gespeichert werden können.

Betrachten Sie die Definition des Typs *BitSetType* in *bset_typ.h*. Das Array *bitset[]* enthält die eigentliche Menge: Ein Element ist genau dann in der Menge enthalten, wenn das entsprechende Bit gesetzt ist. Nun muß man noch wissen, wie *bitset[]* zu interpretieren ist. Dazu dient die Struktur *BitSetDefType* , auf die *bsdef* zeigt: *arrlen* gibt die Länge von *bitset[]* an, *minelem* bzw. *maxelem* das kleinste bzw. größte Element der Grundgesamtheit und *maxcard* die Größe (Kardinalität) der Grundgesamtheit. *mask* dient zum Maskieren des letzen Feldes von *bitset[]*. Das kann nötig sein, wenn die Größe der Grundgesamtheit nicht durch 16 teilbar ist, so daß das letzte Feld nicht voll genutzt wird. Diese überzähligen Bits müssen aber immer 0 sein, da sonst z. B. die Prüfung der Teilmengenrelation falsche Ergebnisse liefert.

Wie können nun die Mengenoperationen konkret in C umgesetzt werden? Bei den bitweise Operatoren handelt es sich um boolesche Operatoren, die die Bits eines Wortes (bzw. Langwortes) einzeln logisch verknüpfen. Wir müssen also die Mengenoperationen durch logische Ausdrücke darstellen und diese dann mit den C-Operatoren schreiben. Zwei Beispiele sollen das Vorgehen verdeutlichen:

1. Die Vereinigung zweier Mengen A und B ist definiert als die Menge, die alle Elemente enthält, die in A *oder* B enthalten sind. Wir benutzen also das bitweise logische *oder*, in C wäre das *A | B* .
2. Schwieriger ist die Umsetzung des Enthaltenseins (Teilmengenrelation). A ist Teilmenge von B, wenn gilt: $x \varepsilon A \Rightarrow x \varepsilon B$. Ein *wenn-dann* läßt sich mit den bitweisen Operatoren nicht direkt schreiben. Man kann sich jedoch überlegen (z. B. mit Hilfe einer Wahrheitswertetafel), daß der logische Ausdruck *a => b* äquivalent ist zu *not a or b*. In C muß also gelten *~A | B == ~0* (d. h. hier müssen alle Bits gesetzt sein), wenn $A \leq B$. In der entsprechenden C-Funktion wurde die Negation benutzt, um eine Schleife zu verlassen, d.h. *A & ~B != 0*, wenn A *nicht* Teilmenge von B ist.

Bezeichnung	Mathematische Schreibweise	Schreibweise mit C-Operatoren für int A, B, x
Vereinigungsmenge	A U B	A | B
Schnittmenge	A ∩ B	A & B
Differenzmenge	A \ B	A & ~B
Symmetrische Differenz	A Δ B	A ^ B
Komplement	C(A)	~A
Teilmenge	A ≤ B	~A | B == ~0
Element von	x ε A	(0x1 « x) & A != 0
Disjunkt	A ∩ B = Ø	A & B == 0
Kardinalität	|A|	Nicht möglich
leere Menge	A = Ø	A == 0
Gleichheit	A = B	A == B

```
 1: /*  BSET_TYP.H - Typdefinitionen für Bitsets */
 2: /*  (c) 1992 MAXON Computer                   */
 3: /*
 4: BitSetDefType - Informationen d.Mengentyps.
 5: Interne Struktur - Nicht darauf zugreifen !!!
 6:  */
 7: typedef struct {
 8:    unsigned arrlen;   /* Länge von bitset[] */
 9:    unsigned maxcard;  /* max. Kardinalität  */
10:    int      minelem;  /* kleinstes mögliches
11:                        * Element           */
12:    int      maxelem;  /* Größtes mögliches
13:                        * Element           */
14:    unsigned mask;     /* zum Maskieren d. letzten
15:                        * Feldes in bitset[].*/
16:              } BitSetDefType;
17:
18: /*
19:  * BitSetType - Datentyp für Mengen
20:  * Interne Struktur - Nicht darauf zugreifen !!!
21:  */
22: typedef struct {
23:    BitSetDefType *bsdef; /* zeigt auf Definition
24:                           * des Mengentyps */
25:    unsigned  bitset[]; /* Menge als Bit-Muster */
26:              } BitSetType;
27:
28: /*
29: BS_OpId - Code f. die möglichen Mengenoperationen
30:  */
31: typedef enum {
32:    BS_UNION, BS_INTERSECT, BS_DIFFERENCE,
      BS_INCLUDE,
33:    BS_EXCLUDE, BS_ISELEMENT, BS_ISEQUAL,
      BS_ISSUBSET,
34:    BS_CARDINAL, BS_COMPLEMENT
35:             } BS_OpId;
```

Typen- und Strukturdefinition zur Mengen-Bibliothek

In der Tabelle sind die wichtigsten Mengenoperationen und -relationen und ihre Operatordarstellung zusammengefaßt. Lediglich für die Kardinalität (Anzahl der Elemente einer Menge) läßt sich keine entsprechende Darstellung finden. Hier muß jedes einzelne Bit geprüft und dabei ggf. ein Zähler

inkrementiert werden. Kommen wir nun zu den einzelnen Bibliotheksfunktionen (*bitset.c*). Mit der Funktion *bs_newsettype()* wird eine neue Struktur vom Typ *BitSetDefType* erzeugt. Als Parameter erhält diese Funktion das kleinstmögliche und das größtmögliche Element. Dadurch ist der Gültigkeitsbereich der Menge festgelegt. Betrachtet man die Informationen in *BitSetDefType* als Typ einer Menge, so kann man sagen, daß *bs_newsettype()* zur Laufzeit einen neuen Mengentyp erzeugt. Als Rückgabewert erhält man einen Zeiger auf die Struktur oder NULL, falls kein Speicher reserviert werden konnte.

Mit *bs_createset()* wird eine neue, leere Menge erzeugt. Als Parameter übergibt man der Funktion einen Zeiger auf den gewünschten Mengentyp, der vorher mit *bs_newsettype()* erzeugt wurde.

Rückgabewert ist ein Zeiger auf die Menge (bzw. NULL im Fehlerfall). Die einzelnen Mengenoperationen wurden sinnvoll zusammengefaßt:

bs_2setop() enthält die Mengenoperationen, die zwei Mengen als Operanden haben und als Ergebnis wieder eine Menge liefern (Vereinigung, Durchschnitt, Differenz).

bs_1setop() enthält die Operationen mit einer Menge als Operand (Kardinalität, Komplement).

bs_elemop() enthält die Operationen, bei denen ein Element und eine Menge verknüpft werden (Einfügen und Entfernen eines Elements, Element-von-Relation).

bs_cmpset() enthält die Vergleichsoperationen für zwei Mengen (Teilmenge, Gleichheit).

Um eine Mengenoperation durchzuführen, haben die Funktionen einen zusätzlichen Parameter, der angibt, welche Operation gemeint ist. Der Typ des Parameters (*BS_OpId*) wurde in *bset_typ.h* als Aufzählungstyp definiert. Benutzt man die in *bset_ext.h* definierten Makros, so braucht man sich um diesen Parameter nicht weiter zu kümmern. Das Maskieren von Bits mit dem Strukturelement *mask* aus *BitSetDefType* ist nur dann erforderlich, wenn eventuell ungenutzte Bits gesetzt werden könnten. Dies ist bei dieser Version von *bitset.c* nur bei der Komplementbildung der Fall.

In *bsdemo.c* wird die Anwendung der Bibliotheksfunktionen demonstriert. Es wird zunächst ein Mengentyp *setofchar* definiert, anschließend werden drei Mengen dieses Typs erzeugt, mit denen dann einige Operationen durchgeführt werden. Außer mit *int* und *char* kann man für Elemente auch den Aufzählungstyp benutzen. Wir haben somit die gleichen Möglichkeiten wie mit *SET OF* in Pascal ohne die dortige Größenbeschränkung.

Aufruf: bs_newsettype(mine, maxe)
Funktion: Anlegen von neuem Mengengentyp
Parameter: mine: kleinstmögliches Element
maxe: größtmögliches Element
Rückgabewert: Zeiger auf neuangelegten Mengentyp

Aufruf: bs_createset(bsd)
Funktion: erzeugt neue, leere Menge vom Typ bsd
Parameter: bsd: Zeiger auf Mengentyp [wird von bs_newsettype() zurückgeliefert]
Rückgabe: Zeiger auf neue Menge

Aufruf: bs_2setop(bs1, bs2, erg, opid)
Funktion: führt Mengenoperationen mit 2 Operanden aus: bs1 <opid> bs2 -> erg
Parameter: bs1, bs2: Zeiger auf Mengen (Typen müssen gleich sein!); bs1 und bs2 werden mittels durch opid bestimmter Operation verknüpft
erg: Zeiger auf Ergebnismenge
opid: gewünschte Operation:
BS_UNIO: Vereinigungsmenge
BS_INTERSECT: Schnittmenge
BS_DIFFERENCE: Differenzmenge
Rückgabe: wenn Ausführung ok: TRUE, bei unterschiedlichen Mengentypen: FALSE

Aufruf: bs_1setop(bsp, opid)
Funktion: führt Mengenoperationen mit einem Operanden aus
Parameter: bsp: Zeiger auf Menge
opid: gewünschte Operation:
BS_CARDINAL: Kardinalität berechnen
BS_COMPLEMENT: Komplement berechnen
Rückgabe: wenn Ausführung ok: TRUE, sonst: FALSE

Aufruf: bs_elemop(e, bsp, opid)
Funktion: führt Operationen mit einem Element und einer Menge durch
Parameter: e: Element
bsp: Zeiger auf Menge
opid: gewünschte Operation:
BS_INCLUDE: Element einfügen
BS_EXCLUDE: Element entfernen
BS_ISELEMENT: prüft, ob e Element von bsp ist
Rückgabe: wenn Ausführung ok: TRUE, sonst: FALSE

Aufruf: bs_cmpset(bs1, bs2, opid)
Funktion: führt Vergleichsoperationen durch
Parameter: bs1, bs2: Zeiger auf zu vergleichende Mengen
opid: gewünschte Operation:
BS_ISSUBSET: prüft auf Teilmenge
BS_ISEQUAL: prüft auf Gleichheit
Rückgabe: wenn Vergleich ok: TRUE, sonst: FALSE

Aufruf: bs_maxcard(bs)
Funktion: liefert maximale Kardinalität
Parameter: bs: Zeiger auf Menge
Rückgabe: maximale Kardinalität

Exget

Die Idee zu der Funktion exget (extended get) kam mir beim Programmieren mit C. Alle Eingaberoutinen für die Tastatur, die in C bereits verfügbar waren, waren relativ unkomfortabel oder sogar gefährlich.

Aufruf:	exget(eing, laenge, dfstr)	
Funktion:	erweiterte get-Funktion zur Eingabe und Editierung eines Text-Strings	
Parameter:	eing:	Zeiger auf den String (beim Editieren muß der Zeiger auf den darzustellenden Text zeigen)
	laenge:	maximale Länge des einzulesenden Strings
	dfstr:	1: String wird zum Editieren ausgegeben; sonst nur Eingabe
Rückgabewert:	keiner	

Helmut Lehmkuhl

Unkomfortabel, weil beim Lesen von Daten mit *scanf* die Eingabe bei einem Whitespace (Leertaste, Tabulatortaste, Return-Taste) beendet wird. Dies macht sich insbesondere bei Eingabe von Strings, in denen ein Leerzeichen enthalten sein soll, unangenehm bemerkbar.

Gefährlich, weil man bei Eingabe mittels *gets* (get String) zwar auch Whitespace einlesen kann, diese Funktion aber nicht prüft, ob man vorher genügend Speicher reserviert hat. Reserviert man also für die Variable, mit der man *gets* aufruft, nicht genügend Speicher, kann man leicht bei der Eingabe zuviel Zeichen tippen und damit unkontrolliert Werte im Speicher zerstören, die vielleicht noch wichtig gewesen wären. Die Folge kann ein Systemabsturz sein.

Hier greift nun die Funktion *exget* ein. Mit ihrer Hilfe wird es möglich, Strings in C sehr komfortabel einzulesen. Der korrekte Aufruf lautet:

exget(&eing, laenge, dfstr, schalter)

Dabei müssen die Parameter wie folgt deklariert worden sein:

```
char *eing
int laenge
int dfstr
int schalter
```

Mit *laenge* kann man angeben, wie lang der einzulesende String maximal werden darf. Hierbei ist zu beachten, daß man den Wert um 1 größer angeben muß als die tatsächliche Länge des Strings, da für das String-Endezeichen ('\0') ja auch ein Byte benötigt wird. Der String kann dann nur bis zu dieser maximalen Länge eingegeben werden. Versucht man mehr Zeichen einzugeben, ertönt eine Glocke.

