ATARI ST INTERN

Band 2

Pauly

Systemprogrammierung

DISKETTE IM BUCH

DATA BECKER

Pauly

Atari ST Intern Bd. 2
Systemprogrammierung

DATA BECKER

1. Auflage 1989

ISBN 3-89011-324-9

DATA BECKER GmbH
Merowingerstr. 30
4000 Düsseldorf

Text verarbeitet mit Word 4.0, Microsoft
Ausgedruckt mit Hewlett Packard LaserJet II
Druck und Verarbeitung Mohndruck, Gütersloh

Wichtiger Hinweis:
Die in diesem Buch wiedergegebenen Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden.

Alle technischen Angaben und Programme in diesem Buch wurden vom Autor mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernommen werden kann. Für die Mitteilung eventueller Fehler ist der Autor jederzeit dankbar.

Vorwort

Intern Band 2 - was dürfen Sie von einem solchen Buch erwarten? Neue, bisher unbekannte Informationen über das Betriebssystem? Noch tiefergehende Beschreibungen der Hardware Ihres Ataris? Die Antwort ist nein. Denn alle Fakten, die Sie zur professionellen Programmierung des Rechners benötigen, sind bereits im "alten" Intern enthalten.

Dieser erste Intern-Band konnte jedoch nicht Nachschlagewerk und Lehrbuch in einem sein. Die Möglichkeiten eines Rechners wie des Atari ST sind so komplex, daß nicht nur Einsteiger, sondern auch bereits computererfahrene Aufsteiger Schwierigkeiten haben, den Rechner richtig zu nutzen, selbst mit einer so guten Systemdokumentation wie sie das Intern I bietet. Deshalb gibt es nun auch den zweiten Intern-Band. Er vermittelt Ihnen nicht nur den Einstieg in diese Materie, sondern wird Sie lange bei Ihrer Arbeit begleiten (man kann ihn nicht in ein paar Wochen durcharbeiten). Voraussetzung zum Verständnis ist allerdings das Intern I oder ein ähnliches Buch (der Data-Becker-Führer Atari ST reicht auch).

Fünf wichtige Themen werden behandelt: TOS-Programmierung, Grafik, Interrupts, Sound-Programmierung und GEM-Programme. Sie sollten die Lektüre mit dem ersten Kapitel beginnen; danach können Sie frei zwischen den Themen wählen. Weil man viele Dinge am besten verstehen kann, wenn sie in Betrieb sind, beinhaltet das Buch sehr viele Beispielprogramme. Bitte nutzen Sie die Gelegenheit, und machen Sie auch Ihre eigenen Experimente (kleine Veränderungen der Beispielprogramme genügen); denn was man selbst herausfindet, sieht man sofort ein und behält es viel besser als alles, was man fertig vorgesetzt bekommt.

Ich wünsche Ihnen nun viel Spaß beim Lesen und Erfolg bei der Programmierung Ihres Rechners!

Martin Pauly *Nettetal, im Februar 1989*

Wichtig!

Die Beispielprogramme dieses Buches, die Sie alle auf der beiliegenden Diskette finden, sind für folgende Interpreter und Compiler gedacht:

- GFA-BASIC Version 3.03 (Sollten Sie die Versionen 3.01 oder 3.02 besitzen, so können Sie diese bei der Firma GFA updaten. Die älteren Versionen laufen nicht immer fehlerfrei!)
- Omikron-BASIC Version 3.0 und Omikron-Compiler
- Megamax Laser C
- Profimat-Assembler

In der Regel ist jedoch die Verwendung anderer C-Compiler oder Assembler kein Problem.

Die große Zahl der Programme machte es leider erforderlich, eine doppelseitige Diskette zu verwenden. Dies scheint vertretbar, weil die meisten ST-Besitzer ohnehin ein doppelseitiges Diskettenlaufwerk besitzen. Jedes Kapitel (1 bis 5) hat einen eigenen Ordner auf der Diskette. Die GEM-Demoprogramme (Kapitel 5.11) sind in einem separaten Ordner untergebracht (DEMOS.511). Im Anhang E finden Sie eine Liste aller Programme.

Inhaltsverzeichnis

1. TOS - Das Betriebssystem des Atari ST

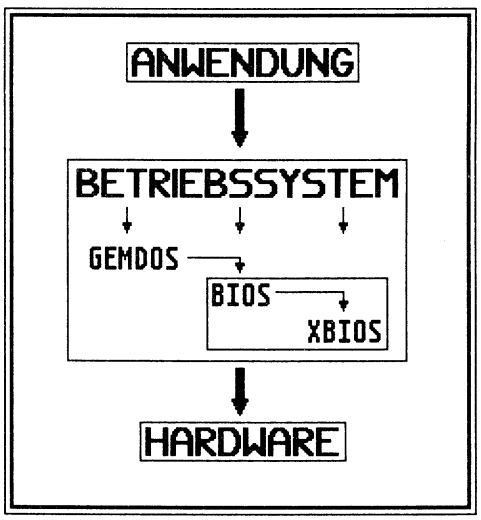
In einem Computerlexikon aus dem Jahre 1983 fand ich neulich eine Erklärung für die Abkürzung TOS: Tape Operating System, also ein Magnetband-Betriebssystem. Glücklicherweise sind die Zeiten, zu denen noch Kassettenrekorder an den Home-Computern hingen, vorbei. In unserem ST heißt TOS auch etwas anderes, nämlich Tramiel Operating System. Die beiden letzten Worte bedeuten wieder Betriebssystem, und Tramiel heißt der Chef der Firma Atari, der, bescheiden wie er ist, der Systemsoftware seines Rechners seinen eigenen Namen geben ließ.

TOS ist also das Betriebssystem des Atari ST. Es ist eine Sammlung von Routinen, die die Verbindung zwischen einem Anwenderprogramm und der Peripherie ermöglichen. Bei diesen Routinen gibt es, ähnlich wie bei Programmiersprachen, problemorientierte und maschinenorientierte Teile. Nehmen wir als Beispiel die Diskettenverwaltung: Ein Programm möchte Daten in eine Datei schreiben. Nun gibt es aber auf der Diskette zunächst einmal gar keine Dateien, sondern nur Tracks (Spuren) und Sektoren, vielleicht auch noch Diskettenseiten. Der systemnahe Teil des TOS muß also solche Sektoren schreiben und lesen können, während der problemorientierte Teil mit Hilfe der systemnahen Routinen Dateien verwaltet. Man könnte auch von zwei Ebenen des Systems sprechen: der eine Teil auf einem niedrigen und der andere auf einem hohen Level.

Diese Teile haben unterschiedliche Namen: Der problemorientierte Bereich heißt GEMDOS (hat nichts mit GEM zu tun) und der systemnahe Teil nennt sich BIOS (BASIC Input/Output System = allgemeines Ein-/Ausgabe-System). Dieses BIOS hat noch eine Unterabteilung namens XBIOS. Das X steht für extended, es ist also ein erweitertes BIOS. Das bedeutet lediglich, daß dieser Teil des BIOS die besonderen Fähigkeiten des Atari versorgt, während das normale BIOS nur für gängige Dinge zuständig ist, über die jeder Computer verfügen sollte.

Wenn nun eine Anwendung eine GEMDOS-Routine aufruft, um eine bestimmte problemorientierte Aktion durchzuführen, dann übersetzt GEMDOS diese Aktion in eine oder mehrere systemnahe Funktionen und gibt diese an BIOS oder XBIOS weiter, wo sie schließlich ausgeführt werden. GEMDOS hat folglich keinen Kontakt zur Hardware des Rechners (abgesehen vom Hauptspeicher). Bei Diskettenoperationen wird sogar

noch ein Zwischenschritt eingelegt: GEMDOS ruft BIOS, BIOS ruft XBIOS und erst hier wird die Hardware aktiviert. Folgendes Bild zeigt noch einmal den Zusammenhang:



Selbstverständlich darf sich eine Anwendung auch direkt an BIOS und XBIOS wenden. Sie kann theoretisch auch sofort die Hardware ansprechen, aber das ist nur im äußersten Notfall zu empfehlen (also dann, wenn die "offiziellen" Routinen eine unbedingt benötigte Funktion nicht anbieten).

1.1 Funktionsaufrufe

Nachdem wir diese allgemeinen Dinge geklärt haben, können wir in medias res gehen, also die Funktionsaufrufe unter die Lupe nehmen. Entscheiden Sie sich bitte für eine Programmiersprache, und lesen Sie da weiter, wo der entsprechende Aufruf beschrieben wird.

GFA-BASIC

In GFA-BASIC bedient man sich der Funktionen GEMDOS, BIOS und XBIOS. Da Funktionen stets einen Wert zurückgeben, müssen wir die Funktionsergebnisse einer Variablen zuweisen oder direkt verarbeiten:

```
A=Gemdos(...)
If Bios(...)<0 ...
Print Xbios(...)
```

Wird der Rückgabewert nicht benötigt, kann man auch schreiben: Void Gemdos(...)

Den Funktionen sind in Klammern die Funktionsnummer und, falls erforderlich, Parameter zu übergeben. Wenn nicht anders vereinbart, sind die angegebenen Parameter Worte (= 2 Bytes). Wollen Sie Langworte (= 4 Bytes) übergeben, so müssen Sie dem Wert oder der Variablen ein L für Langwort und einen Doppelpunkt voranstellen.

Omikron-BASIC

In Omikron-BASIC gibt es die Befehle GEMDOS, BIOS und XBIOS. Den Befehlen wird in Klammern folgender Ausdruck nachgestellt:

```
(Rückgabevariable,Funktionsnummer,Parameter1,Parameter2...)
```

Rückgabevariable und Parameter sind optional. Wenn Sie keine Rückgabevariable angeben möchten (d.h. wenn die Funktion keinen für Sie wichtigen Wert zurückgibt), so müssen Sie trotzdem das Komma zwischen der nicht vorhandenen Variablen und der Funktionsnummer angeben. Beachten Sie bitte, daß die Variable auch wirklich eine solche sein muß (keine Zahlen oder Ausdrücke). In Omikron-BASIC sind leider alle Parameter automatisch ein Wort groß. Wenn eine Funktion ein Langwort als Parameter erwartet, müssen Sie daher erst das high-word (die oberen 16 Bit) und dann das low-word (die unteren 16 Bit) des gewünschten Wertes übergeben. Das machen Sie am besten mit den BASIC-Funktionen HIGH und LOW. Beispiele dazu finden Sie weiter unten.

C

Jeder C-Compiler verfügt über ein Include-File, mit dem alle TOS-Funktionen über ihre Namen aufgerufen werden können. Sie müssen sich also keine Funktionsnummern mehr merken. Dieses Include-File heißt gewöhnlich OSBIND.H (Operating System Bindings). Es wird durch folgende Zeile am Anfang eines Programms geladen:

```
#include <osbind.h>
```

Es gibt aber auch in C die Funktionen gemdos(...), bios(...) und xbios(...). In Klammern werden Funktionsnummer und, wenn nötig, die Parameter übergeben. Wenn Sie von dieser Methode Gebrauch machen, sollten Sie jedoch im Programm die Funktionen als long-Funktionen definieren:

```
extern long gemdos();
extern long bios();
extern long xbios();
```

Zu sagen ist noch, daß die Funktion (wie in C üblich) auch als Befehl geschrieben werden kann. Also:

```
ergebnis = gemdos(...);
```

aber auch:

```
gemdos(...);
```

Assembler

Für die TOS-Aufrufe in Assembler werden drei Trap-Befehle verwendet:

```
trap #1     ; GEMDOS-Aufruf
trap #13    ; BIOS-Aufruf
trap #14    ; XBIOS-Aufruf
```

Den Funktionswert findet man nach dem Aufruf im Register D0. Die Funktionsnummer ist vor dem Trap-Befehl als Wort auf dem Stack abzulegen. Parameter können vor der Funktionsnummer ebenfalls auf den Stack gebracht werden, und zwar in umgekehrter Reihenfolge (verglichen mit C oder BASIC). Nach dem Trap muß der Programmierer eine Stack-Korrektur durchführen, das heißt: er muß so viele Bytes zum Stackpointer addieren, wie er vor dem Aufruf für die Übergabe von Parametern und Funktionsnummer benötigt hat. Ein Beispiel:

```
move.w  parameter1,-(sp)          ; -(sp) = auf den Stack
move.l  parameter2,-(sp)          ; Ein Wort und ein Langwort
move.w  #funktionsnummer,-(sp)
trap    #1                        ; #1 -> GEMDOS-Aufruf
addq.l  #8,sp                     ; Stackkorrektur
```

Wir haben 2 Worte und ein Langwort auf den Stack gelegt, was zusammen 8 Bytes benötigte. Diese 8 Bytes müssen wir durch den addq.l-Befehl kompensieren.