Zudem läßt sich ein Default-String auf den Bildschirm bringen, der dann ediert werden kann. Dazu muß *eing* schon auf einen String zeigen und *dfstr* muß den Wert 1 bekommen. In dem Fall wertet *exget* den Default-String aus und gibt ihn auf dem Bildschirm aus. Hat der Default-String eine kürzere Länge als man in laenge angegeben hat, so wird *laenge* auf diese kürzere Länge gesetzt, so daß sich nur ein String bis zu dieser Länge eingeben läßt. Dies dient der Verhinderung des Hineinschreibens des Strings in wichtige Speicherstellen. Schließlich zeigt *eing* dann vielleicht nicht auf einen Speicherbereich, der groß genug ist. Will man also einen Default-String edieren, der kürzer als *laenge* ist, muß man ihn mit Blanks auffüllen, bis er *laenge* erreicht hat. Default-Strings, die länger sind als in laenge angegeben, werden nur bis *laenge* ausgegeben und können nur bis zu dieser Länge ediert werden. Wird *dfstr* auf einen anderen Wert als 1 gesetzt, wird kein Default-String ausgegeben.

Über den Parameter *schalter* hat man die Möglichkeit, in der 24sten Zeile des Bildschirms eine Ausgabe zu erzeugen, mit der man die Eingabe kontrollieren kann. Dazu muß *schalter* den Wert 1 erhalten. Bei allen anderen Werten erscheint keine Statuszeile. Hat *schalter* den Wert 1, wird folgendes ausgegeben:

AP: XXX AL: XXX ML: XXX

Hierbei bedeutet AP die aktuelle Position des Cursors innerhalb des Eingabe-Strings, AL dessen aktuelle und ML dessen maximale Länge. Da diese Ausgaben bei eigenen Programmen stören können, lassen sie sich über den Parameter *schalter* ausschalten.

Der String kann während der Eingabe ediert werden, wozu man diverse Möglichkeiten zur Verfügung hat:

<Backspace>
Mit der Backspace-Taste wird das links vom Cursor stehende Zeichen gelöscht und der rechts vom Cursor stehende Text nachgezogen. Dies dürfte jedem aus der Textverarbeitung hinreichend bekannt sein.

<Delete>
Mit der Delete-Taste wird das Zeichen, das unter dem Cursor steht, gelöscht und der rechts vom Cursor stehende Text nachgezogen.

<Insert>
Mit der Insert-Taste kann man zwischen dem Überschreib- und dem Einfügemodus umschalten. Bei jedem Druck auf diese Taste wird der Modus gewechselt. Den augenblicklichen Modus kann man am Cursor erkennen. Im Überschreibmodus hat man einen ausgefüllten stehenden Cursor, während man im Einfügemodus einen ausgefüllten blinkenden hat.

<Pfeil links>
Mit dieser Taste wird der Cursor um ein Zeichen nach links bewegt.

<Pfeil rechts>
Mit dieser Taste wird der Cursor um ein Zeichen nach rechts bewegt.

<CLR/HOME> oder **<Shift Pfeil links>**
Mit diesen Tasten wird der Cursor auf den Anfang des Eingabe-Strings gesetzt.

<Shift CLR/HOME> oder **<Shift Pfeil rechts>**
Mit diesen Tasten wird der Cursor auf das Ende des Eingabe-Strings gesetzt.

<Escape>
Mit der Escape-Taste kann man die Eingabe von neuem beginnen. Das bis dahin Getippte (oder der Default-String) verschwindet vom Bildschirm, der Cursor steht am Anfang des Eingabefeldes.

Es ist noch darauf zu achten, daß die Konstanten TRUE und FALSE deklariert werden.

Erweiterter INPUT-Befehl

Wer schon einmal probiert hat, eine Formulareingabe in GFA-BASIC zu schreiben, kennt das Problem: Die INPUT-, LINE INPUT- und FORM INPUT-Befehle des GFA-BASIC lassen zwar eine komfortable Eingabe zu, aber man kann sie nur durch RETURN beenden. Wenn man die Eingabe nun z.B. mit der 'Cursor hoch'- oder 'Cursor runter'-Taste verlassen will, um ins nächste bzw. vorherige Feld zu gelangen, gibt es nur eine Möglichkeit: eine eigene Eingaberoutine muß her.

Aufruf:	@input(v$, *i$, l, C, *r)
Funktion:	erweiterte INPUT-Routine
Parameter:	v$: Prompttext; wird nur ausgegeben, kann nicht verändert werden
	i$: Eingabe-String; Inhalt wird angezeigt und kann editiert werden
	l: Länge des Eingabe-Strings
	C: Anfangsposition des Cursors
	r: Wert der Taste, die zum Abbruch geführt hat

M.G. Berberich

Dabei ergibt sich ein Problem: Wenn man einen String normal an eine Prozedur übergibt, kann man ihn nicht zurückübergeben. Wenn man ihn mit **String* übergibt, kann man in zurückgeben, aber nicht verwenden. Also muß eine Methode her, um einen String hin- und rückzuübergeben. Hier ist die Lösung des 'Hin-rück-Problems': Von dem String liegt nur die Adresse vor. Um an ihn zu gelangen, muß man ihn zuerst in einen lokalen String in ausreichender Größe kopieren. Dies erledigt die Prozedur *gruss()*.

Wie funktioniert die Sache? Die Prozedur erhält die Adresse des *Arrptr* des Strings. Zuerst werden die nötigen Hilfsvariablen als Local definiert. Als nächstes wird die Länge des 'übergebenen' Strings bestimmt. Dann werden der Hilfs-String auf die nötige Länge mit Spaces aufgefüllt und der *Varrptr* des Strings und des Hilfs-Strings bestimmt. Beim Hilfs-String geht dies mit *VARRPTR*, beim Übergabe-String muß man die Adresse aus *Arrptr* holen. Nun kann man den Inhalt des Strings byteweise kopieren. Als Resultat hat man den String im Hilfe-String und kann ihn verarbeiten. Auf diese Art und Weise kann man einen String in eine Prozedur hin- und rückübergeben.

Die Routine wurde in der oben angesprochenen Input-Routine verwendet. Sie arbeitet im Einfügemodus, was bedeutet, daß der Buchstabe, der gerade eingegeben wird, keinen anderen überschreibt, sondern an der Cursor-Position eingesetzt wird. Die Taste Backspace löscht den links neben dem Cursor liegenden Buchstaben, die Taste Delete den rechts neben dem Cursor. Die Taste Esc löscht das gesamte Eingabefeld. Mit den Tasten 'Cursor hoch', 'Cursor runter' und 'Return' bzw. 'Enter' wird die Eingabe verlassen. Der Code der Taste, die den Abbruch verursacht hat, wird zusammen mit dem String zurückgeliefert.

Installieren von STAD-Fonts

Leider lassen sich die Schriften von STAD, im Gegensatz zu GDOS-Fonts, die mit Routinenaufrufen installiert und verwendet werden können, nicht ohne eine Aufarbeitung benutzen. Dazu können Sie das Programm STAD:GEM verwenden, das eine Umsetzung vom STAD- ins GEM-Format durchführt.

Heinz Katzenmayer

Das Format muß deshalb geändert werden, weil bei STAD jeweils die 16 Bytes eines Zeichens hintereinanderliegen und danach die Daten des nächsten. GEM benötigt die Daten aber im folgenden Format: Das 1. Byte des 1. Zeichens, dann das 1. Byte des 2. Zeichens usw. Folglich ist das 257. Byte das 2. Byte des 1. Zeichens. Verstanden? Es klingt komplizierter, als es ist (siehe innerhalb des Programms). Zusätzlich zur Datenumwandlung muß ein Fontheader für den Font angelegt werden. Allerdings muß bei einem neuen Font nicht der gesamte Header geändert werden. Dies ist nur an den fünf folgenden Stellen nötig:

1. Bytes 0 und 1, Font-Identifikationsnummer (hier 2 bis 8, da 1 System-Font)
2. Bytes 4 - 35, Name des Fonts (Font-Name von STAD plus Zusatz '.STAD'
3. Bytes 76 - 79, Adresse der Font-Daten
4. Bytes 84 - 87, Adresse des nächsten Fontheaders
5. Bytes 66 und 67, Flag zu Kennung, ob Font System-Font ist

Deshalb werden nur Font-Nummer und -namen zusammen mit den Font-Daten abgespeichert. Damit ist die Aufgabe des Programms STAD:GEM.GFA erledigt. Im zweiten Programm. F_LADEN.GFA werden nun diese Daten eingelesen. Was dann noch passiert, ist ziemlich einfach. Der Fontheader wird an den oben genannten Stellen geändert und dann zusammen mit den Font-Daten an einem vorher reservierten Speicherplatz abgelegt. Es wird also eine Font-Tabelle an die nächste angehängt. In der letzten Tabelle wird bei der Adresse des nächsten Fontheaders (siehe 4.) keine neue Adresse mehr eingetragen, da ja auch keiner mehr folgt. Alles, was sonst noch im Programm F_LADEN.GFA steht, wird nur zum Testen benutzt. Um das Programm nutzen zu können, muß man es als Unterprogramm einbauen (siehe PROC-FONT.GFA). Wenn man nicht alle sieben Fonts im Programm braucht, muß man die Zählschleife i% nur entsprechend verändern.

Natürlich kann man nur bei GEM-Anwendungen diese Fonts nutzen (in GFA-BASIC beim Befehl TEXT, nicht bei PRINT).

Utilities

Einige der wichtigsten Hilfmittel für Programmierer sind Utilities; meist kurze Programme, die dem Anwender Arbeit abnehmen (z.B. ein Suchprogramm nach bestimmten Strings innerhalb von Dateien) oder die Programmentwicklung vereinfachen (Resource-Einbindung). In dieser letzten Rubrik des Sonderhefts stellen wir Ihnen 14 Utilities vor. Sie finden diese (mit Ausnahme des ACS; dies ist ein kommerzielles Programm) auf den Disketten zum Heft.

Festplatteninfo

Das in diesem Artikel vorgestellte Programm gibt Auskunft auf die Fragen, wie groß die gesamte Platte ist, wieviel davon belegt ist, wieviel noch frei ist, wieviele defekte Sektoren markiert wurden und wie groß die einzelnen Partitionen.

Uwe Seimet

Dazu verwendet das Programm die Informationen im Boot-Sektor der Festplatte und die GEMDO-Funktion *Dfree()*, die den freien Platz auf einem Massenspeichermedium berechnet. Analog zum Boot-Sektor der Diskette enthält der Boot-Sektor der Platte die vom Programm benötigten Informationen über die Größe der Platte sowie der einzelnen Partitionen. Der Boot-Sektor ist - wie bei den Disketten - der erste physikalische Sektor auf der Festplatte. Sein Aufbau ist im Listing in der Struktur BOOTBLOCK definiert (siehe auch in [1]).

Das Programm muß also den Boot-Sektor lesen, um dessen Inhalt auswerten zu können. Ein Problem besteht in der Tatsache, daß der Boot-Sektor einer Festplatte keinesfalls identisch ist mit dem ersten Sektor der ersten Partition (vom Benutzer meist als Laufwerk C angemeldet). Da man unter TOS nicht in der Lage ist, die gesamte Platte anzusprechen, muß der Programmierer einen anderen Weg, nämlich den, die Platte direkt anzusteuern, gehen.

Anzusprechen ist eine Festplatte am ATARI ST nur über den DMA-Chip. Er stellt die Hardware-Schnittstelle zu den externen Massenspeichern wie Diskettenlaufwerken und Festplatten dar und entlastet den Prozessor, indem er unabhängig von diesem arbeitet. Man erreicht den DMA über die in Tabelle 1 aufgeführten Hardware-Register (Achtung: nur im Supervisor-Modus!).

Vor dem Zugriff überprüft das Programm, ob überhaupt zusätzliche Partitionen (und damit eine Festplatte) installiert sind. Danach wird der Boot-Sektor zur Auswertung eingelesen. Die dazu notwendigen Schritte können Sie der entsprechenden Fachliteratur (z.B. [1]) entnehmen.

Das Programm geht alle Partitions-Einträge im Boot-Sektor durch. Bei Partitionen mit einem ID-Code 'GEM' handelt es sich um GEM-Partitionen. Der freie Platz auf ihnen kann mittels *Dfree()* ermittelt werden. Der freie Platz auf der kompletten Platte ergibt sich aus der Addition der freien Partitionsplätze.

$FF8604	Sector-Count-Register (SCR) Zähler für sektorgroße Datenblöcke (512 Bytes)

Nur an dieser Adresse, wenn Bit 4 im DMR gesetzt (= 1) ist.

In diese Register wird die Anzahl der zu übertragenden Sektoren geschrieben. Es kann jederzeit gelesen werden, wieviele Sektoren der DMA-Chip noch übertragen muß.

$FF8606	beim Lesen: DMA-Status-Register (DSR) beim Schreiben: DMA-Mode-Register (DMR)

DMA-Status-Register:
Die unteren drei Bits geben Auskunft über den DMA-Chip.