1.2 Zeichen-Ein und-Ausgabe über BIOS

Als erstes möchte ich Ihnen die Eingabe und Ausgabe von Zeichen vorstellen, weil dies sicher die am einfachsten anzuwendenden Systemroutinen sind.

Prinzipiell können Sie Zeichen auf zwei Ebenen ein- und ausgeben: über GEMDOS- und über BIOS-Funktionen. Beides hat seine Vor- und Nachteile: Während die BIOS-Funktionen schneller sind, verwaltet GEMDOS logische Peripheriegeräte, die beliebigen physikalischen Geräten zugeordnet werden können. Wenn Sie den Sinn dieses Vorgehens nicht verstehen, so seien Sie unbesorgt - etwas später gebe ich Ihnen ein paar anschauliche Beispiele dafür.

Beginnen wir mit den BIOS-Routinen. Wir benötigen folgende Routinen:

C-Aufruf	Funktions-Nummer	Beschreibung
Bconin(dev)	2	Holt ein Zeichen vom Gerät dev
Bconout(dev,zeichen)	3	Gibt zeichen auf Gerät dev aus
Bconstat(dev)	1	Liefert Status von Eingabegerät dev
Bcostat(dev)	8	Liefert Status von Ausgabegerät dev

Dev (Device = Gerät) kann folgende Werte annehmen:

Dev	Gerät	Ausgabe	Eingabe
0	Centronics-Schnittstelle	ja	ja
1	RS232-Schnittstelle	ja	ja
2	Tastatur und Bildschirm	ja	ja
3	Midi-Schnittstelle	ja	ja
4	Tastaturprozessor	ja	nein
5	Bildschirm (ASCII-Kanal)	ja	nein

Wohlgemerkt: Man kann über die Centronics-Schnittstelle Daten einlesen. Das ist natürlich unsinnig, solange ein Drucker an diesem Port hängt, weil der gar keine Daten zum Rechner schickt, aber man könnte hiermit z.B. eine schnelle parallele Schnittstelle realisieren, die Daten zwischen zwei ST's austauschen kann.

Der Unterschied zwischen den beiden mit "Bildschirm" bezeichneten Geräten (2 und 5) liegt übrigens darin, daß bei einer Ausgabe über Nummer 2 Steuerzeichen (Return, Zeilenvorschub, Tab und VT-52-Se-

quenzen) erkannt und ausgeführt werden. Deshalb kann dieses Gerät auch nicht alle Zeichen des Atari-Zeichensatzes ausgeben (z.B. keine LED-Ziffern). Dies kann nur der Kanal 5, der wiederum keine Steuerzeichen auswertet.

Das folgende Beispielprogramm nutzt nun den ASCII-Kanal, um den kompletten Zeichensatz des ST am Bildschirm darzustellen:

GFA-BASIC

```
'
' Ausgabe des gesamten Zeichensatzes mit Bconout
' GFA-BASIC       MP  07-10-88       FONTOUT.GFA
'
VOID BIOS(3,2,27)           ! Steuerzeichen mit verarbeiten: 2VOID
BIOS(3,2,ASC("E"))      ! ESC (27) + E --> Bildschirm löschen
'
FOR char%=0 TO 255
  IF char% MOD 16=0         ! nach jeweils 16 Zeichen eine
    VOID BIOS(3,2,13)       ! neue Zeile anfangen (13 = Carriage
    VOID BIOS(3,2,10)       ! Return, 10 = Line Feed;  Steuerzeichen!)
  ENDIF
  VOID BIOS(3,5,char%)      ! Eigentliches Zeichen ausgeben plus ein
  VOID BIOS(3,5,32)         ! Leerzeichen (5: keine Steuerzeichen)
NEXT char%
'
PRINT AT(2,20);"Der Zeichensatz des Atari ST"
END
```

Omikron-BASIC

```
'
'  Ausgabe des gesamten Zeichensatzes mit Bconout
'  Omikron-BASIC      MP 06-09-88      FONTOUT.BAS
'
BIOS (,3,2,27): BIOS (,3,2, ASC("E")): REM Escape E = Bildschirm löschen
'
FOR Char%=0 TO 255
  IF Char% MOD 16=0
    THEN BIOS (,3,2,13): REM 13 + 10 = Return + Zeilenvorschub
         BIOS (,3,2,10)
     ENDIF
   BIOS (,3,5,Char%): REM Funktion 3 (Bconout), Gerät 5 (ASCII-Kanal)
   BIOS (,3,5,32): REM Leerzeichen
 NEXT Char%
 '
 PRINT @(20,1);"Der Zeichensatz des Atari ST"
 END
```

C

```
/*******************************************************/
/*   Ausgabe des gesamten Zeichensatzes mit Bconout    */
/*   Megamax Laser C     MP  07-09-88     FONTOUT.C    */
/*******************************************************/

#include <osbind.h>     /* Definitionen für GEMDOS, BIOS und XBIOS */
#define  console 2
#define  ascii   5

int  i;

main()
{
  for (i=0; i<256; i++)
  {
    if (i % 16 == 0)            /* neue Zeile alle 16 Zeichen */
    {
      Bconout (console, 13);    /* Carriage Return */
      Bconout (console, 10);    /* Linefeed */
    }
    Bconout (ascii, i);         /* eigentliche Zeichenausgabe */
    Bconout (ascii, ' ');       /* Leerzeichen dazwischen */
  }

  printf ("\n\n\nDer Zeichensatz des Atari ST\n");

  Bconin (console);             /* Warten auf Tastendruck... */
}
```

Assembler

```
          ;
          ; Ausgabe des gesamten Zeichensatzes mit Bconout
          ; Assembler        MP  07-09-88        FONTOUT.Q
          ;
console   = 2
ascii     = 5

bconout   = 3
bconin    = 2

gemdos    = 1
bios      = 13

          TEXT

          clr.w     d3              ;Zähler für Zeichen
          move.w    #$ff,d4         ;Zähler für Schleife
```

```
loop:     move.b    d3,d5           ;Test auf Teilbar durch 16
          andi.b    #%11110000,d5
          cmp.b     d5,d3
          bne.s     no_crlf

          move.w    #13,-(sp)       ;Carriage Return
          move.w    #console,-(sp)  ;Ausgabegerät
          move.w    #bconout,-(sp)  ;Funktionsnummer
          trap      #bios
          addq.l    #6,sp

          move.w    #10,-(sp)       ;Linefeed
          move.w    #console,-(sp)
          move.w    #bconout,-(sp)
          trap      #bios
          addq.l    #6,sp

no_crlf:  move.w    d3,-(sp)        ;Auszugebendes Zeichen
          move.w    #ascii,-(sp)    ;Ausgabegerät ist ASCII-Kanal
          move.w    #bconout,-(sp)  ;Funktionsnummer
          trap      #bios
          addq.l    #6,sp

          move.w    #' ',-(sp)      ;Leerzeichen
          move.w    #ascii,-(sp)    ;Ausgabegerät
          move.w    #bconout,-(sp)  ;Funktionsnummer
          trap      #bios
          addq.l    #6,sp

          addq.w    #1,d3           ;nächstes Zeichen

          dbra      d4,loop

          move.w    #console,-(sp)  ;Warten auf Tastendruck
          move.w    #bconin,-(sp)
          trap      #bios
          addq.l    #4,sp

          clr.w     -(sp)
          trap      #gemdos

          END
```

Betrachten wir nun den Eingabe- bzw. Ausgabestatus, also die Funktionen Bconstat und Bcostat. Ein Eingabestatus von -1 bedeutet, daß mindestens ein Zeichen von einem Eingabegerät, z.B. der Tastatur, bereitsteht und mit Bcônin abgeholt werden kann. Ist der Status 0, so war-

tet kein Zeichen mehr. Entsprechend ist der Ausgabestatus -1, wenn das Ausgabegerät bereit ist, ein Zeichen auszugeben, und 0, wenn es nicht bereit ist.

Was können wir nun mit diesem Status anfangen? Nun, stellen Sie sich einmal vor, Sie möchten ein selbstlaufendes Demo-Programm schreiben, das solange läuft, bis es durch eine bestimmte Taste abgebrochen wird, z.B. durch die Leertaste. Dazu müssen Sie nur ab und zu den Eingabestatus der Konsole (Tastatur) abfragen. Ist der Status 0, führen Sie Ihre Demonstration fort. Ist er aber -1, dann hat der Anwender inzwischen eine Taste betätigt. Welche Taste er gedrückt hat, erfahren Sie aber erst durch die Funktion Bconin. War es die Leertaste, dann beenden Sie das Programm, ansonsten ignorieren Sie den Tastendruck und machen weiter.

Ein Beispiel soll das zeigen:

GFA-BASIC

```
'
' Selbstlaufende Demo (Abbruch durch Tastendruck)
' GFA-BASIC       MP  08-10-88       TASTDEMO.GFA
'
' Solange keine Taste gedrückt wurde...
'
WHILE BIOS(1,2)=0
  ausgabe("DEMONSTRATION. Taste = ENDE...          ")
  PAUSE 2
WEND
'
' Der Tastendruck, der den Abbruch verursachte, muß noch
' 'offiziell' abgeholt (= aus dem Tastaturpuffer entfernt)
' werden --> Bconin
'
VOID BIOS(2,2)
'
END
'
'
PROCEDURE ausgabe(a$)
  '
  ' Ausgabe über BIOS-Funktion Bconout
  '
  FOR i%=1 TO LEN(a$)
    VOID BIOS(3,2,ASC(MID$(a$,i%,1)))
  NEXT i%
  '
```

```
  ' (PRINT ginge natürlich auch, aber wir wollen
  ' schließlich mit BIOS-Routinen arbeiten...)
  '
RETURN
```

Omikron-BASIC

An dieser Stelle habe ich vielleicht eine kleine Enttäuschung für Sie: Omikron-BASIC erlaubt es oft nicht, diese BIOS-Funktion einzusetzen. Oft heißt, daß Sie es im Zweifelsfall am besten selbst ausprobieren. Mein Testprogramm brachte jedenfalls recht merkwürdige Ausgaben auf den Bildschirm. Auch die Bildschirm-Ausgabe bzw. Tastatur-Eingabe unter GEMDOS ist manchmal nicht möglich, wie Sie noch sehen werden. Hier müssen Sie auf BASIC-Befehle und -Funktionen zurückgreifen oder ganz einfach ausprobieren.