Bit 0	wird low (0), wenn ein Fehler aufgetreten ist.
Bit 1	zeigt an, daß das SCR auf Null heruntergezählt ist.
Bit 2	wird low, sowie das /HDRQ-Signal von der ACSI-Schnittstelle aktiv (low) wird.

DMA-Mode-Register:

Bit 1	ist direkt auf den Ausgang CA1 geführt.
Bit 2	geht auf CA2.
Bit 3	gibt an, auf welche Peripherie zugegriffen werden soll: 0: Zugriff auf Register des FDC 1: Zugriff auf den ACSI-Bus
Bit 4	bestimmt, welches Register sich an der Adresse $FF8604 befindet: 0: Controller-Access-Register 1: Sector-Counter-Register
Bit 6	0: DMA-Chip arbeitet. 1: DMA-Chip arbeitet nicht.
Bit 7	0: Datenaustausch mit ACSI-Bus 1: Datenaustausch mit FDC
Bit 8	0: DMA-Chip liest. 1: DMA-Chip schreibt.

$FF8609	DMA-Adressen-Start- und Zähler-Register, Highbyte
$FF860B	DMA-Adressen-Start- und Zähler-Register, Midbyte
$FF860D	DMA-Adressen-Start- und Zähler-Register, Lowbyte

Diese drei Register befinden sich eigentlich in der MMU-Einheit, gehören aber zum DMA-Chip. Sie sollten in der Reihenfolge Low- -> Mid- -> Highbyte gelesen oder beschrieben werden.

Tabelle 1: Register des DMA-Chips

Literatur:

[1] Claus Brod/Anton Stepper, Scheibenkleister II, MAXON Computer

Der TRAP-Trapper

Auf größeren Systemen der Datenverarbeitung finden sich Teile des Betriebssystems, die über bestimmte Ereignisse eine Art Logbuch führen. Für derartige Programme wird üblicherweise der Begriff Logger verwendet, eine Standardanwendung ist z.B. das Führen einer Fehlerliste. Als Testhilfe kann das hier von mir vorgestellte Programm dienen, das die Betriebssystemaufrufe des ATARI ST über die Trap-Befehle des 68000-Prozessors protokollieren kann.

Stephan Simson

Vor der Implementierung eines Loggers waren einige grundsätzliche Betrachtungen anzustellen, die ich vorab kurz darlegen möchte. Zunächst muß die Programmierung wegen der möglichst hohen Ausführungsgeschwindigkeit in Assembler erfolgen. Weil der Start einer solchen Anwendung meiner Meinung nach auch aus dem AUTO-Ordner heraus möglich sein sollte, entfällt - leider - eine Realisierung als Accessory. Darüber hinaus erscheint mir eine Beschränkung des Protokolls auf die drei Bereiche GEMDOS, AES und VDI als sinnvoll, da diese die mächtigsten Funktionen des Betriebssystems zur Verfügung stellen. Um die zu erzeugende Liste auch nach einem Systemabsturz mit sich anschließendem Kaltstart verfügbar zu haben, kann diese nicht im RAM, z.B. in einem Ringspeicher, geführt werden. Sie muß vielmehr dauerhaft abgelegt werden, deshalb entschied ich mich für die Ausgabe über einen am Centronics-Port anzuschließenden Drucker. Als Kennzeichnung eines Betriebssystemaufrufs genügt dafür ein Buchstabe ('A': AES, 'G': GEMDOS, 'V': VDI), kombiniert mit einer zweistelligen Zahl, die die Funktionsnummer hexadezimal beinhaltet. Eine auf diese Weise geführte Liste zieht allerdings, trotz der kryptischen Ausgabe, einen enormen Papierverbrauch nach sich und ist wohl auch in den wenigsten Fällen gewünscht. Deshalb mußte ich die Möglichkeit schaffen, den Ausdruck ein- bzw. auszuschalten. Zur Vereinfachung der Handhabung war zudem die Implementierung einer ebenfalls schaltbaren Zeitlupe notwendig. Auf die Steuerung dieser Funktionen über die Tastatur konnte ich verzichten, da die Möglichkeit besteht, die RS232-Schnittstelle als 2-Bit-PIO zu benutzen. Die Funktion des residenten Programms sollte außerdem durch weitere Aufrufe ausgesetzt bzw. wieder aktiviert werden können.

Die elementare Funktion des Loggers besteht darin, nach der Auslösung eines der beiden Traps die Kennung des entsprechenden Betriebssystemaufrufes auszudrucken. Diese Aufgabe kann dadurch gelöst werden, daß man die jeweils zugehörige Ausnahmebehandlung über eine entsprechende Routine umleitet. Geholt werden die relevanten Daten dann vom Stack (Trap 1) bzw. über ein Register (Trap 2). Wenn es doch nur so einfach wäre! Leider ist eine derartige Installation nicht von Dauer, das Betriebssystem des ST boykottiert ein Verbiegen des Traps 2 gelegentlich durch Überschreiben des Vektors mit einem Zeiger auf sich selbst, z.B. wenn man sich eine Textdatei über das Desktop anzeigen läßt. Vor den Erfolg setzten die Götter die Reise - durch den Speicher des ATARI ST. Nach Lösen mehrerer Fahrkarten bei verschiedenen Reiseveranstaltern bzw. Debuggern konnte ich die gesuchten Übeltäter finden. Verantwortlich für die Reinitialisierung des Vektors für Trap 2 sind Teile des GEM, die durch Auslösung der Line-F-Ausnahme aufgerufen werden. Diese Ausnahmebehandlung wird beim ATARI ST im allgemeinen Line-F-Emulator genannt. Sie wird vom Prozessor eingeleitet, sobald er erkennt, daß im nächsten auszuführenden Befehlswort - in diesem Zusammenhang mit Opcode bezeichnet - die vier höchsten Bits gesetzt sind. Von den Schöpfern des GEM wurde diese Eigenschaft des 68000 benutzt, um häufig verwendete Befehlsfolgen zu ersetzen, dazu gehören auch die Aufrufe der eben angesprochenen Routinen. Auf die Nachteile dieser Art von Programmierung wurde schon in mehreren Artikeln hingewiesen, für den Logger war sie aber durchaus nützlich. Die Arbeitsweise des Line-F-Emulators, beschränkt auf seinen für dieses Programm relevanten Teil, möchte ich kurz erläutern.

Der Line-F-Emulator führt zwei grundsätzlich verschiedene Funktionen aus. Ist das niedrigste Bit des Opcodes gesetzt, erfolgt die Freigabe eines Stack-Bereiches über *Unlink* (UNLK). Andernfalls werden die unteren drei Bytes des Opcodes - er muß glatt durch vier teilbar sein - als Offset in einer Tabelle von Startadressen diverser GEM-Routinen interpretiert. An der auf diese Weise festgelegten Stelle wird das Programm dann mit dem Ende der Ausnahmebehandlung fortgesetzt. Diese stellt vorher noch den Status des Prozessors wieder richtig. Außerdem legt sie die Wortadresse auf dem Stack ab, die dem Befehl folgt, der die Ausnahme einleitete. Bei entsprechender Programmierung kann eine Anwendung die in Frage kommenden Routinen mittels Trace feststellen und ihre unerwünschten Resultate umgehen. Bei einer Installation des so erweiterten Loggers aus dem AUTO-Ordner heraus verweigert das System jedoch beim Aufbau des Desktops unwiderruflich seine Mitarbeit. Bei genauerer Untersuchung stellte ich fest, daß nicht alle über den Line-F-Emulator angesprungenen Unterprogramme im Trace-Modus durchlaufen werden können. Das Problem kann aber doch durch eine Kombination von Änderungen in den Routinen für Line-F, Trace und illegale Adresse gelöst werden. Dabei habe ich den Umstand genutzt, daß die hier interessierenden Unterprogramme mit einem Return-Befehl (RTS) ab-

schließen. Die Ausnahmebehandlung für Line-F muß dann so erfolgen, daß bei Nutzung des Emulators als Sprungverteiler die erzeugte Rücksprungadresse illegal ist. Bei Beendigung einer auf diesem Wege angesprungenen Routine erkennt der Prozessor dann die illegale Adresse und kann den Vektor für Trap 2 gegebenenfalls wieder aufsetzen.

```
014A74  MOVE.W   (A7)+,D2          ; hole Kopie des alten Status'
014A76  MOVE.L   (A7)+,A0          ; und Opcode (FXXX), Rücksprung-
014A78  MOVE.W   (A0)+,D1          ; adresse jetzt in A0,
014A7A  BTST     #0,D1             ; Opcode testen, gegebenenfalls
014A7E  BNE      $014A94           ; UNLK-Funktion (...) ausführen,
014A80  MOVE.W   D2,SR             ; sonst Status restaurieren,
014A82  MOVE.L   A0,-(A7)          ; Rücksprungadresse auf Stack,
014A84  ANDI.W   #$0FFF,D1         ; Adresse der
014A88  MOVE.L   #$00FEE56A,A0     ; gewünschten Routine aus
014A8E  MOVE.L   $00(A0,D1.W),A0   ; der Tabelle holen und
014A92  JMP      (A0)              ; anspringen
014A94  ...                        ; ab hier Unlink-Funktion
```

Listing 1: Der Line-F-Emulator

Genug der langen Einleitung, ich komme zur Sache. Der ausführbare Code beginnt wie üblich mit der Berechnung des Stackpointers und der Programmngröße, dann wird Trap 1 nach dem XBRA-Standard abgeklopft. Befindet sich das Programm schon resident im Rechner, wird sein Status geändert, und eine entsprechende Meldung erscheint auf dem Monitor. Andernfalls wird der Logger für GEMDOS eingerichtet, und das Programm bleibt nach Beendigung im Speicher. Der beim Rücksprung in das Betriebssystem ausgelöste Trap 1 führt sofort wieder in die Anwendung zum GEMDOS-Logger. Dieser besteht aus einem ersten Teil, der die Installation für Trap 2 besorgen soll, und einem zweiten, der die Protokollierung für Trap 1 erledigt. Sobald das entsprechende Bit in der Steuervariablen ein fertig installiertes Desktop anzeigt, wird der erste Teil übersprungen. Solange das aber, wie auch beim ersten Durchlauf, noch nicht zutrifft, testet das Programm, ob das Desktop läuft. Dabei wird davon ausgegangen, daß nach Ende der Boot-Phase unterschiedliche Ausnahmebehandlungen für Line-F und Trace existieren. Verläuft der Test negativ, wird bei den folgenden Aufrufen des GEMDOS geprüft, ob sowohl Line-F-Emulator als auch AES/VDI zwischenzeitlich eingebunden wurden. Sobald das Desktop dann läuft, wird der Merker gesetzt, und die Vektoren für illegale Adresse, Line-F-Emulator, Trace und Trap 2 werden geändert. Die Installation ist damit beendet, und die Aufrufe von AES und VDI werden ebenfalls über den Drucker aufgelistet.

Für die Ausgabe des Protokolls wird bei Trap 1 und auch Trap 2 dasselbe Unterprogramm benutzt. Es beinhaltet außerdem noch die Zeitlupe und läuft über das BIOS. Zur Steuerung habe ich die Eingangssignale CTS (clear to send) und CD (carrier detect) bzw. die Ausgangssignale RTS (request to send) und DTR (data terminal ready) der seriellen Schnittstelle vorgesehen. An Elektrikteilen braucht man einen 25poligen Stecker, ein Stück 4poliges Kabel und zwei Schalter. Die Teile sind jetzt so zu verlöten, daß mit dem einen Schalter CTS und RTS, mit dem anderen CD und DTR verbunden bzw. getrennt werden können. Am Parallel-Port des MFP 68901 kann dann die Schalterstellung gelesen werden.