C

```
/*********************************************************/
/*   Selbstlaufende Demo (Abbruch durch Tastendruck)   */
/*   Megamax Laser C      MP 16-09-88     TASTDEMO.C   */
/*********************************************************/

#include <osbind.h>
#define  CONSOLE 2

int  i;

ausgabe (string)
char *string;
{
  while (*string != 0)
    Bconout (CONSOLE, *(string++));
}

main()
{
  while (!Bconstat (CONSOLE))  /* Ausführen, solange keine Taste */
                                          /* gedrückt wird */
  {
    ausgabe ("DEMONSTRATION. Taste = ENDE...         ");

    for (i=0; i<9999; i++)              /* Verzögerungsschleife */
      ;
  }

  Bconin (CONSOLE);          /* Das wartende Zeichen muß aus dem */
                             /* Tastaturpuffer geholt werden. */
}
```

Assembler

```
          ;
          ; Selbstlaufende Demo (Abbruch durch Tastendruck)
          ; Assembler         MP 22-09-88        TASTDEMO.Q
          ;
gemdos    = 1
BIOSc

BCONSTATc
BCONINc
BCoNoUT=

TEXT-
-
cLR1wD3-
-
LooP→LeaausgaBe
a3;stringaufbildschirmausgeben-*-*
outloop:  tst.b     (a3)            ;stringende erreicht?-*
          beq.s     outfin          ;dann zur warteschleife-*-*
          move.b    (a3)+,d3        ;Sonst Zeichen ausgeben
          move.w    d3,-(sp)        ;Als Wort erweitert auf Stack
          move.w    #2,-(sp)        ;2 für Konsole (Bildschirm)
          move.w    #bconout,-(sp)
          trap      #bios
          addq.l    #6,sp

          bra.s     outloop         ;Nächstes Zeichen...

outfin:   moveq.l   #-1,d0          ;Verzögerungsschleife
wait:     dbra      d0,wait

          move.w    #2,-(sp)        ;Inzwischen Taste gedrückt?
          move.w    #bconstat,-(sp)
          trap      #bios
          addq.l    #4,sp
          tst.w     d0
          beq.s     loop            ;Nein, dann weiter...

          move.w    #2,-(sp)        ;Sonst Zeichen aus Tastatur-
          move.w    #bconin,-(sp)   ;puffer holen...
          trap      #bios
          addq.l    #4,sp

          clr.w     -(sp)           ;... und Programm beenden
          trap      #gemdos

          DATA
```

```
ausgabe:  DC.b 'DEMONSTRATION. Taste = Ende...          ',0

          END
```

In unserem Beispielen haben wir auf eine beliebige Taste gewartet. Versuchen Sie einmal, ein Programm so zu ändern, daß nur bei einer Leertaste das Programm verlassen wird.

Auch für den Ausgabestatus gibt es sinnvolle Anwendungen,um zum Beispiel zu überprüfen, ob der Drucker angeschlossen ist und auf ON LINE steht:

GFA-BASIC

```
:
'
' Test ob Drucker empfangsbereit mit Bcostat
' GFA-BASIC      MP  07-10-88     ONLINE.GFA
'
IF BIOS(8,0)
  PRINT "Der Drucker ist bereit!"
ELSE
  PRINT "Der Drucker ist nicht bereit!"
ENDIF
END
```

Omikron-BASIC

```
'
' Test ob Drucker empfangsbereit mit Bcostat
' OMIKRON-BASIC    MP 08-09-88    ONLINE.BAS
'
BIOS (Status,8,0)
IF Status=-1
  THEN PRINT "Der Drucker ist bereit!"
  ELSE PRINT "Der Drucker ist nicht bereit!"
ENDIF
END
```

C

```
/*****************************************************/
/*   Test ob Drucker empfangsbereit (mit Bcostat)    */
/*   Megamax Laser C     MP 08-09-88     ONLINE C    */
/*****************************************************/

#include <osbind.h>
```

```
#define  drucker 0
#define  console 2

int  status;

main()
{
  status = Bcostat (drucker);

  if (status == -1)
    printf ("Der Drucker ist bereit!\n");
  else
    printf ("Der Drucker ist nicht bereit!\n");

  Bconin (console);     /* Warten auf Taste */
}
```

Assembler

```
          ;
          ; Test ob Drucker empfangsbereit mit Bcostat
          ; Assembler       MP 08-09-88       ONLINE.Q
          ;

drucker   = 0
console   = 2

bconin    = 2
bcostat   = 8
cconws    = 9

gemdos    = 1
bios      = 13

          TEXT

          move.w    #drucker,-(sp)  ;Ist Drucker bereit?
          move.w    #bcostat,-(sp)
          trap      #bios
          addq.l    #4,sp

          tst.b     d0              ;Dann ist d0 = -1
          beq.s     not_ok          ;Nicht bereit, dann d0 = 0ready:
          pea       ok_text         ;Meldung ausgeben
          bra.s     cont

not_ok:   pea       err_text
cont:     move.w    #cconws,-(sp)
          trap      #gemdos
          addq.l    #6,sp
```

```
        move.w    #console,-(sp)  ;Warten auf Tastendruck
        move.w    #bconin,-(sp)
        trap      #bios
        addq.l    #4,sp

        clr.w     -(sp)           ;Programm beenden
        trap      #gemdos

        DATA

ok_text:  DC.b "Der Drucker ist bereit!",0

err_text: DC.b "Der Drucker ist nicht bereit!",0

        END
```

1.3 Ein- und Ausgabe mit GEMDOS

Mit den GEMDOS-Funktionen zur Zeichen-Ein-und-Ausgabe kann man im Prinzip genau dasselbe machen wie mit den BIOS-Routinen. Und um es gleich zu sagen: Die GEMDOS-Funktionen sind um einiges langsamer als ihre Kollegen vom BIOS. Sie werden aber bald sehen, daß es sich oft lohnen kann, den Geschwindigkeitnachteil in Kauf zu nehmen.

Als auffälligster Unterschied wäre zu bemerken, daß es nicht mehr die Funktion zur Ein- oder Ausgabe gibt, sondern daß jedes Gerät dafür eigene Routinen erhalten hat. Betrachten wir einmal die Funktionen, die die Konsole (Bildschirm und Tastatur) bedienen:

C-Aufruf	Funktions-Nummer	Beschreibung
Cconin()	1	Wartet auf ein Zeichen von der Tastatur und gibt Scan- und ASCII-Code zurück.
Cconout(zeichen)	2	Gibt Zeichen auf Bildschirm aus.
Crawcin()	7	Wie Cconin(), gibt Zeichen aber nicht auf Bildschirm aus.
Cconws(string)	9	Gibt einen ganzen String auf dem Bildschirm aus. Steuerzeichen werden ausgewertet. Stringende: Nullbyte
Cconis()	11 dez.	-1 = Zeichen von Tastatur verfügbar oder 0 = kein Zeichen verfügbar.

Bemerkung:
Die BASIC- und Assembler-Programmierer müssen beachten, daß ein String, der mit der Funktion Cconws(string) ausgegeben werden soll, mit

einem Nullbyte abgeschlossen sein muß. Das gilt auch für alle folgenden TOS-Funktionen, denen eine Zeichenkette als Parameter zu übergeben ist. In BASIC schreiben Sie daher z.B.

```
A$="Das ist ein Satz."+CHR$(0)
```

In Assembler sähe es so aus:

```
string:  DC.B  'Das ist ein Satz',0
```

Der C-Compiler hängt die Null automatisch an eine Stringkonstante. In C und Assembler steht der Name eines Strings für seine Startadresse. In BASIC muß diese mit VARPTR und ggf. SEGPTR (Omikron) bestimmt werden. Noch eleganter geht es in Omikron-BASIC mit der MEMORY-Funktion (bitte mal im Handbuch nachschauen). Beispiele zu beiden Verfahren finden Sie in zahlreichen Listings.

Übrigens werten auch die GEMDOS-Ausgabefunktionen Steuerzeichen aus. Oft werde ich sie für den Zeilenvorschub einsetzen, manchmal aber auch für die VT-52-Kommandos. Eine Liste dieser Kommandos finden Sie im Intern Band 1. Erkennen können Sie sie am Dezimalcode 27 (Escape) oder, in C-Programmen, am Octalcode "\33" (= 27 dezimal) innerhalb eines Strings.

Zurück zu unseren GEMDOS-Ausgabe-Funktionen. Vielleicht schauen Sie sich einmal meine Beispielprogramme an, die die Anwendung der Routinen verdeutlichen. Das erste gibt Zeichen ein bzw. aus, während das zweite einen String auf den Bildschirm bringt:

GFA-BASIC

```
'
' Ein- und Ausgabe von Zeichen unter GEMDOS
' GFA-BASIC      MP  08-10-88      CHROUT.GFA
'
PRINT "Ausgabe von Scan- und ASCII-Codes (Ende = SPACE)"
PRINT
'
REPEAT
  '
  ' Cconin aufrufen (wartet auf Taste und liefert Scan-
  ' und ASCII-Code der gedrückten Taste). Das Zeichen wird
  ' dabei gleich auf dem Bildschirm ausgegeben
  '
  code%=GEMDOS(1)
```

```
  '
  PRINT ,"ASCII = ";code% AND 255,"Scan = ";SHR(code%,16)
  '
UNTIL (code% AND 255)=32
END



'
' String-Ausgabe über die GEMDOS-Funktion Cconws
' GFA-BASIC       MP  08-10-88       STROUT.GFA
'
esc$=CHR$(27)
'
a$=esc$+"pINVERSE SCHRIFT"+esc$+"q ist mit GEMDOS kein Problem!"'
' Ausgabe von a$ mit Cconws:
'
VOID GEMDOS(9,L:VARPTR(a$))
'
' Warten auf Tastendruck (Crawcin):
'
VOID GEMDOS(7)
'
END
```

Omikron-BASIC

Wie bereits erwähnt, sind sämtliche oben aufgeführten GEMDOS-Funktionen in Omikron-BASIC leider oft nicht einzusetzen. So erging es mir auch bei dem Versuch, die beiden Demo-Programme für Omikron-BASIC umzusetzen.

C

```
/************************************************/
/*   Ein- und Ausgabe von Zeichen unter GEMDOS   */
/*   Megamax Laser C    MP 09-09-88   CHROUT.C   */
/************************************************/

#include <osbind.h>

char  zeichen;
int   scancode;
int   ascii;
long  funktionswert;

main()
{
```

```
  printf ("Ausgabe von Scan- und ASCII-Codes (Ende = SPACE)\n\n");

  do
  {
    funktionswert = Cconin();
    ascii = (int) funktionswert;
    zeichen = (char) funktionswert && 255;  /* Ergebnis aufteilen in */
    scancode = funktionswert >> 16;         /*  ASCII- und Scancode  */

    Cconout (zeichen);

    printf ("\t ASCII = %d \t Scan=%d\n", ascii,
                scancode);
  }
  while (ascii != 32);
}


/*****************************************************/
/*   String-Ausgabe über die GEMDOS-Funktion Cconws   */
/*   Megamax Laser C     MP  09-09-88     STROUT.C   */
/*****************************************************/

#include <osbind.h>

char  string[255];

main()
{
  strcpy (string, "\33pINVERSE SCHRIFT\33q ");      /* ESC p = Revers */
  strcat (string, "ist mit GEMDOS kein Problem!");  /* ESC q = Normal */

  Cconws (string);          /* Ausgabe des vorbereiteten Textes */
  Cconin ();                /* Warten auf Tastendruck */
}
```

Assembler

```
          ;
          ; Ein- und Ausgabe von Zeichen unter GEMDOS
          ; Assembler      MP 09-09-88       CHROUT.Q
          ;

gemdos    = 1

crawcin   = 7
cconout   = 2

          ; Das Programm läßt Sie einen Text am
```

```
        ; Bildschirm eingeben.   Ende: RETURN

        TEXT

loop:   move.w   #crawcin,-(sp)   ;Zeichen holen
        trap     #gemdos          ;(wird nicht am Bildschirm
        addq.l   #2,sp            ;ausgegeben)

        cmpi.b   #13,d0           ;Return gedrückt?
        beq.s    quitloop

        move.w   d0,-(sp)         ;nicht, dann Zeichen ausgeben...
        move.w   #cconout,-(sp)
        trap     #gemdos
        addq.l   #4,sp

        bra.s    loop             ;... und neues holen

quitloop: clr.w  -(sp)            ;Programm beenden
        trap     #gemdos

        END
```

Und last but not least:

```
        ;
        ;  String-Ausgabe über die GEMDOS-Funktion Cconws
        ;  Assembler       MP  09-09-88         STROUT.Q
        ;

gemdos  = 1

cconin  = 1
cconws  = 9

        TEXT

        pea      ausgabe          ;Funktion zur String-Ausgabe
        move.w   #cconws,-(sp)    ;(String muß mit Nullbyte
        trap     #gemdos          ;abgeschlossen sein)
        addq.l   #6,sp

        move.w   #cconin,-(sp)    ;Warten auf Tastendruck...
        trap     #gemdos
        addq.l   #2,sp

        clr.w    -(sp)            ;Programmende
        trap     #gemdos
```

```
        DATA

ausgabe: DC.b 27,'pINVERSE SCHRIFT',27,'q'
        DC.b ' ist mit GEMDOS kein Problem!',0

        END
```

Ein ganz wichtiger Unterschied zwischen der GEMDOS- und der BIOS-Zeichen-Ausgabe ist bisher nicht erwähnt worden: GEMDOS arbeitet mit logischen Geräten, BIOS mit physikalischen. Das dürfen Sie sich so vorstellen: Wenn ein Programm das BIOS aufruft, um ein Zeichen am Bildschirm auszugeben, dann kommt es auch zum Bildschirm. Wendet sich die Applikation dagegen ans GEMDOS, dann geht das Zeichen an die sogenannte Standard-Ausgabe-Einheit. Diese Einheit ist zwar gewöhnlich auch der Bildschirm, doch kann der logische Standard-Kanal auch auf andere physikalische Geräte geschaltet werden. Der dazu nötige Verwaltungsaufwand erklärt auch, warum die GEMDOS-Ausgaben langsamer vonstatten gehen. Die Anwendung werde ich allerdings erst etwas später mit einem Beispiel erläutern, wenn Sie die nötigen Vorkenntnisse des nächsten Abschnitts hinter sich gebracht haben.

1.4 Dateien

1.4.1 Lesen und Schreiben

Jeder weiß es: Dateien dienen dazu, Daten beliebiger Art auf einem nicht-flüchtigen Massenspeicher festzuhalten. Wie können wir als Anwenderprogramm nun solche Dateien benutzen?

Zunächst dürfte klar sein, daß die Datei einen Namen haben muß, der den üblichen Beschränkungen unterliegt (8 Zeichen, danach optional Punkt und drei Zeichen Extension). Zu diesem Namen gehört, wenn sie sich nicht im aktuellen Unterverzeichnis (Ordner) befindet, die Laufwerksbezeichnung und der Pfad. Beispiel: B:\ORDNER.FOL\DATEI

Wenn Sie einen Namen für die Datei gefunden haben, dann können Sie die Datei öffnen, d.h. zur Bearbeitung anmelden. Sie müssen dazu erst einmal entscheiden, diese Datei neu anlegen möchten oder ob Sie nur eine schon bestehende Datei zum Bearbeiten der gespeicherten Daten öffnen möchten. Ersteres besorgt die Funktion Fcreate, letzteres Fopen. Gemeinsam ist beiden Funktionen, daß sie als Parameter den Dateinamen erwarten und als Funktionswert eine Zahl zurückgeben. Unterschiedlich ist der zweite Parameter, der neben dem Dateinamen übergeben wird.

Übrigens: Alle TOS-Funktionen, die mit Dateien zu tun haben, gehören zur Gruppe der GEMDOS-Routinen, d.h. zur höchsten Systemroutinen-Klasse.

C-Aufruf	Nummer	Funktions-Beschreibung
Fcreate (filename, attribut)	$3C	Legt neue Datei an
Fopen (filename, modus)	$3D	Öffnet bestehende Datei

Die zurückgegebene Zahl kann zwei Bedeutungen haben: Ist sie negativ, dann gab es irgendwo Probleme (keine Diskette eingelegt, bei Fopen wurde die Datei nicht gefunden etc.). Ist sie positiv, dann nennt man die Zahl das File-Handle. Dieses File-Handle ist ab sofort ein Synonym für den Dateinamen und wird bei allen folgenden Zugriffen auf diese Datei angegeben, denn mit Zahlen kann GEMDOS viel besser hantieren als mit ganzen Datei- oder sogar Pfadnamen.

Zurück zu Fcreate und Fopen: Jede der Funktionen verlangt noch einen Parameter, der von der Funktion abhängt. Bei Fcreate gibt dieser bestimmte Eigenschaften der Datei an, der Parameter heißt deshalb Datei-Attribut:

0 "Normale" Datei;

1 Datei erhält nach dem Schließen den Status read-only (nurlesbar).

2 Datei erscheint nicht im Inhaltsverzeichnis, kann aber normal bearbeitet werden (versteckte Datei).

Fopen hingegen erwartet ein sogenanntes Modus-Wort. Dieses bestimmt das Zugriffsrecht auf die Datei:

0 Datei kann nur gelesen werden.

1 Datei kann nur beschrieben werden.

2 Datei kann gelesen und beschrieben werden.

Nehmen wir an, Sie haben eine Datei mit Fcreate neu eröffnet und wollen nun Daten in diese Datei schreiben. Dazu dient die Funktion Fwrite:

C-Aufruf	Nummer	Funktions-Beschreibung
Fwrite (handle, anz, start)	$40	Speichert anz Bytes ab start in der Datei handle.

Mit Fwrite können Sie also einen beliebigen Speicherbereich auf einer Diskette oder Festplatte sichern. Deshalb sollten Sie vor dem Aufruf auch dafür sorgen, daß die Daten, die Sie abspeichern möchten, sich an einem Stück im Speicher befinden. Wenn nicht, dann können Sie Fwrite auch mehrfach aufrufen - es wird dann jedesmal am Ende der Datei weitergeschrieben. Das dauert natürlich länger, als wenn Sie alles in einem Rutsch erledigen lassen.

Das letzte, was Sie noch tun müssen, ist, die Datei wieder zu schließen. Damit stellen Sie "Ihr" File-Handle wieder der Allgemeinheit zur Verfügung. Außerdem wird dabei die Dateilänge im Inhaltsverzeichnis aktualisiert. Erst wenn eine neu angelegte Datei geschlossen wurde, kann sie später problemlos zur Bearbeitung geöffnet werden. Das Schließen der Datei besorgt für uns die GEMDOS-Funktion Fclose:

C-Aufruf	Nummer	Funktions-Beschreibung
Fclose (handle)	$3e	Schließt eine Datei ordnungsgemäß.

Mit diesen Vorgaben können wir nun ein Programm erstellen, das einen kurzen Text in eine Datei schreibt, den Sie sich anschließend im Desktop mit einem Doppelklick anzeigen lassen können ("Diese Datei kann nur angezeigt..."). Die Datei soll den bezeichnenden Namen READ.ME tragen und eine Nur-Lesen-Datei sein. Das Programm sieht so aus:

GFA-BASIC

```
'
' Beispielprogramm für Dateien unter GEMDOS
' GFA-BASIC      MP 08-10-88      FILES.GFA
'
zeile$="Das ist der Text, der in die Datei kommt."
filename$="READ.ME"+CHR$(0)
nurlesen%=1
'
' Datei mit Fcreate eröffnen:
'
handle%=GEMDOS(&H3C,L:VARPTR(filename$),nurlesen%)
'
IF handle%<0
  PRINT "Fehler beim Öffnen der Datei!"
ELSE
  '
  ' Zeile$ in Datei schreiben
  '
  VOID GEMDOS(&H40,handle%,L:LEN(zeile$),L:VARPTR(zeile$))
  '
```

```
  ' Datei schließen
  '
  VOID GEMDOS(&H3E,handle%)
ENDIF
'
END
```

Omikron-BASIC

```
'
' Beispielprogramm für Dateien unter GEMDOS
' OMIKRON-BASIC    MP 16-09-88    FILES.BAS
'
Filename$="READ.ME"+ CHR$(0)
Zeile$="Das ist der Text, der in die Datei kommt."
'
' Adresse der STRINGs ausrechnen:
Nameptr= LPEEK( VARPTR(Filename$))+ LPEEK( SEGPTR +28)
Zptr= LPEEK( VARPTR(Zeile$))+ LPEEK( SEGPTR +28)
'
' Datei anlegen:
GEMDOS (Handle,$3C, HIGH(Nameptr), LOW(Nameptr),1)
IF Handle<0
THEN PRINT "Fehler beim Öffnen der Datei!"
     REPEAT : UNTIL INKEY$ <>""
ELSE ' Text in Datei schreiben
     Anz= LEN(Zeile$)
     GEMDOS (,$40,Handle, HIGH(Anz), LOW(Anz), HIGH(Zptr), LOW(Zptr))
     ' Datei schließen
     GEMDOS (,$3E,Handle)
ENDIF
'
END
```

C

```
/*************************************************/
/*   Beispielprogramm für Dateien unter GEMDOS   */
/*   Megamax Laser C    MP 15-09-88    FILES.C   */
/*************************************************/

#define  NUR_LESEN 1
#include <osbind.h>

int  handle;
char *zeile = {"Das ist der Text, der in die Datei kommt."};

main()
{
  handle = Fcreate ("READ.ME", NUR_LESEN);
```

```
  if (handle < 0)
    printf ("Fehler beim Öffnen der Datei!\n");
  else
  {
    Fwrite (handle, (long) strlen (zeile), zeile);
    Fclose (handle);
  }
}
```

Assembler

```
          ;
          ;  Beispielprogramm für Dateien unter GEMDOS
          ;  Assembler      MP  16-09-88       FILES.Q
          ;

gemdos    = 1

crawcin   = 7
cconws    = 9
fcreate   = $3c
fclose    = $3e
fwrite    = $40

          TEXT

          move.w    #1,-(sp)          ;Nur-Lesen-Attribut
          pea       filename
          move.w    #fcreate,-(sp)    ;Datei anlegen
          trap      #gemdos
          addq.l    #8,sp

          tst.w     d0                ;d0 negativ?
          bmi.s     error             ;dann Fehlermeldung
          move.w    d0,handle         ;sonst als Handle merken

          pea       anfang            ;Textzeile schreiben
          move.l    #schluss-anfang,-(sp)         ;Länge
          move.w    d0,-(sp)          ;File-Handle
          move.w    #fwrite,-(sp)
          trap      #gemdos
          adda.l    #12,sp

          move.w    handle,-(sp)      ;Datei schließen
          move.w    #fclose,-(sp)
          trap      #gemdos
          addq.l    #4,sp

ende:     clr.w     -(sp)             ;Programmende
```

```
          trap      #gemdos

error:    pea       errtext          ;Fehlermeldung ausgeben
          move.w    #cconws,-(sp)
          trap      #gemdos
          addq.l    #6,sp

          move.w    #crawcin,-(sp)   ;Warten auf Taste
          trap      #gemdos
          addq.l    #2,sp

          bra.s     ende

          DATA

filename: DC.b 'READ.ME',0

anfang:   DC.b 'Das ist der Text, der in die Datei kommt.'
schluss:

errtext:  DC.b 'Fehler beim Öffnen der Datei',13,10,0

          BSS

handle:   DS.w 1

          END
```

Nun ist es zwar ganz nett, sich einen Text vom Desktop aus anzeigen zu lassen, doch sollen die Daten auch wieder in eigenen Programmen eingelesen werden können. Abgesehen davon, daß eine solche Datei statt mit Fcreate natürlich mit Fopen geöffnet werden muß, benötigen wir auch noch das Gegenstück zu Fwrite. Diese Funktion, die Daten aus einer Datei wieder zurück in den Hauptspeicher des Rechners holt, heißt Fread:

C-Aufruf	Nummer	Funktions-Beschreibung
Fread (handle, anz, start)	$3f	Liest anz Bytes aus Datei in Speicher (ab Start).

Auch hierzu möchte ich Ihnen eigentlich ein kleines Beispiel geben, nämlich die Ausgabe eines Textes aus einer Datei - genau das Gegenteil von dem, was unser letztes Programm gemacht hat. Was ist zu tun? Wir lesen die ganze Datei in den Speicher und geben sie mit der GEMDOS-Funktion Cconws auf dem Bildschirm aus. Das Problem dabei liegt nun darin, das GEMDOS schon vor dem Lesen aufs Byte genau wissen möchte, wie viele Zeichen es beschaffen soll (Parameter anz bei Fread).

Im Klartext heißt das, daß wir die Größe der Datei herausfinden müssen, bevor wir sie laden können. Wie das geht, wird im nächsten Abschnitt erklärt. Das Beispiel zum Lesen aus einer Datei folgt dann sofort nach!

1.4.2 DTA - Disk Transfer Address

Vorweg ein Hinweis: Im folgenden Abschnitt kommen eine Menge Informationen auf Sie zu, die - scheinbar - unabhängig voneinander sind. Leider sind diese Funktionen nur im Zusammenhang zu verstehen, also lesen Sie bei Unklarheiten alles Folgende am besten zweimal.