Was passiert nun, wenn das Betriebssystem den Line-F-Emulator aufruft? Die modifizierte Ausnahmebehandlung prüft zunächst das niedrigste Bit des Opcodes, denn wenn der Line-F-Emulators im Unlink-Modus benutzt werden soll, kann ja ab hier wie bisher verfahren werden. Ansonsten wird der dem aktuellen Opcode-Wert entsprechende, beim Start des Loggers mit $FF vorbesetzte Merker getestet. Beinhaltet der Merker den Wert $00, so kann eine Reinitialisierung des Vektors für Trap 2 durch die zuzuordnende Routine ausgeschlossen werden; ohne Manipulationen läuft die Ausnahmebehandlung dann weiter über den Line-F-Emulator des GEM. Andernfalls wird die analog zur Vorgehensweise des Vorbildes erzeugte Rücksprungadresse auf dem Supervisor-Stack abgelegt. Anschließend wird in der Kopie des auf dem Stack abgelegten System-Bytes das Tracebit gesetzt. Schließlich kann über den gespeicherten alten Vektor der Sprung zum Original erfolgen. Hier weicht der Lauf der Dinge nur dann vom bisher bekannten ab, wenn die Kopie des Statuswortes, wie beschrieben, geändert wurde. Sobald dieses vom Stack geholt wird, um den Prozessorstatus zu restaurieren, wird der Trace-Modus eingeschaltet. Das Trace-Programm prüft ab jetzt nach jedem vom Line-F-Emulator ausgeführten Befehl, ob der auf dem entsprechenden Stack befindliche Eintrag identisch ist mit dem vorher abgelegten Rücksprungziel. Sobald dieser Zustand erreicht ist, wird das niedrigste Bit der weiterhin maßgeblichen Adresse gesetzt, die jetzt überzählige auf dem Supervisor-Stack gelöscht. Dann erfolgt die Beendigung des Trace-Modus', und der Aufruf der gewünschten Routine schließt sich an. Nach deren Abarbeitung liest der Prozessor dann einen ungeraden Wert als Rücksprungadresse, also eine illegale Adresse. In der hierdurch ausgelösten Ausnahmebehandlung wird zuerst überprüft, ob sie durch eine Aktion des Loggers ausgelöst wurde, wenn nicht, hagelt es Bomben wie üblich. Ansonsten berichtigt die Routine entweder den Vektor für Trap 2, oder der dem soeben beendeten GEM-Teil zugeordnete Merker wird gelöscht. Dann wird das Programm an der richtigen Adresse fortgesetzt.

Zum Schluß gebe ich noch einige Anregungen zur eventuell nötigen Anpassung des Loggers an eine andere als die vorgesehene Konfiguration. Ein serieller Drucker kann relativ einfach angeschlossen werden, indem man die Stellung der Schalter über den Centronics-Port einliest. Diese Abfrage kann natürlich auch mit Hilfe einer I/O-Karte erfolgen, aber die hat ja nun mal nicht jeder. Der kritische Punkt ist sicherlich der Test des Desktops. Das gilt vor allem für den neuen 1040 STE. Auf ihm, wie auch anderen Derivaten und zukünftigen Versionen des Betriebssystems (TOS030, PAK-68K, ...), die ohne Line-F-Emulator auskommen, kann der Logger in dieser Form nicht zum Laufen gebracht werden. Das ist der Preis, den man für eine solche Programmierung zahlen muß.

Literatur:

[1] Service Manual Mega 1, ATARI Corp.

[2] Brückmann, Englisch, Gerits, ATARI ST Intern, Data Becker GmbH Düsseldorf

DelBak & Find

Der Computer-User, der über sich ewig das Damoklesschwert des Stromausfalls oder des 'Daten auf Disk X defekt?' sieht, läßt natürlich bei jedem Abspeichern das alte File irgendwie umbenennen und überschreibt es selbstverständlich nicht. Was aber, wenn nach arbeitsreicher Nacht alles geklappt hat und man die diversen BAK-, DUP-, SBK- etc. Files nicht mehr benötigt?

Martin Wunderli

Grund für das 1. Utility: DelBak. Und dann doch der Frust: Die wichstigste Datei wurde nicht korrekt gespeichert, das Backup-File mittlerweile gelöscht. Aber irgendwo auf der Platte war doch noch eine alte Version! Nur wo? Schon mal durch 'zig Ordner durchgeklickt? Womit wir beim 2. Utility wären: Find!

DelBak

Das 1. Utility sucht auf einem Laufwerk nach allen Dateien mit den Endungen BAK, DUP und SBK. Diese werden jeweils auf dem Bildschirm angezeigt, wo sie dann der Benutzer mit einer beliebigen Taste außer *n/N* löschen kann. Das zu durchsuchende Laufwerk muß beim Programmstart dem TTP-Programm als Argument übergeben werden.

Beispiel:

delbak d	Alle Backups auf Laufwerk D werden gesucht, angezeigt und eventuell gelöscht.

Find

Mit dem zweiten Utility können Sie Ihre irgendwo auf dem Laufwerk versteckte (wer die da nur hin hat ...?) Datei wiederfinden. Find benötigt neben dem zu durchsuchenden Laufwerk natürlich auch den Namen (ohne Extension) der zu findenden Datei.

Beispiel:

find d test1	Alle Dateien mit dem Namen test1 (z.B. test1.c, test1.txt, test1.bas ...), die sich auf Laufwerk D befinden, werden angezeigt.

Resource-Einbindung in C

Wollen Sie auf dem ATARI ST in Ihrem Programm ein selbsterstelltes Bild, einen neuen Font, eine RSC-Datei oder ein sonstiges Daten-File einbinden? Einige BASIC-Dialekte stellen für solche Zwecke (und für Maschinenspracheroutinen) den INLINE-Befehl zur Verfügung. Was aber z.B. in 'C' und Assembler machen?

Christoph Conrad

Einige Resource-Construction-Sets bieten die Möglichkeit, die Resource-Datei als C-Quelltext auszugeben. Dieses Feature bieten aber nicht alle Resource-Construction-Sets, außerdem erfordert dieser Weg einen mehr oder weniger großen Anpassungsaufwand. Andere Daten-Files können Sie eigentlich nur zur Laufzeit des Programmes dazuladen, oder mit einem Hilfsprogramm in ein C-Quelltext-File umwandeln, in dem ein global initialisiertes char-Array die Bytes des Daten-Files aufnimmt. Eine andere Möglichkeit zeigt das Programm *RSC2OBJ.TTP* (lies: Resource to OBJEKT.TTP) auf, die gerade beim Einbinden von Resourcefiles sehr elegant ist. Die vom Programm erwartete Kommandozeile lautet: *infile outfile labelname* [-r]. *infile* ist der Dateiname des zu konvertierenden Daten-Files. *outfile* ist die konvertierte Objektdatei. *labelname* ist der Name des öffentlichen Labels, unter dessen Adresse Sie Ihre Daten ansprechen (siehe Beispielprogramm).

Um die Arbeitsweise des Konverters zu verstehen, müssen Sie etwas über den Aufbau von Objektdateien, wie Sie Compiler und Assembler erstellen, wissen. Die vorliegenden Betrachtungen sind teilweise etwas vereinfacht und beschränken sich auf den für Sie wesentlichen Aspekt. Das auf dem ATARI ST von den gängigen Compilern/Assemblern benutzte Format ist jenes von Digital-Research (GEM stammt auch daher). In diesen Objekt-Files sind die Größen des Codes, der initialisierten und der uninitialisierten Datenbereiche vermerkt sowie die Code- und Datenbereiche abgelegt. Unsere Daten-Files landen komplett im Datensegment der erzeugten Objektdatei; die Größe des Datenbereiches entspricht der des Daten-Files. Codegröße und Größe der uninitialisierten Datenbereiche werden auf Null gesetzt. Weiterhin gibt es eine Symboltabelle, die hauptsächlich Auskunft gibt über die Namen von Variablen, welche Sie aus einem Programm-Modul exportieren bzw. importieren, und die Art der Variablen. Ein Beispiel: Sie definieren in einem C-Modul eine Variable global (außerhalb jeder Funktion), z.B. *int flag*.

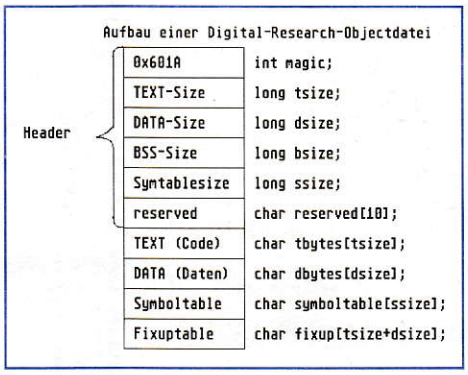


Bild 1: Der Aufbau einer Digital-Research-Objektdatei

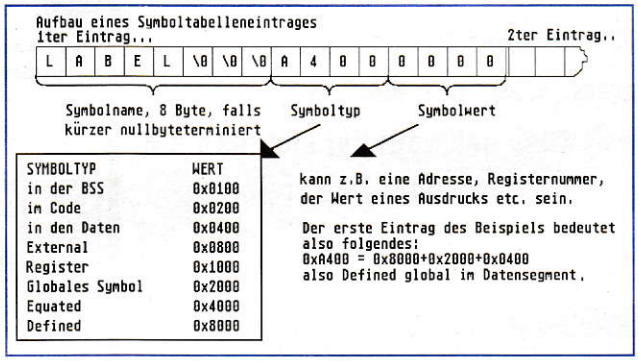


Bild 2: Der Aufbau eines Symboltabelleneintrags

Diese Variable soll in einem zweiten Programm-Modul benutzt werden und wird dort als *extern int flag* deklariert. Der Linker weiß nun aus der Symboltabelle des Objekt-Files von Modul 2, daß in einem anderen Modul eine Variable *flag* existieren muß und findet diese in der Symboltabelle des Objekt-Files von Modul 1. Der Label-Name, den Sie über die Kommandozeile angegeben haben, wird in der Symboltabelle vermerkt als "global definiert im Datensegment" mit dem Symbolwert 0. Dies bedeutet, daß die Adresse dieses Labels direkt auf die Daten (ohne Versatz) zeigt.

Den für uns wichtigsten Teil des Objekt-Files (zumindest bei RSC-Dateien) stellt die Fixup-Tabelle dar. Programme können zur Laufzeit vom Betriebssystem an ganz unterschiedliche Startadressen geladen werden, eben dort, wo gerade genug Platz frei ist. Aus Codegröße- und Geschwindigkeitsgründen benutzen Compiler (und Assembler-Programmierer) aber viele Maschinenbefehle, die auf eine feste Adresse (z.B. einer Variablen) zugreifen, und zwar so, als begänne der Code/die Daten ab Speicheradresse 0. Das bedeutet z.B.: Falls ein Label relativ zum Codesegmentstart $4711 Bytes entfernt liegt, ist im compilierten Maschinencode (Objektdatei) die Zugriffsadresse $4711. Dem Linker kann aber über die Fixup-Tabelle gesagt werden, daß er für solche Zugriffe Einträge im fertigen Programm in der sogenannten Relokationstabelle macht. Der Lader (das ist die Komponente des Betriebssystems, welches ein Programm von Diskette/Platte lädt und alle nötigen Anpassungen vornimmt) wertet solche Einträge aus und addiert bei allen solchen absoluten Adressen die tatsächliche Startadresse des Codes (er 'reloziert' die Adressen). Die Fixup-Tabelle ist genauso groß wie die Größen von Code- und Datensegment zusammengenommen. Jede zu relozierende Adresse (Länge = 4 Bytes = 1 Long) wird durch 4 Bytes in der Fixup-Tabelle vermerkt, wobei der

Abstand des Longs vom Fixup-Tabellenanfang genau dem Abstand des zu relozierenden Longs vom Start des Code- bzw. Datensegments entspricht. Für die Konversion sind zwei beschreibende Werte der Fixup-Tabelle wichtig: 0x00000000 und 0x00050001. Der erste Wert führt dazu, das der Linker in der Relokationstabelle des fertig gelinkten Programms einen Wert einträgt, der dem Lader "bitte dieses Wort (long) nicht verändern!" mitteilt. Der zweite Wert sagt dem Linker, daß es sich um ein zu relozierendes Long im Datensegment handelt, was er entsprechend in der Relokationstabelle berücksichtigt. Bei reinen Daten-Files [KEIN Switch -r (!)] wird die gesamte Fixup-Tabelle mit Nullen aufgefüllt, es sollen ja auch keine Daten vom Lader verändert werden. Bei Resourcefiles existieren eine Menge interner Zeiger, z.B. auf TEDINFOs, ICONBLKs, BITBLKs sowie innerhalb dieser Strukturen Zeiger auf Daten etc. (siehe [2],[6]). Alle diese Zeigerwerte beziehen sich aber auf den Anfang der Resource-Datei und werden von *rsrc_load(..)* beim Einladen der Resourcedaten auf die absoluten Speicheradressen durch Addition der RSC-Speicherbasisadresse angepaßt. Na, klingelt's? Das ist ja genau dasselbe wie beim Relozieren! Wenn wir in der Fixup-Tabelle diese Zeiger als relozierbar markieren, wird der Lader uns alle RSC-internen Zeiger setzen.

Die einzige zur Laufzeit notwendige Anpassung besteht in der Umrechnung von Zeichen- in Pixel-Koordinaten. In Resourcefiles sind nämlich die Positionen und Abmessungen der Objekte nicht in Pixeln, sondern in Zeichenbreiten bzw. Zeichenhöhen angegeben, wobei noch ein vorzeichenbehafteter Byteoffset (-128..127) einen Pixel-Versatz angeben kann. Diese Umrechnung kann sehr bequem mittels der AES-Funktion *rsrc_obfix(..)* gemacht werden. Die Funktion *rsrc_gaddr(..)* muß durch eine eigendefinierte Funktion ersetzt werden, da die Originalfunktion auf Daten im global-Array zurückgreift, die durch *rsrc_load* gesetzt werden.