Oft kommt es vor, daß Sie das Inhaltsverzeichnis eines Laufwerks benötigen, sei es, um es auf dem Drucker auszugeben oder um eine neue File-Selector-Box (Datei-Auswahl-Fenster) zu programmieren.

Nehmen wir an, Sie brauchen eine Liste aller C-Programme. Diese Dateien enden bekanntlich mit der Extension .C. Unser Betriebssystem kann nun alle Dateinamen suchen, die auf .C enden. Dazu benötigt es lediglich eine sogenannte Namens-Maske. Eine solche Maske haben Sie bestimmt schon einmal gesehen; sie enthält meist Sternchen (*) oder auch Fragezeichen. GEMDOS legt bei Suche die von Ihnen vorgegebene Maske über jeden Dateinamen, den es im Inhaltsverzeichnis findet, und überprüft, ob die Maske auf diesen Namen paßt. Wenn ja, dann wird der Name als gefunden gemeldet; wenn nicht, so wird weitergesucht.

Bleibt zu klären, wann eine Maske auf einen Dateinamen paßt. Der einfachste Fall ist der, daß Maske und Name identisch sind. Dann paßt die Maske selbstverständlich auf den Namen. Allerdings ist einzusehen, daß der Befehl "Suche alle Dateien, die ASSEMBLER.PRG heißen!" nicht sehr sinnvoll erscheint. Was wir brauchen, ist eine allgemeinere Maske, die auf mehrere Dateinamen zutreffen kann.

Diesem Zweck dienen die sogenannten Wildcards, auch Joker genannt. So besagt ein Fragezeichen (?) in der Maske, daß an der gleichen Stelle im Dateinamen ein beliebiger Buchstabe stehen darf. Der Asterisk (*) geht noch weiter: Er sagt dem System, daß ab der Position, an der er in der Maske zu finden ist, beliebig viele (also auch gar keine) Zeichen stehen dürfen. Dabei ist allerdings zu beachten, daß ein Asterisk im maximal achtstelligen Dateinamen nicht auch für die dreistellige Extension gilt. Die allgemeinste Maske, die es gibt, lautet demnach *.* und paßt auf alle nur denkbaren Dateinamen.

So, jetzt haben wir also eine Maske. Bleiben wir beim Beispiel der C-Programme von vorhin, so lautet sie *.C. Diese Maske übergeben wir an die GEMDOS-Funktion Fsfirst (steht für File-Funktion Search First), die bereitwillig das ganze Inhaltsverzeichnis durchsuchen wird, bis sie einen Dateinamen gefunden hat, der auf C endet. Der Rückgabewert der Funktion ist entweder Null, dann konnte ein Name gefunden werden, oder er ist negativ, dann war die Diskette defekt oder der Name ganz einfach nicht vorhanden.

Ihr berechtigter Einwand wird nun lauten: Was bringt mir das, wenn ich nur weiß, daß auf der Diskette ein C-Programm ist? Ich wollte doch den Namen erfragen. Seien Sie unbesorgt - den vollständigen Dateinamen (also ohne * und ?) hat Fsfirst schon herausgefunden und sicher deponiert. Wo,das können Sie selbst bestimmen, indem Sie - Sie erinnern sich an die Überschrift dieses Abschnitts - eine Disk Transfer Address festlegen. Womit sich der Kreis meiner Erklärungen schließt.

Die DTA ist der Beginn eines 44 Bytes großen Speicherbereichs, in dem Sie wichtige Angaben über die letzte mit Sfirst gefundene Datei vorfinden. Dieser Bereich ist wie folgt organisiert:

Offset	Größe	Bedeutung
0	21 Bytes	reserviert, keine Bedeutung.
21	1 Byte	Datei-Attribut (nur-lesen, ...)
22	2 Bytes	Uhrzeit
24	2 Bytes	Datum
26	4 Bytes	Dateilänge in Bytes
30	variabel	vollständiger Name mit Extension (mit Nullbyte abgeschlossen)

Das ist doch schon einiges an Informationen! Am Rande: Uhrzeit und Datum liegen in einem speziellen Format vor, das im Intern I beschrieben steht (suchen Sie dazu am besten die Funktionen zum Setzen/Lesen von Datum und Zeit).

Nun sollte unser Inhaltsverzeichnis natürlich nicht nur aus dem ersten Dateinamen bestehen, sondern aus allen, auf die die vorgegebene Maske paßt. Eine nochmalige Anwendung der Funktion Fsfirst würde wieder den gleichen, bereits gefundenen Namen ergeben (wie der Name Search First schon sagt). Zum Glück gibt es da noch die Funktion Fsnext (steht für Search Next = Suche nächsten Namen). Fsnext hält sich an die gleiche Maske, die Sie bei Fsfirst angegeben haben. Das bedeutet, daß einem Fsnext-Aufruf ein Fsfirst vorangehen muß. Fsnext sucht also den nächsten Dateinamen im Directory und legt Informationen darüber

ebenfalls im DTA-Puffer ab. Sie können Fsnext so oft aufrufen, bis es einen negativen Funktionswert liefert. Dann wurden bereits alle Dateinamen gefunden.

Was bisher noch nicht erwähnt wurde: Fsfirst benötigt nicht nur eine Maske, sondern auch einen Attribut-Wert als Parameter. Die Funktion sucht dann automatisch nur solche Dateien, die der Attribut-Wert angibt. Folgende Werte sind möglich:

0 Normale Datei.

1 Nur-Lesen-Datei.

2 Versteckte Datei.

8 Name der Diskette (eigentlich nur sinnvoll bei einer Maske von *.*).

16 Unterverzeichnis (Ordner).

Das ist so zu verstehen, daß z.B. ein Attribut-Wert von 0 alle Dateinamen findet, die normal sind oder Nur-Lesen-Status haben. Die 1 findet alle versteckten und nicht versteckten Dateien, die nicht auf Nur-Lesen geschaltet sind, und eine 2 schließlich findet alle Dateien. Wird als Attribut aber 8 oder 16 angegeben, so werden nur die dadurch ausdrücklich gewünschten Informationen (also Name der Diskette oder der Ordner) geliefert.

Abschließend noch einmal beide Funktionen in der Übersicht:

C-Aufruf	Nummer	Funktions-Beschreibung
Fsfirst (name, attribut)	$4e	Suche ersten Dateinamen.
Fsnext	$4f	Suche nächsten Dateinamen.

Das letzte, was Sie noch wissen müssen, ist die Festlegung der DTA,:Sie müssen dem Betriebssystem sagen, ab welcher Adresse es die Daten über gefundene Dateien ablegen soll. Die dazu bestimmte GEMDOS-Funktion heißt Fsetdta:

C-Aufruf	Nummer	Funktions-Beschreibung
Fsetdta (adresse)	$1a	Setzt Startadresse des DTA-Puffers für Fsfirst und Fsnext.

Nach all diesen Erläuterungen wird es Zeit für ein Beispielprogramm, das Licht in eventuell vorhandenes Dunkel bringt. Zunächst möchte ich das

Lesen einer Datei nachholen; Sie erinnern sich sicher an unser Problem im vorigen Abschnitt, die Länge einer Datei zu bestimmen. Mit unserem jetzigen Wissen ist das kein besonderes Problem:

GFA-BASIC

In diesem Programm werden zum ersten Mal die Funktionen MALLOC und MFREE verwendet. Sie dienen dazu, einen Speicherbereich bestimmter Größe, die hinter MALLOC in Klammern angegeben wird, zu reservieren und dessen Adresse zu ermitteln. In diesem Bereich können wir uns dann beliebig austoben. Es ist also dem MEMORY-Befehl des Omikron-BASIC ähnlich. Mit MFREE müssen wir diesen Bereich wieder zurückgeben, wenn er nicht mehr gebraucht wird. MALLOC und MFREE sind übrigens direkte Systemaufrufe des GEMDOS, nur sparen Sie sich das Schreiben der Funktionsnummer. Zum besseren Verständnis lesen Sie bitte das Kapitel 1.5 über die Speicherverwaltung.

```
'
' Ermitteln der Dateilänge und Anzeigen der Datei
' GFA-BASIC        MP  07-10-88      SHOWFILE.GFA
'
' Speicherbereich für DTA-Puffer reservieren:
'
dta%=MALLOC(44)
'
' Speicherbereich für eigentlichen Text:
'
puffer%=MALLOC(1000)
'
filename$="READ.ME"+CHR$(0)   ! CHR$(0) als Stringende-Kennungnurlesen%=0
'
'
' DTA festlegen:
'
VOID GEMDOS(&H1A,L:dta%)
'
' Fsfirst aufrufen für Dateilänge:
'
IF GEMDOS(&H4E,L:VARPTR(filename$),nurlesen%)<0
  PRINT "Datei wurde nicht gefunden!"
ELSE
  lang%=LPEEK(dta%+26)   ! Länge in Bytes
  '
  ' Datei öffnen, lesen und schließen:
  '
  handle%=GEMDOS(&H3D,L:VARPTR(filename$),0)
  VOID GEMDOS(&H3F,handle%,L:lang%,L:puffer%)
  VOID GEMDOS(&H3E,handle%)
```

```
  '
  ' Ausgabe der Daten:
  '
  VOID GEMDOS(9,L:puffer%)
  '
ENDIF
'
' Warten auf Tastendruck:
'
VOID GEMDOS(7)
'
CLS
'
' Speicherbereiche wieder freigeben
'
VOID MFREE(dta%)
VOID MFREE(puffer%)
'
END
```

Omikron-BASIC

```
 '
 ' Ermitteln der Dateilänge und Anzeigen der Datei
 ' OMIKRON-BASIC      MP  21-09-88    SHOWFILE.BAS
 '
 ' Speicherbereiche reservieren:
 Dta= MEMORY(44)' für DTA-Puffer
 Puffer= MEMORY(1000)' und für den eigentlichen Text
 Filename= MEMORY("READ.ME")' als Dateiname
 '
 ' Löschen des VT-52 Bildschirms
 BIOS (,3,2,27)' Escape
 BIOS (,3,2, ASC("E"))' + E = Schirm löschen
 '
 ' Festlegen der DTA:
 GEMDOS (,$1A, HIGH(Dta), LOW(Dta))
 '
 ' Fsfirst anwenden, um Dateilänge zu ermitteln:
 GEMDOS (Back%,$4E, HIGH(Filename), LOW(Filename),Nurlesen%)
 IF Back%<0
   THEN PRINT "Datei wurde nicht gefunden!"
   ELSE Lang= LPEEK(Dta+26)' Länge in Bytes
        ' öffnen, lesen und schließen
        GEMDOS (Handle%,$3D, HIGH(Filename), LOW(Filename),0)
        GEMDOS (,$3F,Handle%, HIGH(Lang), LOW(Lang),...
...HIGH(Puffer), LOW(Puffer))
        GEMDOS (,$3E,Handle%)
        '
        ' Ausgabe der Daten:
```

```
        GEMDOS (,9, HIGH(Puffer), LOW(Puffer))
    ENDIF
 GEMDOS (,7)' Warten auf Tastendruck
 CLS
 END
```

C

```
/*********************************************************/
/*   Ermitteln der Dateilänge und Anzeigen der Datei   */
/*   Megamax Laser C      MP 20-09-88     SHOWFILE.C   */
/*********************************************************/

#include <osbind.h>
#define  NUR_LESEN 1

struct  infoblock                       /* Ein struct bietet sich als */
{       short int reserviert[21];      /* Datentyp für den DTA-Puffer */
        short int attribut;                                  /* an */
        int       uhrzeit;
        int       datum;
        long      groesse;         /* Für uns wichtig: Größe in Bytes */
        char      name[14];
}   dta_puffer;

int  handle;
char puffer[1000];                /* Hierhin werden die Daten gelesen */

main()
{
  Fsetdta (&dta_puffer);   /* Startadresse des DTA-Puffers festlegen */

  if (Fsfirst ("READ.ME", NUR_LESEN) < 0)          /* Wir müssen die */
    printf ("Datei wurde nicht gefunden!\n");  /* Dateilänge heraus- */
  else                                                  /* finden */
  {
    handle = Fopen ("READ.ME", 0);     /* Datei zum (0=) Lesen öffnen */
    Fread (handle, dta_puffer.groesse, puffer);        /* Daten lesen */
    Fclose (handle);

    Cconws (puffer);          /* Alle Daten auf einen Schlag ausgeben */
  }

  Cconin ();                                        /* Warten auf Taste */
}
```