Bei der Konversion von Resourcefiles (Flag -r) wird sicherheitshalber noch ein kleiner Test vorgenommen, ob es sich wirklich um ein Resourcefile handelt, damit beim versehentlichen Setzen dieses Flags normale Daten-Files durch die Relokationsinformationen nicht verunstaltet werden. Aber nicht nur deswegen: Sämtliche relativen Zeiger beziehen sich auf den Resource-Dateistart (wie schon erwähnt) und können deshalb auch nicht negativ sein. Die für die Konversion allokierten Speicherbereiche liegen meist direkt hinter dem Konversionsprogramm im Speicher (an höheren Adressen). Negative Zeiger-Offsets (das sind bei normalen Daten-Files fälschlicherweise als Zeiger-Offsets aufgefaßte Daten) könnten (beim 'Poken' der Fixup-Werte in den dafür allozierten Buffer) das Konversionsprogramm teilweise überschreiben und zu Programmabstürzen führen. Details zum Aufbau der Resource-Dateien würden den Rahmen dieses Artikels bei weitem sprengen. Ich verweise speziell auf [2],[3] und 'die Bibel' [6].

Im Programm setze ich zur Behandlung von Ausnahmesituationen (sprich Fehlern) 'Goto's ein. Da Sie diesen Satz noch lesen, darf ich davon ausgehen, das Sie nicht zu den absoluten Anti-'Goto'-Puristen gehören, welche sicherlich gerade damit beschäftigt sind, diesen Artikel fein säuberlich aus dieser Zeitung herauszutrennen. Daß 'Goto's zum Verlassen tief verschachtelter Kontrollstrukturen gut geignet sind und eine Menge überflüssiger if-Abfragen ersparen, hat sich ja zum Glück herumgesprochen. Aber auch bei Fehlern ersparen Sie es sich, jedesmal den Ausnahmebehandlungscode durchzulesen, was das Verständnis der eigentlichen Programmabsicht doch erheblich stört. Außerdem sind alle Fehlerbehandlungsmaßnahmen übersichtlich am Ende einer Routine gesammelt.

Zum Beispielprogramm können Sie eine beliebige, mit einem Resource-Construction-Set erstellte und durch den Konverter geschleuste Re- source-Datei dazulinken. Der Label-Name sollte *recource* sein. Das Programm zeigt Ihnen alle Objektbäume zentriert auf dem Bildschirm. Menüleisten werden normalerweise nicht zentriert; hier schon, da das Programm nicht wissen kann, ob Ihre Resource eine Menüleiste enthält. Bei Menüs bleibt etwas Pixel-Schrott am oberen Bildschirmrand.

Literatur:

[1] Jürgen Schultz-Kappler, Dialog über Tasten, c't 3/88

[2] Stefan Höhn, Objektstrukturen im AES, ST-Computer Sonderheft 2 (sehr informativ und verständlich!)

[3] Stefan Höhn, Einführung in das Resource-Construction-Set von Digital Research, ST-Computer Sonderheft Nr. 2

[4] Jürgen Leonhard, Resource-Datei? Nein danke!, ST-Computer 12/87

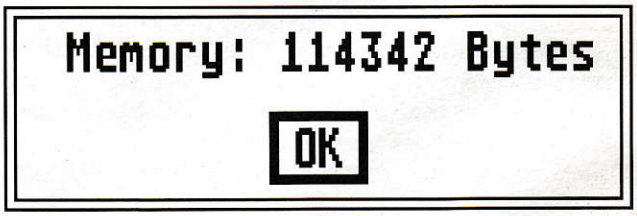
[5] Lutz Preßler, Resourcen in GFA-Basic 3.0, ST-Computer 10/88

[6] Jankowski/Reschke/Rabich, ATARI Profibuch ST-STE-TT, Sybex Verlag

Freier Speicherplatz

Jürgen Stessun

Das Accessory RAM_FREE hat zwei nützliche Eigenschaften: Zum einen zeigt es nach Aufruf den noch im Rechner vorhandenen freien Speicher an, und zum zweiten ist es nur etwas mehr als 500 Bytes lang. RAM_FREE wird, wie jedes Accessory, zur Installation in den ROOT-Ordner der Boot-Partition kopiert. Bei Bedarf kann es dann durch Anklicken in der Menüzeile aktiviert werden.



Fast-File- und Link-Viren-Finder

Vor einiger Zeit war es mal wieder soweit: Ich suchte das Backup von einem Source-Text auf einer der 10 Partitions meiner Megafile 60. Aber in welchem Ordner befand es sich? Eine Szene später: Habe ich nun einen Link-Virus auf der Platte oder nicht? Sagrotan starten und warten, ...

Markus Fritze

Wie sich der geneigte Leser sicherlich vorstellen kann, habe ich für beide Probleme eine Lösung anzubieten - und zwar in Form eines Fast-File-Finders bzw. des Link-Virus-Finders:

Der Fast-File-Finder (FFF) ist in der Lage, die gesamte Festplatte nach einem Dateinamen zu durchsuchen. Ganz nebenbei ermittelt er auch noch, wieviel Platz auf jeder Partition noch frei ist.

Der Link-Virus-Finder (LVF) durchsucht alle ausführbaren Programme auf der Festplatte nach Link-Viren. Er tut dies, indem er den Programm-Header mit den Mustern des VCS und des Milzbrand-Virus' vergleicht. Beim jedem Programm wird zudem eine Prüfsumme über die ersten 256 Bytes berechnet, mit welcher bei jedem neuerlichen Start vom LVF verglichen wird. Ach ja, der LVF braucht für meine 10 Partitions (mit über 2800 Dateien in mehr als 280 Ordnern) etwa 13.5s, um festzustellen, ob Link-Viren vorhanden sind oder nicht. Ich denke, daß ist im Vergleich zu 15 min bei Sagrotan angenehm schnell ...

Kommen wir nun zum Eigentlichen: dem Programm. Wie man leicht feststellen kann, ist das Programm in Assembler geschrieben. Es besteht aus einigen Teilen, die man auch prima einzeln verwenden kann. Deshalb glaube ich, daß jeder Assembler-Programmierer etwas von dem Listing hat. Es ist recht gut dokumentiert.

Da der FFF und der LVF in wesentlichen Teilen gleich sind, gibt es nur ein Listing, aus welchem man mit dem Flag *virus* wahlweise den FFF oder den LVF erzeugen kann.

Um ein Laufwerk nach Link-Viren oder Dateien zu durchsuchen, wird zuerst die Routine *set_drive()* für das zu durchsuchende Laufwerk aufgerufen. Diese Routine ermittelt den BPB des Laufwerkes, liest die FAT und ermittelt so ganz nebenbei noch dessen freien Speicherplatz.

Nun wird *read_dir()* aufgerufen. Diese Routine liest das Root-Directory des Laufwerkes ein und ruft für jeden Ordner die Routine *read_sub_dir()* auf. Diese Routine liest dann rekursiv alle weiteren Ordner ein.

Der Speicherplatz für die Ordner wird durch die Routine *get_mem()* ermittelt, die eine Heap-Verwaltung darstellt. Das Prinzip eines solchen Heaps ist sehr einfach: Man hat einen Zeiger auf einen großen Speicherblock - den Heap. Will nun jemand Speicher allozieren, wird dieser Wert des Zeigers zurückgegeben. Das ist die Adresse des Speicherblockes, den man angefordert hat. Bevor das Unterprogramm verlassen wird, erhöht man diesen Zeiger noch um die Größe des angeforderten Speicherblockes, so daß bei einem erneuten Aufruf der Zeiger hinter den ersten Speicherblock zeigt.

Noch zwei Dinge gilt es zu beachten: Die Größe des Speicherblockes sollte aufgerundet werden, damit dieser stets auf einer geraden Adresse anfängt. Und man sollte abfragen, ob der Zeiger hinter den Speicherblock zeigt, der für den Heap reserviert ist. Wenn dem so ist, reicht der Speicher nicht, und das Programm wird abgebrochen.

Jetzt wird sich der eine oder andere eventuell noch fragen: Wie kann man denn Speicherblöcke wieder freigeben? Die Antwort ist ganz einfach: gar nicht! Das ist bei unserem Directory-Baum auch gar nicht nötig, da wir vor jedem erneuten Einlesen eines Baumes den Heap-Pointer einfach wieder zurück auf den Anfang setzen und somit alle Blöcke freigegeben haben.

So, nach diesem Ausflug in die Heap-Verwaltung kommen wir zur Routine *hunt_dir()*. Sie durchsucht einen eingelesenen Directory-Baum nach Dateien bzw. die Dateien nach Viren. Auch diese Routine funktioniert rekursiv, d.h. wenn ein Ordner gefunden wird, ruft sie sich mit einem Zeiger auf den Ordner und der Anzahl der Dateien des Ordners erneut auf. Wenn der Ordner komplett durchsucht wurde, wird zurückgekehrt, und es geht weiter.

In dieser Routine besteht auch der größte Unterschied zwischen dem FFF und dem LVF. Während der FFF den Dateinamen nur mit seiner Suchmaske vergleicht und den ihn mit Pfad gegebenenfalls ausgibt, hat der LVF erheblich mehr zu tun. Wenn ein erster Vergleich der Extensions „PR?", „TO?", „TT?", „AC?", „AP?", „DR?" (Treiber), „SY?" (Festplattentreiber!) positiv ist, wird der erste Sektor dieser Datei eingelesen. Bei einer Programmdatei müssen am Anfang die Bytes $601A stehen. Nun wird der Anfang des TEXT-Segmentes mit den zwei bekannten Link-Viren verglichen.

Wenn kein Link-Virus vorhanden ist, wird eine Prüfsumme über die ersten 256 Bytes der Datei berechnet. Dies klingt zwar nach sehr wenig, wenn man bedenkt, daß z.B. Sagrotan eine CRC-Prüfsumme über die gesamte Datei berechnet, reicht aber, da in den ersten 256 Bytes die Länge der einzelnen Programmsegmente steht, welche durch einen Virus an sich immer verändert wird. Zudem muß sich ein Virus am Programmanfang aufrufen, und über genau diese Stelle wird die Prüfsumme ja auch berechnet. Wie man sieht, wird zwar Zeit gespart, aber die Sicherheit ist immer noch gewährleistet.

Nach der Berechnung der Prüfsumme wird in der Vergleichsliste nach dem Programm gesucht. Wenn das Programm bereits in der Liste steht, wird die Prüfsumme in der Liste gesucht, denn für jedes Programm merkt sich der LVF bis zu sieben Prüfsummen. Das ist sehr praktisch, wenn man z.B. verschiedene Programmversionen auf der Festplatte hat. Wenn die Prüfsumme nicht in der Liste steht, wird gefragt, ob die neue Prüfsumme mit in die Liste aufgenommen werden soll. Steht der Programmname noch nicht in der Liste, wird ein neuer Eintrag für das Programm angelegt und eine entsprechende Meldung ausgegeben. Die Vergleichsliste wird beim Start vom LVF automatisch geladen, und sie kann vor dem Programmende natürlich auch geschrieben werden.

Sowohl LVF als auch auf FFF können in der Kommandozeile einige Parameter übergeben werden.

Mit einem „-" am Anfang der Kommandozeile wird die Ausgabe der Dateinamen unterdrückt. Beim LVF spart es einfach Zeit, beim FFF ist es lediglich dann praktisch, wenn man nur wissen wollte, wieviel „*.TXT"-Dateien z.B. auf der Festplatte stehen, aber nicht, wo sie sind. Eventuell hat man den FFF auch nur gestartet, um festzustellen, auf welcher Partition noch Platz ist, denn das wird ja praktischerweise auch ausgegeben.

Mit einem „+" wird beim LVF eine unbekannte Datei oder eine unbekannte Prüfsumme automatisch, d.h. ohne Rückfrage, übernommen. Das ist insbesondere dann praktisch, wenn man den LVF zum ersten Mal, also ohne die „LVF.DAT"-Datei startet.

Sowohl beim LVF als auch beim FFF kann man mit „:x" nur das Laufwerk „x" durchsuchen lassen. Beispiel: „:C", es wird nur das Laufwerk C: durchsucht. Ansonsten wird ab Laufwerk „C:" bis zum letzten benutzten Laufwerk alles durchsucht - also üblicherweise die gesamte Festplatte.

Beim FFF kann man nun noch eine Suchmaske angeben. Diese kann ein kompletter Dateiname wie z.B. „TEST.DOC" sein oder aber auch nur ein Teil davon („T*.D?C"), wie beim GEMDOS also.

Literatur:

[1] Scheibenkleister II, MAXON Computer
[2] Atari Profibuch ST-STE-TT, Sybex
[3] Anleitung zum TurboAss

VIRSPY
Der GEMDOS-Türwächter

Wem ist so etwas noch nicht passiert: Man kopiert sich ein Programm auf die RAM- oder Harddisk, aber es beschwert sich nach dem Starten darüber, daß noch irgendein File fehle. Meistens liegt es an falschen Suchpfaden oder Namen.