Assembler

```
          ;
          ;  Ermitteln der Dateilänge und Anzeigen der Datei
          ;  Assembler        MP  12-09-88        SHOWFILE.Q
          ;

nurlesen  = 1

gemdos    = 1

crawcin   = 7
cconws    = 9
fsetdta   = $1a
fopen     = $3d
fclose    = $3e
fread     = $3f
fsfirst   = $4e

          TEXT

          pea       dta_puf          ;Startadresse DTA-Puffer festlegen
          move.w    #fsetdta,-(sp)
          trap      #gemdos
          addq.l    #6,sp

          move.w    #nurlesen,-(sp)  ;Fsfirst-Aufruf (normale und
          pea       filename         ;nur lesen-Dateien suchen)
          move.w    #fsfirst,-(sp)
          trap      #gemdos
          addq.l    #8,sp

          tst.w     d0               ;Fehler?
          bmi.s     error            ;Ja, dann bitte melden...
          clr.w     -(sp)            ;Datei zum Lesen öffnen
          pea       filename
          move.w    #fopen,-(sp)
          trap      #gemdos
          addq.l    #8,sp
          move.w    d0,handle

          pea       puffer           ;Datei komplett einladen
          move.l    groesse,-(sp)    ;Länge ist von Fsfirst bekannt
          move.w    handle,-(sp)
          move.w    #fread,-(sp)
          trap      #gemdos
          adda.l    #12,sp

          move.w    handle,-(sp)     ;Datei schließen
          move.w    #fclose,-(sp)
          trap      #gemdos
```

```
          addq.l    #4,sp

          pea       puffer          ;Text am Bildschirm ausgeben...
          bra.s     print_on

error:    pea       err_text        ;... oder Fehlermeldung
print_on: move.w    #cconws,-(sp)
          trap      #gemdos
          addq.l    #6,sp

          move.w    #crawcin,-(sp)  ;Warten auf Tastendruck...
          trap      #gemdos
          addq.l    #2,sp

          clr.w     -(sp)           ;und Ende des Programms
          trap      #gemdos

          DATA

err_text: DC.b 'Datei wurde nicht gefunden!',0
filename: DC.b 'READ.ME',0

          BSS

handle:   DS.w 1

dta_puf:  DS.b 21
attribut: DS.b 1
zeit:     DS.w 1
datum:    DS.w 1
groesse:  DS.l 1
name:     DS.b 14

puffer:   DS.b 1000

          END
```

Ein zweites Beispiel soll die Verwendung der Funktion Fsnext verdeutlichen. Es handelt sich dabei um eine Ausgabe des Inhaltsverzeichnisses einer Diskette in Laufwerk A. Dabei gehen wir so vor, daß erst einmal Fsfirst mit der Maske *.* und einer Null als Attributwert (für normale Dateien) aufgerufen und der so (hoffentlich) gefundene Dateiname ausgegeben wird. Anschließend wird in einer Schleife solange Fsnext aufgerufen und der jeweils erhaltene Name angezeigt, bis der Rückgabewert dieser Funktion negativ ist, sprich: keine weiteren Dateien vorhanden sind.

GFA-BASIC

```
'
' Anzeigen des aktuellen Inhaltsverzeichnisses
' GFA-BASIC         MP  07-10-88       DIR.GFA
'
' Speicherbereich für DTA reservieren:
'
dta%=MALLOC(44)
'
maske$="*.*"+CHR$(0)                    ! Diese Maske findet alle'
! Dateien
' DTA einstellen:
'
VOID GEMDOS(&H1A,L:dta%)
'
' Aufruf von Fsfirst:
'
IF GEMDOS(&H4E,L:VARPTR(maske$),1)<0    ! 1: Normale und Nur-Lesen-
  PRINT "Keine Dateien gefunden!"       !    Dateien suchen
ELSE
  REPEAT
    VOID GEMDOS(9,L:dta%+30)            ! Dateinamen ausgeben
    PRINT                               ! Neue Zeile
    '
    ' Fsnext aufrufen (gleichzeitig Abbruchbedingung):
    '
  UNTIL GEMDOS(&H4F)<0
ENDIF
'
VOID GEMDOS(7)                          ! Auf Taste warten
VOID MFREE(dta%)                        ! Speicher wieder freigeben
'
END
```

Omikron-BASIC

```
 '
 ' Anzeigen des aktuellen Inhaltsverzeichnisses
 ' OMIKRON-BASIC      MP  21-09-88      DIR.BAS
 '
 ' Speicherbereiche reservieren:
 Dta= MEMORY(44)' für DTA-Puffer
 Nameptr=Dta+30' Offset für Dateiname: 30 Bytes
 Maske= MEMORY("*.*")' als Suchmaske
 Crlf= MEMORY( CHR$(13)+ CHR$(10))' für Zeilenvorschub
 '
 ' Löschen des VT-52 Bildschirms
 BIOS (,3,2,27)' Escape
 BIOS (,3,2, ASC("E"))' + E = Schirm löschen
```

```
'
' Festlegen der DTA:
GEMDOS (,$1A, HIGH(Dta), LOW(Dta))
'
' Fsfirst aufrufen:
GEMDOS (Back%,$4E, HIGH(Maske), LOW(Maske),Nurlesen%)
IF Back%<0
  THEN PRINT "Keine Dateien gefunden!"
  ELSE ' Ersten gefundenen Dateinamen ausgeben
       GEMDOS (,9, HIGH(Nameptr), LOW(Nameptr))
       GEMDOS (,9, HIGH(Crlf), LOW(Crlf))
       '
       ' Solange weitersuchen, bis Fsnext<0:
       WHILE 1' Scheinbare Endlosschleife
         GEMDOS (Back%,$4F)
         IF Back%<0 THEN EXIT
         GEMDOS (,9, HIGH(Nameptr), LOW(Nameptr))
         GEMDOS (,9, HIGH(Crlf), LOW(Crlf))
       WEND
ENDIF
GEMDOS (,7)
CLS
END
```

C

```
/*****************************************************/
/*   Ausgabe des aktuellen Inhaltsverzeichnisses   */
/*   Megamax Laser C     MP 20-09-88       DIR.C   */
/*****************************************************/

#include <osbind.h>
#define  NUR_LESEN 1

struct  infoblock                        /* Wie gehabt - ein struct */
{       short int reserviert[21];             /* für den DTA-Puffer */
        short int attribut;
        int       uhrzeit;
        int       datum;
        long      groesse;
        char      name[14];
}   dta_puffer;

char *cr_lf = {"\15\12\0"};            /* Sequenz für Zeilenvorschub */

ausgabe()
{
  Cconws (dta_puffer.name);           /* Ausgabe des Dateinamens und */
  Cconws (cr_lf);                             /* des Zeilenvorschubs */
}
```

```
main()
{
  Fsetdta (&dta_puffer);   /* Startadresse des DTA-Puffers festlegen */

  if (Fsfirst ("*.*", NUR_LESEN) < 0)      /* normale und nur-lesen- */
    Cconws ("Keine Dateien gefunden!");           /* Dateien suchen */
  else
  {                                                /* Namen gefunden? */
    ausgabe();                                  /* Dann diesen ausgeben... */

    while (Fsnext() == 0)          /* ... und alle folgenden (solange */
      ausgabe();                              /* welche gefunden werden) */
  }

  Cconin ();                                        /* Warten auf Taste */
}
```

Assembler

```
          ;
          ;  Ausgabe des aktuellen Inhaltsverzeichnisses
          ;  Assembler        MP  12-09-88         DIR.Q
          ;

nurlesen  = 1

gemdos    = 1

crawcin   = 7
cconws    = 9
fsetdta   = $1a
fsfirst   = $4e
fsnext    = $4f

          TEXT

          pea       dta_puf          ;Startadresse DTA-Puffer festlegen
          move.w    #fsetdta,-(sp)
          trap      #gemdos
          addq.l    #6,sp

          move.w    #nurlesen,-(sp)  ;Fsfirst-Aufruf (normale und
          pea       maske            ;nur-lesen-Dateien suchen)
          move.w    #fsfirst,-(sp)
          trap      #gemdos
          addq.l    #8,sp

          tst.w     d0               ;Fehler?
          bmi.s     error            ;Ja, dann gibt's keine Dateien
```

```
          bsr.s     ausgabe          ;Sonst den Dateinamen ausgeben

loop:     move.w    #fsnext,-(sp)    ;Nach weiteren Namen suchen
          trap      #gemdos
          addq.l    #2,sp
          tst.w     d0               ;Fehler?
          bmi.s     taste            ;Dann abbrechen
          bsr.s     ausgabe          ;Sonst Namen ausgeben...
          bra.s     loop             ;... und weitermachen

error:    pea       err_text         ;Meldung ausgeben
          move.w    #cconws,-(sp)
          trap      #gemdos
          addq.l    #6,sp

taste:    move.w    #crawcin,-(sp)   ;Warten auf Taste...
          trap      #gemdos
          addq.l    #2,sp

          clr.w     -(sp)            ;Programmende
          trap      #gemdos

ausgabe:  pea       name             ;Ausgabe des gefundenen Namens
          move.w    #cconws,-(sp)
          trap      #gemdos
          addq.l    #6,sp

          pea       crlf             ;Zeilenvorschub ausgeben
          move.w    #cconws,-(sp)
          trap      #gemdos
          addq.l    #6,sp
          rts

          DATA

err_text: DC.b 'Keine Dateien gefunden!',0
maske:    DC.b '*.*',0
crlf:     DC.b 13,10,0

          BSS

dta_puf:  DS.b 21
attribut: DS.b 1
zeit:     DS.w 1
datum:    DS.w 1
groesse:  DS.l 1
name:     DS.b 14

          END
```

1.5 Speicherverwaltung

An dieser Stelle stecke ich in einer Art Zwickmühle. Einerseits kann ich Ihnen zu diesem Thema jetzt noch keine sinnvollen Beispielprogramme geben; andererseits werden Sie im folgenden einige Informationen darüber benötigen. Um diese nicht stückchenweise liefern zu müssen, schiebe ich sie hier ein.

Wenn in einem Computer immer nur ein Programm zur gleichen Zeit läuft, dann kann es sich eigentlich selbst aussuchen, welchen Speicherplatz es für sich selbst und für seine Daten beansprucht - zu Konflikten mit anderen Programmen kann es nicht kommen, weil die ja gar nicht vorhanden sind. In unserem Atari sieht die Sache allerdings anders aus: Kontrollfeld, Spooler, RAM-Disk und was der schönen Dinge mehr sind, wollen alle zusammen mit einer Anwendung, z.B. einer Textverarbeitung, unter einem Dach im Rechner wohnen. Würde jetzt jedes Programm seine Daten irgendwo in die erstbesten Speicherzellen schreiben, dann können Sie sich vorstellen, wie schnell es zu Zusammenstößen kommt. Das gilt natürlich nicht nur für die Daten der einzelnen Programme; auch die Programme selbst dürfen nicht miteinander und mit Daten eines anderen Programms kollidieren.

Aus diesem Grund "gehört" der gesamte Speicher des Ataris dem TOS, also dem Betriebssystem. Etwas genauer: Es ist GEMDOS, ein Teil von TOS, der sich um die Speicherverwaltung kümmert. Weil GEMDOS aber mit dem vielen Speicher gar nichts anfangen kann, gibt es gerne auf Anfrage Teile davon an Programme zurück. GEMDOS merkt sich jeweils, welche Speicherbereiche noch frei und welche schon vergeben sind, um kein Byte zweimal zu verteilen.

Wenn ein Programm vom Desktop aus gestartet wird, dann geschieht folgendes: GEMDOS sucht den größten noch freien Speicherbereich und kennzeichnet ihn als belegt. Dieses als belegt kennzeichnen heißt übrigens allokieren. Dann lädt es die Programmdatei in diesen Speicherbereich und gibt die Kontrolle an das so geladene Programm ab. GEMDOS erledigt dabei gleich noch ein paar andere Kleinigkeiten, doch die interessieren uns im Moment nicht.