Gerrit Gehnen

Mit VIRSPY hat man ein Werkzeug in der Hand, mit dem die Diskettenoperationen auf dem Drucker mitgeschrieben werden. Außerdem: Wen interessiert es nicht, ob sein Lieblings-Compiler irgendwelche Zwischendateien anlegt und wo sie erzeugt werden? Nebenbei ist VIRSPY ein Lehrstück zum Schreiben von residenten Programmen mit XBRA-Protokoll und sauberer Installation. VIRSPY fängt die GEMDOS-Aufrufe ab, die in solchen Fällen benutzt werden, nämlich *Pexec()*, *Fopen()*, *Fcreate()* und *Sfirst()*. Allerdings hatte VIRSPY ursprünglich eine ganz andere Aufgabe (wie schon der Name sagt), nämlich die Suche nach Link-Viren, die sich an Programme anhängen und von da aus munter weiterverbreiten.

In der Routine, die zwischen Benutzer (Programm oder Desktop) und GEMDOS eingeschoben wird, werden diese Funktionen analysiert und der dazu passende Pfad- und File-Namen auf dem Drucker protokolliert. Wem nach erfolgreicher Analyse das Druckergeratter zuviel wird, braucht nicht einen Reset auszulösen, sondern nur VIRSPY noch einmal zu starten; dadurch wird ein Flag umgesetzt, mit dem die Druckerausgabe gesteuert wird.

Zum GEMDOS hin ist Virspy vollkommen transparent, d.h. alles, was vorne reingeschickt wird, kommt auch hinten wieder raus (wenn auch mit einer kleinen Verzögerung wegen der Druckerausgabe).

Bei der Entwicklung habe ich auf möglichst geringen Speicherverbrauch geachtet: die Installationsroutine fliegt, nach getaner Arbeit, raus. Wer sich ein wenig mit der Parameterausgabe ans GEMDOS auskennt, dem dürfte es nicht schwerfallen, das Programmauf weitere Funktionen auszuweiten und damit noch flexibler zu machen.

File-Info

Jeder ST-Besitzer kennt selbstverständlich die Option "zeige Info" im Desktop-Menü. Für eine ausgewählte Datei erscheint beim Aufruf eine Dialogbox, die Informationen über diese anzeigt. Dabei handelt es sich um Angaben über den Dateinamen, Erstellungsdatum und -zeit sowie über das Schreibschutzattribut. Ändern lassen sich leider nur dieses Attribut und der Dateiname.

Uwe Seimet

Will man jedoch die Erstellungszeit oder andere Attribute ändern, läßt sich nichts machen. Dabei ist es manchmal sinnvoll, nachträglich das Erstellungsdatum einer Datei ändern zu können, z.B. dann, wenn sich nach einem Kopiervorgang zeigt, daß man wieder einmal vergessen hat, die Uhr im Kontrollfeld zu stellen. Das Ergebnis: Die Datei weist ein unsinniges Erstellungsdatum auf. Bisher blieb in einem solchen Fall nichts anderes übrig, als den Kopiervorgang zu wiederholen (natürlich nach dem Stellen der Uhr), mit einem Disk-Monitor zu drohen oder sich gar ganz geschlagen zu geben.

Einstellungen

Mit dem Accessory *FILEINFO* ist dieser Ärger nun vorbei. Für die über den Fileselector angewählte Datei erhält man eine ausführlichere Information als das Desktop sie bietet, wobei alle angezeigten Angaben auch geändert werden können. Neben dem Schreibschutzattribut kann eingestellt werden, ob es sich bei der ausgewählten Datei in Zukunft um eine Systemdatei oder um eine versteckte handeln soll. Zur Auswahl der Attribute genügt es, die dafür vorgesehen Buttons in der Dialogbox anzuklicken. Die Einstellung der Buttons beim Betreten der Box entspricht den Attributen, wie sie die Datei im Augenblick des Aufrufs besitzt. Besonders interessant sind die Dateiattribute *System* und *Versteckt,* die vom Desktop aus nicht geändert werden können. Geben Sie einer Datei das Attribut *System*, wird sie nicht mehr auf dem Desktop angezeigt, und auch für den normalen Fileselector ist sie nicht mehr vorhanden. Dennoch kann der Inhalt dieser Datei gelesen werden, denn sie ist nicht wirklich von der Diskette verschwunden. Wozu das gut sein kann? Nun, wenn Sie Disketten besitzen, die hoffnungslos überfüllt sind, geben Sie allen .RSC-Dateien doch einfach den Systemstatus. Danach sieht der Disketteninhalt schon viel übersichtlicher aus, denn die .RSC-Dateien werden nicht mehr angezeigt.

Das Betriebssystem bzw. die Programme, die die .RSC-Dateien lesen sollen, bemerken von der Änderung nichts. Selbstverständlich kann man dieses Verfahren auch auf andere Dateien anwenden, die nur von anderen Programmen nachgeladen werden, also ohnehin nicht über einen Mausklick gestartet werden können. Geben Sie einer Datei das Attribut *Versteckt*, so ist sie für das TOS so gut wie gar nicht mehr zu finden, es sei denn, man geht mit einem Disk-Monitor auf die Suche.

Kopierschutz

Auf diese Art und Weise kann ein einfacher Kopierschutz für einzelne Dateien realisiert werden, denn wenn man nicht weiß, daß sich eine Datei auf einer Diskette befindet, kommt man auch nicht auf die Idee, sie kopieren zu wollen. (Selbstverständlich hilft dieses Verfahren nur dann, wenn nicht gleich ein Backup der kompletten Disk gemacht wird.) Daraus resultiert jedoch auch folgende Warnung: Merken Sie sich, welche Dateien Sie versteckt haben, denn nur wenn Sie die Dateinamen kennen, können Sie diese Files wieder sichtbar machen! Hierzu rufen Sie erneut das FILEINFO-Accessory auf, geben den Namen der unsichtbaren Datei über die Tastatur ein (der Fileselector zeigt diesen Namen nicht an!), und voilà: Die Information über die anscheinend nicht vorhandene Datei erscheint auf dem Bildschirm. Ändern Sie nun die File-Attribute entsprechend ab, wird die Datei ab sofort wieder für jedermann sichtbar.

Wichtig ist noch der folgende Hinweis: Wird die Datei, deren Namen oder Attribut Sie soeben geändert haben, in einem Window auf dem Desktop angezeigt, muß die ESC-Taste gedrückt werden, damit die vorgenommenen Änderungen auf das Desktop übertragen werden. Welche Routinen des GEMDOS ermöglichen nun das Holen und Ändern der File-Daten? Für das Holen und Setzen der File-Attribute ist die Funktion *CHANGE MODE* ($31) verantwortlich. Sie ermöglicht das Setzen oder Löschen einzelner Attribute. Erstellungsdatum und -zeit werden mit der Funktion *GSDTOF* ($57) geholt oder gesetzt. Als Parameter wird unter anderem ein Flag übergeben, das bestimmt, ob die Daten ermittelt oder gesetzt werden sollen.

XBRA

Das Programm XBRA.TOS zeigt Ihnen alle, mittels XBRA-Verfahren installierten Programme auf, die sich in Systemvektoren eingehängt haben. Sie haben dann die Möglichkeit einzelne Programme zu deaktivieren. Bedenken Sie dabei aber, daß Sie das Programm zwar aus der Vektor-Routine aushängen, es von nun an nicht mehr aufgerufen wird, der vom Programm reservierte Speicher, und natürlich der vom Programm belegte, jedoch nicht freigegeben wird.

Andreas Kohler

Kernstück des Programms ist die Funktion *getxbra()*. Man übergibt ihr in *vec* die Adresse einer Systemvariablen. Der Inhalt der Systemvariablen und Ihre Adresse werden zunächst auf Plausibilität geprüft, der Zugriff auf eine nichtexistierende oder ungerade Adresse würde nämlich zum Absturz führen. Sollte die in die Systemvariable eingetragene Routine dem XBRA-Standard entsprechen, so liefert die Funktion 1 zurück und legt in *name* die Programmkennung und in *next* die Adresse des nächsten Routinezeigers ab, sonst wird 0 zurückgeliefert. Beim nächsten Aufruf von *getxbra()* kann der Wert von *next* wieder in *vec* übergeben werden, um die XBRA-Kette weiter zu verfolgen. Ein Beispiel für dieses Vorgehen finden Sie in der Funktion *prnpage()*.

un_link() entfernt aus allen XBRA-Ketten die Routine mit der Programmkennung *id*. Sollte die Kennung mehrmals in der Kette auftauchen, wird nur der erste Eintrag (d.h. das zuletzt installierte Programm) entfernt. Nicht alle Systemvektoren stehen an einer festen Adresse, manchmal muß diese auch erst erfragt werden. Ein Beispiel dafür findet sich in *varinit()*.

Nach dem Start zeigt das Programm die XBRA-Ketten für eine Reihe von wichtigen Systemvektoren an. Die Umschaltung der Seiten durch direkte Eingabe der Seitennummer ist möglich. Will man nun ein Programm 'aushängen', so kann man nach der Anwahl der Option *R* die Programmidentifikation eingeben (Klein- und Großbuchstaben werden unterschieden). Nach einer Rückfrage entfernt das Programm die Routine mit dieser *ID* aus allen Ketten. Mit der Escape-Taste können Sie das Programm beenden.

Directory als Baumdiagramm

Das Desktop stellt mit seinen Fenstern zwar komfortabel Ausschnitte der Ordnerstruktur dar, möchte man aber einen Überblick über das gesamte Directory einer Diskette oder gar einer Festplatten-Partition gewinnen, reicht die Darstellung nicht mehr aus. Eines der 'Norton Utilities' (für PCs) stellt u.a. die Ordnerstruktur als Baumdiagramm dar, genau dies leistet auch das hier vorgestellte Programm.

Peter Ubachs

Zunächst wird der Pfad ausgewählt, von wo aus der Dateibaum gezeichnet werden soll. Danach wird das Ausgabegerät bestimmt. Die Bildschirmausgabe läßt sich mit der linken Maustaste anhalten, mit der rechten geht es weiter. Bei der Datei-Ausgabe muß man noch den Namen der Ausgabedatei eingeben. Beim Ausdruck erfolgt keine Anpassung der Sonderzeichen (hier '|' und '\'). Diese muß man für seinen Drucker evtl. noch selber einfügen.

Wie wird das Baumdiagramm gezeichnet?

Da Bäume rekursive Datenstrukturen sind, geht man beim deren Aufbau und Zeichnen am einfachsten auch rekursiv vor. Die Prozedur *direktory_zusammensuchen()* ruft sich solange selber auf, bis keine weiteren Dateien mehr gefunden werden. Zur Dateisuche werden die GEMDOS-Funktionen *Fsfirst()* und *Fsnext()* verwendet. *Fsfirst()* durchsucht das aktuelle Directory nach Dateien, auf die der angegebene Name paßt. Dabei bestimmt der zweite Parameter *attribut*, nach welchem Dateinamen gesucht werden soll (16 bedeutet hierbei, daß auch nach Ordnern gesucht werden soll). *Fsnext()* setzt die Suche fort. Die Prozedur *dateinamen_bestimmen()* liest schließlich den Dateinamen aus.

Der Ton 'A'

Ein Blick in die Atari-Dokumentation (oder entsprechende Fachliteratur) verrät uns, daß mit dem XBIOS-Aufruf *Dosound()* die Soundgeneratoren angesteuert werden. Dazu muß ein Zeiger auf einen String übergeben werden, der den Soundchip programmiert. Details mag der interessierte Leser der entsprechenden Literatur entnehmen. Uns soll hier genügen, daß der String sound genau dies tut, und daß an der Position 1 das Low-Byte und an der Position 3 das Highbyte des Teilers einzutragen ist. Die Frequenz wird durch die Formel 125000Hz / Teiler bestimmt. Unsere musikerzeugende Routine ton muß also den Benutzerzeiger *USERP1* aus dem erweiterten Objekt ermitteln, um daraus den Teiler zu berechnen. Um möglichst wenig Genauigkeit zu verlieren, wurden die Werte vor der Division mit 1000 multipliziert. Anschließend wird die Routine *Dosound()* aufgerufen. So, nun brauchen wir nur noch den Compiler zu starten und können, ohne daß wir uns um die Fensterverwaltung direkt gekümmert haben, bis zu sieben *PIANO*-Fenster öffnen und uns, bei Anwahl der Taste (mittels Maus oder durch Drücken von A auf der Tastatur), am Ton A erfreuen.

Leider werden wir beim Ausprobieren, vor allem bei Verwendung der Tastatur, eine Unschönheit feststellen: die Tasten klicken, und die Tastenwiederholung ist auch unpassend. Außerdem ist die visuelle Rückmeldung (Invertierung der Taste) doch ein sehr kurz geratenes Blinzeln. Und wenn wir schon bei Verbesserungsvorschlägen sind: Wie wäre es, wenn das Programm auch als Accessory laufen würde, wobei genau unser PIANO-Fenster erscheinen würde. Und außerdem haben wir natürlich immer noch die Erzeugung eines 'richtigen' Keyboards im Hinterkopf.