Die erste Amtshandlung des Programms sollte nun sein, den eigenen Speicherbedarf zu ermitteln und, wenn dieser geringer als der von GEMDOS allokierte Bereich ist, letzteren zu verkleinern, um soviel wie möglich Platz für die anderen Programme freizuhalten.

Stellt das Programm nun später fest, daß es doch mehr Platz benötigt, als ursprünglich angenommen wurde, dann kann es sich an GEMDOS wenden und um einen zusätzlichen Bereich bitten. Auch ein solcher erst später zugeteilter Bereich kann, wenn er nicht mehr gebraucht wird, zurückgegeben werden.

Dieses Prinzip funktioniert natürlich nur dann, wenn sich auch alle anderen Programme an die Spielregeln halten und nur soviel Platz in Anspruch nehmen, wie sie unbedingt benötigen. Das gilt übrigens auch, wenn Sie Programme nachladen möchten. GEMDOS reserviert nämlich für das nachzuladende (genau wie für das nachladende) Programm den größten Speicherbereich, der gerade frei ist. Und wenn keiner mehr frei ist, kann folglich auch nichts nachgeladen werden.

Vielleicht werden Sie jetzt fragen, warum wir uns denn bisher in den Beispielprogrammen nicht um die Speicherverwaltung kümmern mußten. Die Antwort ist einfach: Es war nicht nötig, d.h. unsere Programme liefen alleine im Speicher (jedenfalls hätte man Accessories nicht aufrufen können), wir wollten keine anderen Programme nachladen und auch nicht nachträglich Speicherplatz anfordern. Das wird sich aber in den folgenden Kapiteln ändern, so daß ich Ihnen hier für jede Programmiersprache kurz das Wichtigste erkläre. Unabhängig davon sollten Sie sich im ersten Band des Intern einmal kurz über die Befehle Malloc, Mfree und Mshrink (auch SETBLOCK genannt) informieren.

GFA-BASIC

Wenn Sie den Interpreter laden, so "behält" dieser fast den ganzen ihm zugeteilten Speicher, um möglichst große Programme und viele Variablen zu erlauben. Sie können aber durch die Anweisung RESERVE einen Teil des Speichers zurückgeben. Dieser zurückgegebene Bereich steht dann dem Interpreter nicht mehr direkt zur Verfügung. Er kann aber mit GEMDOS-Funktionen vom Programm verwendet werden.RESERVE ohne Parameter setzt die Ausgangseinstellung, d.h. 16 KByte des Speichers stehen GEMDOS zur Verfügung, der Rest dem Interpreter. Wird eine negative Zahl übergeben, so wird der BASIC-Arbeitsspeicher um den Betrag dieser Zahl vermindert und der freie GEMDOS-Speicher erhöht. Schließlich kann mit einem positiven Parameter hinter RESERVE die Größe des BASIC-Arbeitsspeichers eingestellt werden.

Omikron-BASIC

Die Ausgangssituation ist die gleiche wie in GFA-BASIC. Allerdings wird die Größe des GEMDOS-Speichers über den CLEAR-Befehl be-

stimmt. CLEAR anzahl besagt, daß der GEMDOS-Speicher ab sofort anzahl Bytes groß sein soll. Nach dem Laden des Interpreters sind 64 KByte für GEMDOS bestimmt.

C

In C sieht die Sache anders aus: Wird ein C-Programm geladen, so steht der Speicherbedarf für den Programmcode und die Variablen bereits fest. Deshalb wird automatisch der gesamte nicht benötigte Speicher an GEMDOS zurückgegeben. Automatisch heißt, daß der C-Compiler den Code zur Berechnung des Speicherbedarfs generiert.

Assembler

Hier geht es im Prinzip so wie auch in C, nur wird die Berechnung nicht automatisch vorgenommen. Um dies von Hand zu machen, müssen Sie wissen, daß beim Start eines Programms das zweite Langwort auf dem Stapel (über 4(sp) zu erreichen) auf die sogenannte Basepage zeigt. Diese enthält unter anderem die Größen des Programms und dessen Daten. Den genauen Aufbau der Basepage entnehmen Sie bitte dem Intern Band 1.

Die Berechnung selbst ist ein Standard-Verfahren, das in fast jedem Programm zu Anfang verwendet wird. Bei dieser Gelegenheit wird meist auch gleich ein größerer Prozessor-Stapel (Stack) eingerichtet. Zusammen kann das z.B. so aussehen (mit 4 KByte Stack):

```
gemdos  =  1
mshrink = $4a

     movea.l   4(sp),a5          ;4(sp) ist Start der Basepage
     move.l    12(a5),d0         ;Länge des Programmcodes
     add.l     20(a5),d0         ;+ Länge des Data-Segments
     add.l     28(a5),d0         ;+ Länge des BSS-Segments
     addi.l    #$1100,d0         ;+ Basepage (256 Bytes) + Stack (4 KB)

     move.l    d0,d1             ;Länge plus
     add.l     a5,d1             ;Startadresse
     andi.l    #-2,d1            ;(gerundet)
     movea.l   d1,sp             ;ergibt Stackpointer

     move.l    d0,-(sp)          ;Länge des benötigten Speichers
     move.l    a5,-(sp)          ;Startadresse des Bereichs
     clr.w     -(sp)             ;dummy-Byte ohne Bedeutung
     move.w    #mshrink,-(sp)
     trap      #gemdos
     adda.l    #12,sp
```

Mehr ist zu diesem Thema im Moment nicht zu sagen. Und weil in einem Demo-Programm die Speicherverwaltung ganz alleine wenig Sinn macht, bekommen Sie hierzu auch kein eigenes Beispielprogramm.

1.6 Umlenken der Ein-und Ausgabe

Erinnern Sie sich noch an das, was ich über den Unterschied zwischen den GEMDOS- und BIOS-Routinen zur Bildschirm-Ausgabe bzw. Tastatur-Eingabe gesagt habe? Da war von logischen und physikalischen Geräten die Rede und von der Möglichkeit, physikalische Geräte logischen zuzuordnen. Darum wird es jetzt gehen.

Genauso wie jede geöffnete Datei ein File-Handle besitzt, so haben auch Tastatur, Bildschirm, Drucker und die serielle Schnittstelle jeweils ein Handle. Weil diese Handles festgelegt, das heißt Geräten fest zugeordnet sind und nicht erst durch einen Fopen-Befehl erfragt werden müssen, spricht man von Standard-Handles. Alles andere (also Handles von Diskettendateien) heißt demnach Non-Standard-Handle.

Folgende Standard-Handles kennt der Atari:

0	Konsol-Eingabe	(Tastatur)
1	Konsol-Ausgabe	(Bildschirm)
2	RS232-Schnittstelle	(Modem)
3	Centronics-Schnittstelle	(Drucker)

Die Funktion Cconout (Zeichen) zur Ausgabe eines Zeichens kann man sich also als Fwrite-Aufruf vorstellen, der in die Datei mit dem Handle 1 (also Bildschirm-Ausgabe) ein einzelnes Byte schreibt, nämlich das Zeichen, das Cconout als Parameter übergeben wird - wobei einsichtig sein sollte, daß für den Programmierer der Cconout-Aufruf viel bequemer ist als der Fwrite-Aufruf. Intern geht aber auch Cconout über eine Art Fwrite, und zwar mit dem Standard-Handle 1.

Übrigens gibt es noch die Standard-Handles 4 und 5, die allerdings im Atari keine Funktion haben. Sie wurden von MS-DOS übernommen, wo sie implementiert sind. Wenn Sie MS-DOS kennen (den internen Aufbau, also nicht nur Kommandos wie DIR oder COPY), so werden Sie schon bemerkt haben, daß das GEMDOS mit dem BDOS von MS-DOS verwandt ist. Wichtig für Sie ist aber noch, daß alle Handles ab 6 Non-Standard-Handles sind und vom TOS bei Bedarf vergeben werden.

Doch jetzt zu unserem eigentlichen Problem. Denken wir uns ein Programm, das eine String-Eingabe vornehmen soll. Dabei wissen wir aber noch nicht, von welchem Eingabegerät der String später einmal kommen soll (Tastatur, RS232 oder Datei sind möglich). Deshalb schreiben wir das Eingabe-Unterprogramm einfach so, als ob die Zeichen von der Tastatur kämen, weil dies die wohl einfachste Methode ist (bequeme Cconin-Aufrufe). Bei Bedarf wollen wir dann die Tastatur-Eingabe (Standard-Handle 0) auf z.B. eine geöffnete Datei umleiten. Alle Zeichen-Eingabefunktionen des GEMDOS warten dann nicht mehr auf einen Tastendruck, sondern bedienen sich aus der Datei!

Die GEMDOS-Funktion, die solche Wunderdinge leistet, heißt Fforce. Sie hat die Funktionsnummer $46. Der genaue Aufruf lautet:

```
Fforce (Standard-Handle, Non-Standard-Handle)
```

Nach diesem Aufruf gehen alle Ein- bzw. Ausgaben, die bisher über das Standard-Handle erfolgten, über das Non-Standard-Handle. Das betrifft auch Funktionen wie Cconin oder Cconout; die einzige Ausnahme betrifft die Funktionen zum Überprüfen auf Sende- und Empfangsbereitschaft, also z.B. zur Überprüfung, ob ein Zeichen im Tastaturpuffer bereitliegt (Cconis); diese Funktionen können nicht umgeleitet werden.

Zurück zu unserer Aufgabe: Wir müssen also erst eine Datei zum Lesen eröffnen. Das Handle dieser Datei ist natürlich ein Non-Standard-Handle. Wir "forcen" es auf die Konsol-Eingabe (Standard-Handle 0). Anschließend lesen wir die Datei mit Cconin-Aufrufen aus. In unserem Beispiel wollen wir dabei so vorgehen, daß eine Zeile gelesen wird, bis zum Erreichen des Codes 13 (Return). Zur Kontrolle geben wir den so erhaltenen String auf dem Bildschirm aus. Die Datei können wir wieder schließen, und als einziges Problem bleibt, die Standard-Eingabe wieder auf die Tastatur zurückzuleiten. Der Aufruf Force (0,0) hilft hier nicht weiter, und zwar aus zwei Gründen:

1. Das Standard-Handle 0 ist ja immer noch auf die Datei geschaltet (erster Fforce-Aufruf). Wenn wir also als neu zu schaltende Quelle der Standard-Eingabe die 0 angeben, so muß der Rechner ganz konsequent annehmen, daß wir auch die Datei meinen.

2. Wie Sie oben lesen konnten, muß der zweite Parameter des Fforce-Aufrufs ein Non-Standard-Handle sein.

Ein Non-Standard-Handle für den Bildschirm existiert aber doch gar nicht!

Glücklicherweise gibt es da noch die Funktion Fdup, der ein Standard-Handle als einziger Parameter übergeben wird. Dup steht für Duplizieren. Diese Funktion fertigt nämlich eine gleichwertige Kopie des angegebenen Standard-Handles an und gibt sie zurück. Die Kopie selbst ist aber ein Non-Standard-Handle, ein Handle also, das größer oder gleich 6 ist, wie wir es für den Fforce-Aufruf benötigen! Deshalb duplizieren wir ganz zu Anfang das Handle 0 und forcen am Ende die Standard-Eingabe auf die Kopie von 0. Ganz zu Anfang übrigens deshalb, weil ja nach dem ersten Fforce die Standard-Eingabe unsere Datei ist, und auch die Kopie würde sich auf die Datei beziehen. Daraus können Sie auch entnehmen, daß ein Fforce-Aufruf keine Auswirkung auf die Kopie eines Standard-Handles hat.

Und das soll einer verstehen, werden Sie sagen. Nun, auch ich hatte anfangs meine Probleme mit diesem schwierigen Thema, aber es hat sich gelohnt - halten Sie durch!

Fassen wir die einzelnen Schritte einmal zusammen:

1. Das Standard-Handle 0 wird mit Fdup dupliziert, so daß wir ein Non-Standard-Handle erhalten, welches eine exakte Kopie der Standard-Eingabe darstellt (nicht nur eine Kopie des Standard-Eingabe-Handles).