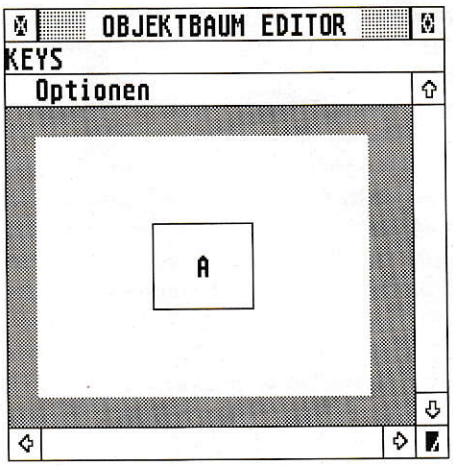


Bild 2: Die Taste 'A' im Objektbaum-Editor

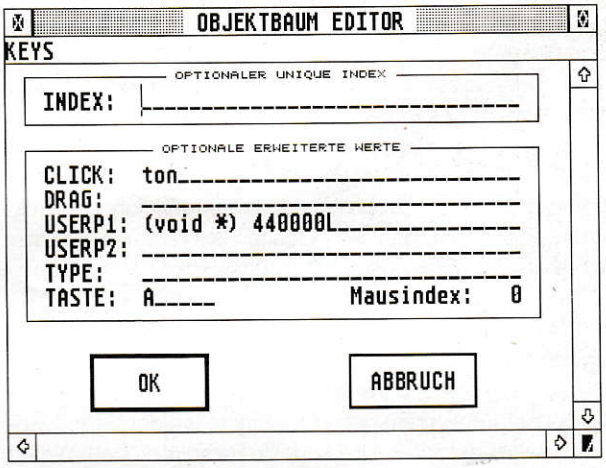


Bild 3: Notwendige Einträge in die erweiterte Objektstruktur

Tastaturklick

In der Initialisierungs-Routine *ACSinit()* können mit Hilfe der Variablen *conterm* der Tastaturklick und die Tastenwiederholung ausgeschaltet werden. Als ordentliche Programmierer merken wir uns den Wert dieser Variablen und restaurieren ihn beim Verlassen des Programmes. Hierzu dient die *ACSterm*-Routine, die zum Abschluß des Programms durchlaufen wird. Um die visuelle Rückmeldung deutlicher zu gestalten, fügen wir in die Routine *ton* einfach eine Warteschleife ein, die 80 Millisekunden wartet.

Keyboard

Für ein 'richtiges' Keyboard benötigen wir noch einmal den ACS-Editor. Mit seiner Hilfe bauen wir ein schönes Keyboard auf (Bild 1). Für die weißen Tasten benutzen wir als Auslösetasten die untere Tastenreihe von '<' bis '-', für die schwarzen die entsprechenden Tasten der Reihe darüber. Außerdem wird für die Startphase als Accessory das Fenster *PIANO* als Root-Fenster festgelegt.

Wenn wir nun das übersetzte Programm starten, erhalten wir in der Tat ein Keyboard, das, mit der Tastatur bedienbar, Töne spielt [Wie Sie vielleicht gemerkt haben, haben wir an der Routine *ton()* zur Erzeugung des Sounds nichts geändert! Den benötigten Parameter für die verschiedenen Tonhöhen werden ja jeweils vom ACS in die erweiterte Objektstruktur eingetragen und von unserer Routine ausgelesen; sofern Sie beim Erzeugen der neuen Tasten den jeweiligen Wert in *USERP1* eingetragen haben].

Nachdem Sie ein so schönes Keyboard besitzen und bereits einige Melodien gespielt haben, kommt Ihnen vielleicht der Gedanke, ob man nicht das Spielen aufnehmen könnte, um es ein weiteres Mal zu hören.

Rekorder

Wir müssen dazu Tastenanschläge speichern und später wieder abspielen. Dazu definieren wir im ACS-Editor zwei neue Buttons: *START* und *PLAY*. *START* spult quasi ein Band an den Anfang; man hätte es auch als Löschen bezeichnen können. *PLAY* wiederholt alle Tastenanschläge. Da wir für das Band einen Eintrag in der Komponente *USER* des Fenstera benötigen, die wir bei Beenden des Fensters auch wieder freigeben müssen, muß nun auch eine eigene Service-Routine angelegt werden. In die Komponente *USER* tragen wir erst zur Laufzeit eine dynamisch erzeugte Struktur ein, die zur Aufnahme der Tastenfolge dient.

Wenden wir uns dem Applikationsteil zu. Selbstverständlich definieren wir auch die Prototypen der neuen Routinen. Wie soll die Datenstruktur aussehen, die die Anschläge speichert? Wir legen einfach ein Feld an, dessen Einträge aus einer Struktur aus Index und Zeitpunkt bestehen. Außerdem muß noch ein Zeiger auf den nächsten zu schreibenden

Eintrag verwaltet werden. Die Struktur tape leistet genau das.

Die *create*-Routine *piano_make()* muß mittels *Ax_malloc()* die Datenstruktur *tape* anfordern, die in der Komponente *USER* des Fensters eingetragen wird. Die Initialisierung darf nicht vergessen werden.

Da beim Beenden des Fensters korrekterweise auch die eigenen Daten freizugeben sind, müssen wir eine eigene Terminier-Routine angeben. Sie ist sehr einfach. Wenn das System dem Fenster die Aufforderung zum Terminieren sendet (*AS_TERM*), werden die *USER*- und die Fensterstruktur freigegeben.

Die Routine *start()* ist ebenfalls sehr einfach. Sie setzt nur den Schreibzeiger auf den ersten Wert zurück.

Die Routine *ton()* muß so ergänzt werden, daß der Index des angewählten Objektes und der aktuelle Zeitpunkt gespeichert werden. Als Zeitgeber verwenden wir den eingebauten 200Hz-Timer. Er ist für unsere Zwecke mit 5 ms Auflösung genau genug.

Die Routine *play()* spielt das Band wieder ab. Es wird gewartet, bis der Zeitpunkt zum Spielen des nächsten Tones eintritt. Teile der Routine *ton()* können hier übernommen werden. Zur visuellen Unterstützung werden die entsprechenden Taste invertiert, als wären sie wirklich gedrückt worden. Wichtig ist, daß die Schleifen abbrechbar gestaltet werden. Im Beispiel erlauben beliebige Maustasten einen Abbruch.

Nach dem Compilieren steht die neue Version zur Verfügung. Was, Sie wollen noch mehr? Abspeichern und Laden, Editieren der Töne und Zeiten, veränderbare Parameter des Soundgenerators, Transponieren, Einstellen der Geschwindigkeit usw. Nun, dies sollte ja nur als Beispiel dienen. Weiter Ergänzungen müssen Sie schon selbst durchführen ...

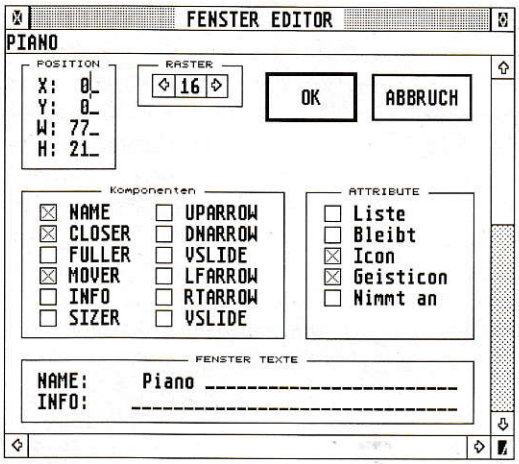
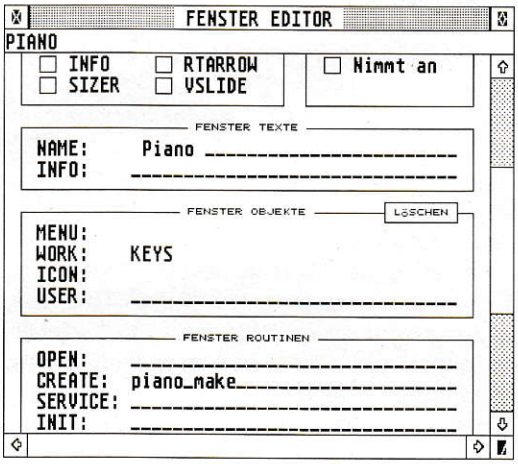


Bild 4a und ...



... Bild 4b: Der Fenster-Editor

Impressum
Sonderheft ST-Computer 4/92

Chefredakteur: Harald Egel (HE)

Stellvertr. Chefredakteur: Thomas Werner (tw)

Redaktion:
Harald Egel (HE)
Dieter Kühner (DK)
Christian Möller (CM)

Redaktionelle Mitarbeiter:
C.Borgmeier (CBO), Claus Brod (CB), Ingo Brümmer (IB), Derek dela Fuente (ddf), Thorsten Luhm (thl), U.Seimet (US), R.Tolksdorf (RT)

Autoren dieser Ausgabe:
Hans-H. Ackermann, Matthias Baldauf, M.G. Berberich, Jan Bolt, L. Canisius, C. Conrad, K. Elsbernd, C. Fihl, M. Fritze, G. Gehnen, S. Geier, E. Grah, D. Haun, U. Hax, R. Hodek, A. Hollermann, J. Jahn, H. Katzenmayer, A. Kohler, R. Kotzian, H. Lehmkuhl, A. Lötscher, M. Malich, U. Meumann, L. Preßler, T. Quellenberg, D. Rabich, H.-J. Richstein, K. Rindfrey, W. Sattler, G. Scheibler, G. Schmieder, M. Schumacher, U. Seimet, S. Simson, J. Starzynski, J. Stessun, P. Ubachs, U. Witte, M. Wunderli

Auslandskorrespondenz:
D.Dela Fuente (UK)

Redaktion: MAXON Computer GmbH
Postfach 59 69
Industriestr. 26
6236 Eschborn
Tel.: 0 61 96/48 18 14, FAX : 0 61 96/4 11 37

Verlag: Heim Fachverlag
Heidelberger Landstr. 194
6100 Darmstadt 13
Tel.: 0 61 51/5 60 57, FAX : 0 61 51/59 10 47 + 5 60 59

Verlagsleitung:
H.J.Heim

Anzeigenverkaufsleitung:
U.Heim

Anzeigenverkauf:
K.Sterna, H. Arbogast

Anzeigenpreise:
nach Preisliste Nr.7, gültig ab 2.1.92
ISSN 0932-0385

Grafische Gestaltung:
Manfred Zimmermann, Raoul Deubler

Titelgestaltung:
Manfred Zimmermann

Illustration:
Manfred Zimmermann

Produktion:
B.Kissner

Druck:
Frotscher Druck GmbH

Lektorat:
V.Pfeiffer

Bezugsmöglichkeiten:
ATARI-Fachhandel, Zeitschriftenhandel, Kauf- und Warenhäuser oder direkt beim Verlag

Einzelpreis: DM 14,-, ÖS 112,-, SFr 14,-, Lit. 10000,-
In den Preisen sind die gesetzliche MWSt. und die Zustellgebühren enthalten.

Manuskripteinsendungen:
Programmlistings, Bauanleitungen und Manuskripte werden von der Redaktion gerne angenommen. Sie müssen frei von Rechten Dritter sein. Mit seiner Einsendung gibt der Verfasser die Zustimmung zum Abdruck und der Vervielfältigung auf Datenträgern der MAXON Computer GmbH. Honorare nach Vereinbarung. Für unverlangt eingesandte Manuskripte wird keine Haftung übernommen.

DIGITAL DATA DEICKE, NORDRING 9, 3000 HANNOVER 1, TEL: 0511/6370 -54, -55

Nur solange Vorrat reicht: Original ATARI Diskettenlaufwerke(720KB) intern: DM 111,- / extern anschlußfertig: DM 166,-

Schon wieder kleinere Preise !

alle HD-Stationen ab sofort günstiger, Festplatten-Kits bis zu DM 100,- günstiger !

Mega STE - mit HD-Drive

Ein Mega STE ohne HD-Laufwerk ist wie ein Porsche ohne Räder (=unsere Meinung), deshalb gibt's bei uns die Mega STEs gleich mit HD-Laufwerk (720KB und 1.44MB, siehe unten). Preis des Mega STE mit 1 MByte RAM und HD-Laufwerk: 1444,-

Was Sie für's HD-Laufwerk mehr bezahlen, können Sie hier wieder sparen:

Aufpreise für mehr Speicher:
mit 2 MByte RAM +111,-
mit 4 MByte RAM +255,-

Festplatten (eingebaut):
mit 48 MB Platte +444,-
mit 52 MB Platte +633,-
mit 85 MB Platte +777,-
mit 105 MB Platte +888,-
mit 210 MB Platte +1444,-

Tip: SPAREN Sie DM 345,- gegenüber EZ-Preis bei Kauf mit Farb-Multiscan FMA 14-II

AT-Tastatur

für ST, STE und TT. Eine der besten Tastaturen, die Cherry G80/1000 gibt's jetzt anschlußfertig(!) für Ihren ATARI. Kein Löten oder IC-Tauschen ! Anschließen - fertig ! Dank neuem Prozessor auch noch schneller. Mehr dazu im Info: gleich kostenlos anfordern.

nur DM 249,-

HD-Diskettenstationen

HD-Diskettenstationen, die auch 1.44MB verarbeiten, sind heute das MUSS für einen modernen Computer. Die Gründe: 1.) doppelt so viel Speicherplatz pro Diskette; 2.) doppelt so schnelle Datenübertragung; 3.) IBM-Diskettenformate können gelesen werden (außer mit uraltem TOS); 4.) sehr günstiges Speichermedium !!! 5.) voll kompatibel zu 720KB Disketten (also normales Arbeiten wie bisher; 6.) sehr hochwertige Qualität (alle Laufwerke von TEAC !) zum günstigen Preis. Um die HD-Option zu nutzen, wird das HD-Modul benötigt.

Preissturz

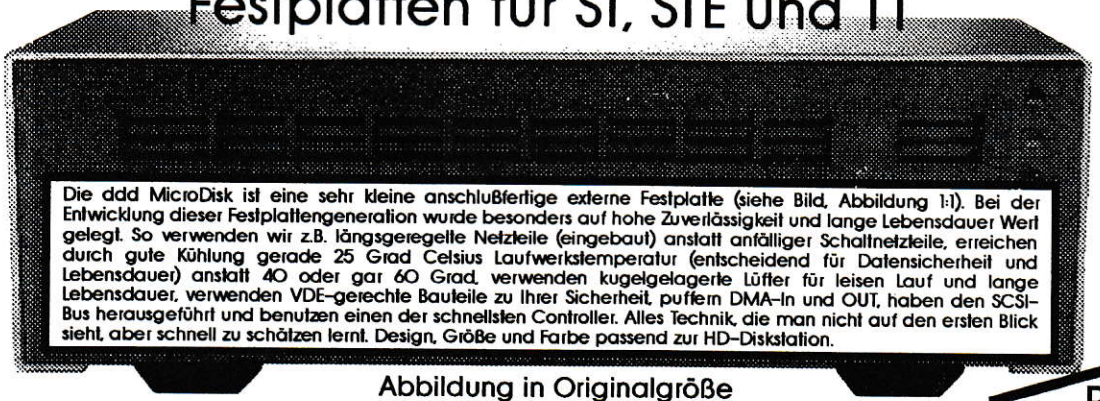
neue Preise

3.5" HD-Station zum Einbau incl. Anleitung	DM 129,-
3.5" wie vor mit ddd HD-Modul	DM 185,-
3.5" externe HD-Station anschlußfertig	DM 196,-
3.5" wie vor mit ddd HD-Modul	DM 255,-
5.25" HD-Station zum "Einbau" incl. Anleitung	DM 149,-
5.25" wie vor mit ddd HD-Modul	DM 199,-
5.25" externe HD-Station anschlußfertig	DM 222,-
5.25" wie vor mit ddd HD-Modul	DM 277,-

EXTRAS

dyn. Mouse für ST,STE u.TT	77,-
optische Mouse für ST,STE u.TT	111,-
Scanner 400 DPI, 105mm	393,-
AT-SPEED C16	422,-
Coprozessor für AT-SPEED C16	166,-
1 MByte SIMM für alle STE	88,-
Coprozessor für Mega STE	188,-
Coprozessor für Mega ST	299,-
Megafile 44 mit Medium	1333,-
Laserdrucker für ST,STE u.TT ab	1694,-
FMA 14-II Multiscan	1194,-
dto. zus.mit Mega STE	+849,-
Neu: SM 144 für ST	333,-
Einschaltverzögerung	49,-
Leiser Lüfter für Mega ST	39,-
Hypercache+, 16MHz	388,-
Lieferbar: WD 1772 02-02	55,-

Festplatten für ST, STE und TT



Die ddd MicroDisk ist eine sehr kleine anschlußfertige externe Festplatte (siehe Bild, Abbildung 1:1). Bei der Entwicklung dieser Festplattengeneration wurde besonders auf hohe Zuverlässigkeit und lange Lebensdauer Wert gelegt. So verwenden wir z.B. längsgeregelte Netzteile (eingebaut) anstatt anfälliger Schaltnetzteile, erreichen durch gute Kühlung gerade 25 Grad Celsius Laufwerkstemperatur (entscheidend für Datensicherheit und Lebensdauer) anstatt 40 oder gar 60 Grad, verwenden kugelgelagerte Lüfter für leisen Lauf und lange Lebensdauer, verwenden VDE-gerechte Bauteile zu Ihrer Sicherheit, puffern DMA-In und OUT, haben den SCSI-Bus herausgeführt und benutzen einen der schnellsten Controller. Alles Technik, die man nicht auf den ersten Blick sieht, aber schnell zu schätzen lernt. Design, Größe und Farbe passend zur HD-Diskstation.

Abbildung in Originalgröße

neuer Preissturz

Der Controller
Speziell für höchste Geschwindigkeit entwickelt, garantieren wir einen Interleave von 1 und erreichen Übertragungsraten bis über 1500 KByte/s. Integrierter Hardwareschreibschutz zur Sicherheit vor Viren (vergessen Sie Passwörter !). Bis zu 7 Festplatten anschließbar. Adressen von außen bzw. durch Software einstellbar (s.Software) Echtzeituhr (baugleich dem Mega ST) nachrüstbar.

Die Software
Der Treiber ist voll Atari AHDI 4.0 kompatibel. Neu: Durch Cache bis 512KB (einstellbar) um bis zu Faktor 3.4 schneller ! Neu: Softwaremäßige Unit-Adresseinstellung (52er u. 105er). Voll autobootfähig von jeder Partition. Jede MicroDisk wird komplett eingerichtet geliefert, also anschließen, einschalten und sofort arbeiten (wie mit Disketten, nur bis zu 50 mal schneller).

Die Laufwerke
Zum Einsatz kommen ausschließlich modernste 3.5" SCSI-Drives von Seagate und Quantum. Aber Achtung: Quantum ist nicht gleich Quantum ! Wir verwenden nur die Laufwerke der neuen LPS Serie mit 1" Bauhöhe aufgrund des geringeren Laufgeräusches und der höheren Geschwindigkeit. Alle Laufwerke haben Hardware-Autopark-Funktion, parken überflüssig.

Die Preise
ddd-MicroDisk 48
mit Seagate ST 157N-1
DM 794,- (555,-)

ddd-MicroDisk 52
mit Quantum LPS 52 S
DM 922,- (666,-)

ddd-MicroDisk 85
mit Seagate ST 1096N
DM 1055,- (794,-)

ddd-MicroDisk 105
mit Quantum LPS 105 S
DM 1222,- (944,-)

Kit-Preise in Klammer (Platte, Controller, Kabel, Software)

Jetzt umsteigen - NEU: Atari SM 144 (S/W, 14") ersetzt den kleinen SM 124. Unser Preis: ATARI SM 144 incl. Drehfuß: nur DM 333,-

Versand per NN, europaweit und Direktverkauf in Hannover

Öffnungszeiten: MO. - FR. von 10 - 18 Uhr durchgehend
Samstag und Sonntag geschlossen.

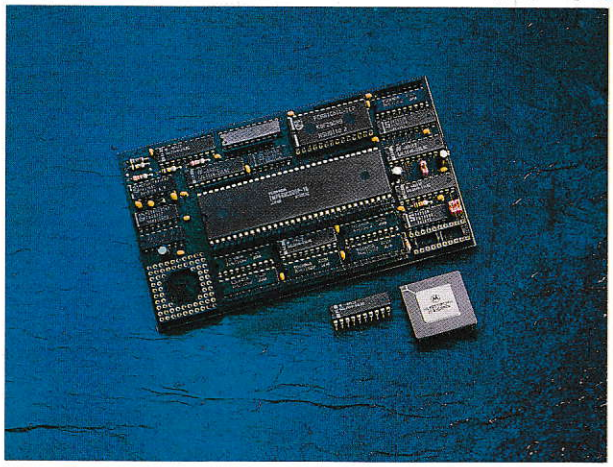
Es gelten unsere Geschäftsbedingungen

Rufen sie doch mal an

digital data deicke, einer der weltweit größten Anbieter im ST-Bereich rät: Achten Sie auf guten technischen Support und eine erreichbare Hotline. Es zahlt sich aus !

Heyer & Neumann CeBit Special

Zu langsam?



RAM, also Hauptspeicher kann ein Computer eigentlich garnicht genug haben. 4MB hat Atari für die Computer der ST Serie vorgesehen. Wollen Sie auch 4MB in Ihrem Atari ST Computer haben? Kein Problem! Mit der stromsparenden CMOS 4MB Speichererweiterungskarte IMEX4 kann jeder Atari ST Computer auf 4MB erweitert werden. Ein paar Features:

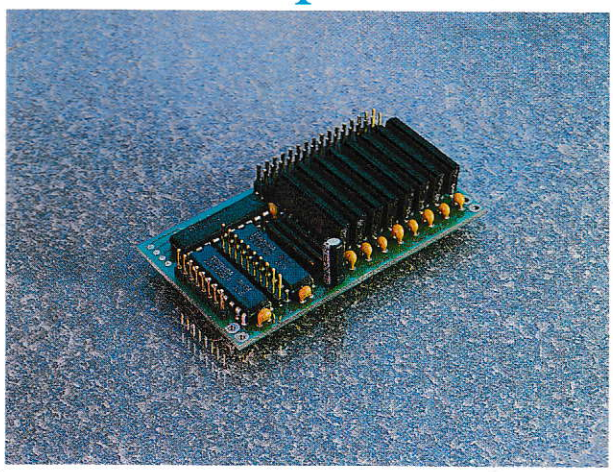
- deutsche Anleitung
- anschlußfertig, ready to go
- externer DMA und SCSI Bus
- interne Echtzeituhr
- thermogeregelter Lüfter
- inklusive aller Kabel
- inklusive Backup Software

HBS 240!

Sie benutzen Ihren Mega ST nicht nur zum spielen, sondern arbeiten ernsthaft mit ihm? Manchmal wünschen Sie sich, daß der Computer schneller wäre, oder wollen Sie sogar einen Floating-Point Coprozessor verwenden? Dann brauchen Sie keinen neuen Computer sondern einen professionellen Hardwarebeschleuniger. Der hier vorgestellte HBS 240 arbeitet mit 16MHz Taktfrequenz und hat einen 16KB Cache Speicher on board! Damit erreicht ein normaler Mega ST eine Beschleunigung von ca. 1.8 bis 2.0. Damit wir Ihr Mega ST dann genauso schnell wie ein neuer Mega STE. Sie wollen nicht die Katze im Sack, Sie wollen objektive Information? Dann fordern Sie einfach den Nachdruck eines Testberichts an, natürlich kostenlos.

HBS 240	**299.-**
Aufpreis für NVDI	**60.-**
Aufpreis für FPU	**200.-**

Mehr Speicher?



IMEX 4!

77x43mm winzig
12 Monate Garantie
Kompatibel!!!
modernste 4Mbit ZIP Rams
Insiderinformation:
Ein 1040ST benötigt im 5V Zweig ca. 1.7A. Nach dem Einbau einer IMEX4 sinkt diese Verlustleistung auf ca. 1.4A!!!

Einbauservice	**100.-**
IMEX4	**399.-**
IMEX3	**277.-**

HardDisk?

Kein ST muß länger ohne Harddisk sein! Wir bieten anschlußfertige Festplatten in jeder Größe und für jeden Geldbeutel. Ein paar allgemeine Features:

HD 52	**999.-**
HD 105	**1255.-**
HD 210	**1888.-**
HD 330	**3099.-**
HD 425	**3099.-**
HD 660	**4222.-**
HD 1200	**5555.-**
WP 44	**1255.-**
WP 88	**1755.-**

44-1200MB!

Ausgewählt servicefreundliche und erfahrene Fachhändler:

DATASOUND
Schillerpromenade 24
W-1000 Berlin 44
030/6228604

DIGIT
Mühlgasse 19
W-6903 Neckargemünd
06223/72095

UniCom SX GmbH
Eisenbahnstr. 93
O-7050 Leipzig
+37/0/41/65523

Shift GmbH
Kompagniestr. 13
W-2390 Flensburg
0461/22828

EDV Stiftung Grünau
Erlenstr. 73
CH-8805 Richterswil
+41/0/1/7848947

JOTKA Computing BV.
Vening Meinesz 1
NL-6717 AJ Ede
+31/0/8380/21675