2. Eine Datei wird zum Lesen eröffnet.

3. Die Standard-Eingabe wird mit Fforce auf das Non-Standard-Handle dieser Datei umgeleitet.

4. Die Eingabe kann vorgenommen werden. Dabei benutzen wir die Funktion Cconin, die gewöhnlich von der Tastatur liest.

5. Die Datei wird wieder geschlossen.

6. Die Umleitung wird mit einem zweiten Fforce-Aufruf rückgängig gemacht, d.h. wir forcen die Standard-Eingabe auf das in Schritt 1 kopierte Non-Standard-Handle, das von dem Fforce in Schritt 3 nicht betroffen war, denn (siehe unter 1) wir haben ja die Standard-Eingabe dupliziert und nicht etwa nur ein synonymes Non-Standard-Handle erzeugt.

Nach einer so langen Vorrede haben Sie sich ein Programm zur Demonstration des oben gesagten redlich verdient:

GFA-BASIC

```
'
' Umleiten der Eingabe von Tastatur auf Datei
' GFA-BASIC      MP 08-10-88     UMLEITEN.GFA
'
filename$="UMLEITEN.DAT"+CHR$(0)
stdin%=0        ! Standard-Eingabehandle
lesen%=0        ! Parameter für Fopen-Funktion
'
' Datei mit Fopen zum Lesen öffnen:
'
handle%=GEMDOS(&H3D,L:VARPTR(filename$),lesen%)
'
IF handle%<0
  ' PRINT "Datei nicht gefunden!"
ELSE
  '
  ' Standard-Eingabe (Tastatur) duplizieren:
  '
  non_std_handle=GEMDOS(&H45,stdin%)
  '
  ' Standard-Eingabe auf Datei 'forcen':
  '
  VOID GEMDOS(&H46,stdin%,handle%)
  '
  leseroutine(a$)     ! Unterprogramm liest von Tastatur
  '
  PRINT a$            ! Kontroll-Ausgabe
  '
  ' Standard-Eingabe zurück umleiten (auf Kopie):
  '
  VOID GEMDOS(&H46,stdin%,non_std_handle%)
ENDIF
END
'
'
PROCEDURE leseroutine(VAR a$)
  a$=""
  DO
    zeichen%=GEMDOS(1)              ! Cconin für Tastatur-Eingabe
    EXIT IF (zeichen% AND 255)=13   ! RETURN gedrückt?
    a$=a$+CHR$(zeichen%)            ! nein, dann Zeichen anhängen  LOOP
RETURN

Omikron-BASIC'
' Umleiten der Eingabe von Tastatur auf Datei
' OMIKRON-BASIC    MP 25-09-88   UMLEITEN.BAS
'
Nameptr= .MEMORY("UMLEITEN.DAT")
'
```

```
' Datei UMLEITEN.DAT zum Lesen öffnen
'
GEMDOS (Handle%,$3D, HIGH(Nameptr), LOW(Nameptr),0)
IF Handle%<0
  THEN
    PRINT "Datei nicht gefunden!"
  ELSE
    '
    ' Standard-Eingabe-Handle (0) duplizieren:
    '
    GEMDOS (Non_Std_Handle%,$45,0)
    '
    ' Umleiten: Standard-Eingabe (0) auf Datei (Handle%)
    '
    GEMDOS (,$46,0,Handle%)
    '
    Eingabe(Zeile$)'   Eingabe von Datei
    PRINT Zeile$'      und anschließend Kontroll-Ausgabe
    '
    ' Umleitung rückgängig machen: Standard (0) auf Non_std
    '
    GEMDOS (,$46,0,Non_Std_Handle%)
    '
    ' Schließen der Datei:
    '
    GEMDOS (,$3E,Handle%)
  ENDIF
END
'
DEF PROC Eingabe(R A$)' Unterprogramm: GEMDOS-Tastatur-EingabeLOCAL Zeichen%
  A$=""
  REPEAT
    GEMDOS (Zeichen%,1)'  Cconin = Zeichen-Eingabe
    IF Zeichen%=13 THEN EXIT ENDIF '   Abbruch, wenn Return
    A$=A$+ CHR$(Zeichen%)
  UNTIL 0'    Endlosschleife
RETURN
```

C

```
/*****************************************************/
/*   Umleiten der Eingabe von Tastatur auf Datei   */
/*   Megamax Laser C    MP 25-09-88   UMLEITEN.C   */
/*****************************************************/

#include <osbind.h>

#define FILENAME "UMLEITEN.DAT"
#define STDIN 0                          /* Standard-Eingabehandle */
```

```
#define LESEN 0

int  non_std_handle;
int  handle;
char string[80];

eingabe (string)                /* Kleine Eingaberoutine für String */
char string[];
{
int  i;

  i = -1;
  do
  {
    string[++i] = Cconin();     /* Zeichen von Standard-Eingabe lesen */
    if (string[i] == 13)         /* bis Return (=13) gedrückt wurde; */
      string[i] = 0;           /* Return wird durch Nullbyte ersetzt */
  }
  while (string[i] != 0);
}

main()
{
  handle = Fopen (FILENAME, LESEN);      /* Aus dieser Datei sollen */
  if (handle < 0)                       /* gleich die Eingaben kommen */
    Cconws ("Datei nicht gefunden!");
  else
  {
    non_std_handle = Fdup (STDIN);   /* Tastatur-Handle duplizieren */
    Fforce (STDIN, handle);         /* Umleiten der Standard-Eingabe */
                                         /* auf die Datei <handle> */

    eingabe (string);          /* Die Eingabe kommt aus der Datei... */

    Cconws (string);           /* Kontroll-Ausgabe auf dem Bildschirm */

    Fforce (STDIN, non_std_handle);              /* Zurück umleiten */
    Fclose (handle);                           /* und Datei schließen */
  }

  Crawcin();                                       /* Taste beendet */
}
```

Assembler

```
        ;
        ; Umleiten der Eingabe von Tastatur auf Datei
        ; Assembler     MP   25-09-88      UMLEITEN.Q
        ;
```

```
gemdos  = 1

cconin  = 1
crawcin = 7
cconws  = 9
fopen   = $3d
fclose  = $3e
fdup    = $45
fforce  = $46

        TEXT

        clr.w   -(sp)           ;Datei zum Lesen öffnen
        pea     filename
        move.w  #fopen,-(sp)
        trap    #gemdos
        addq.l  #8,sp

        tst.w   d0              ;Fehler beim Öffnen?
        bmi.s   error           ;Dann melden...
        move.w  d0,handle       ;Sonst Wert als handle merken

        clr.w   -(sp)           ;0 ist Standard-Eingabe-Handle
        move.w  #fdup,-(sp)     ;und wird mit Fdup dupliziert
        trap    #gemdos
        addq.l  #4,sp
        move.w  d0,non_std      ;Ergebnis ist non-standard-handle

        move.w  handle,-(sp)    ;Umleitung: Bisherige Standard-
        clr.w   -(sp)           ;Eingabe (Tastatur) wird auf Datei
        move.w  #fforce,-(sp)   ;gelegt
        trap    #gemdos
        addq.l  #6,sp

        bsr.s   eingabe         ;Unterprogramm für Eingabe

        pea     string          ;Kontroll-Ausgabe
        move.w  #cconws,-(sp)
        trap    #gemdos
        addq.l  #6,sp

        move.w  non_std,-(sp)   ;Nochmalige Umleitung der
        clr.w   -(sp)           ;Standard-Eingabe (auf alte
        move.w  #fforce,-(sp)   ;Standard-Eingabe)
        trap    #gemdos
        addq.l  #6,sp

        move.w  handle,-(sp)    ;Datei schließen
        move.w  #close,-(sp)
        trap    #gemdos
```

```
          addq.l    #4,sp

ende:     move.w    #crawcin,-(sp)   ;Warten auf Taste
          trap      #gemdos
          addq.l    #2,sp

          moveq.l   #20,d0
l1:       moveq.l   #-1,d0
l2:       dbra      d1,l2
          dbra      d0,l1


          move.w    #2,-(sp)
          move.w    #2,-(sp)
          trap      #13
          addq.l    #4,sp


          clr.w     -(sp)            ;Programm beenden
          trap      #gemdos

error:    pea       errtext          ;Fehlermeldung ausgeben
          move.w    #cconws,-(sp)
          trap      #gemdos
          addq.l    #6,sp

          bra.s     ende

eingabe:  lea       string,a3        ;Eingaberoutine für eine Zeile
          clr.w     d3               ;d3 ist Index für nächstes Zeichen

loop:     move.w    #cconin,-(sp)    ;Zeichen von Standard-Eingabegerät
          trap      #gemdos          ;holen
          addq.l    #2,sp

          cmpi.b    #13,d0           ;Return betätigt?
          beq.s     finish           ;Dann sind wir fertig

          move.b    d0,0(a3,d3.w)    ;Sonst Zeichen in String schreiben
          addi.w    #1,d3            ;und Index erhöhen
          bra.s     loop

finish:   clr.b     0(a3,d3.w)       ;String mit Nullbyte abschließen
          rts


          DATA

filename: DC.b 'UMLEITEN.DAT',0
```

```
errtext:  DC.b 'Datei nicht gefunden!',0

          BSS

handle:   DS.w 1
non_std:  DS.w 1
string:   DS.b 80

          END
```

1.7 Nachladen von Programmen

Wenn GEMDOS es auch nicht erlaubt, mehrere Programme gleichzeitig laufen zu lassen (Multitasking), so kann ein Programm doch immerhin ein anderes vom Massenspeicher nachladen und starten lassen. Ist dieses zweite Programm beendet, so wird das erste an der Stelle fortgesetzt, wo es durch das zweite unterbrochen wurde. Im Prinzip funktioniert das also genauso wie der Aufruf von Unterprogrammen oder Funktionen, nur daß sie jeweils nachgeladen werden müssen.

Damit dürfte auch klar sein, wann sich ein solches Vorgehen lohnt, nämlich nur bei sehr großen Unterprogrammen, die nicht allzuoft aufgerufen werden, also z.B. nicht innerhalb einer Schleife. Ein Beispiel: Die Benutzeroberfläche (Shell) eines C-Entwicklungssystems ist ein Programm, das Editor, Compiler und Linker als sehr große Unterprogramme aufruft.

Ein Programm wird durch die GEMDOS-Funktion Pexec (Nummer $4B) geladen. Der Aufruf sieht so aus:

```
Pexec (Modus, P1, P2, P3)
```

P1, P2 und P3 sind Zeiger, also Long-Werte. Die Tatsache, daß ein Modus angegeben werden muß, läßt vermuten, daß Nachladen nicht gleich Nachladen ist. So ist es denn auch: Sie können wählen, ob das Programm geladen und sofort gestartet werden soll (Modus = 0) oder ob es nur in den Speicher geholt werden soll (Modus = 3), um es später besonders schnell starten zu können. In beiden Fällen werden die Zeiger P1 bis P3 wie folgt verwendet:

P1: Pfad-/Dateiname des Programms

P2: Parameter, die an das Programm übergeben werden sollen, und die gewöhnlich bei *.TTP-Programmen beim Start eingegeben werden

P3: Environment-String. Spielt unter MS-DOS eine wichtige Rolle. Beim Atari wird er eigentlich nie verwendet. Mein Tip: Vergessen Sie ihn wieder.

Pexec gibt auch einen Wert zurück. Bei einem Modus von 0 ist der Funktionswert vom Typ Integer (2 Bytes) und enthält den sogenannten Return-Code, also eine Art Funktionswert des nachgeladenen Programms. Dieser Wert wird am Ende des nachgeladenen Programms bestimmt: Wird es normal beendet, so ist der Return-Code 0; wird es aber mit der Funktion Pterm(retcode) (Nummer $4C) abgeschlossen, so wird der Parameter retcode an den Aufrufer zurückgegeben. Damit ist es z.B. möglich zu melden, daß das nachgeladene Programm auf Grund eines Fehlers (Datei nicht gefunden etc.) nicht korrekt arbeiten konnte.

Im Modus 3 wird hingegen ein Langwort zurückgegeben. Es ist die Startadresse der Basepage, die bei der Speicherverwaltung schon einmal kurz erwähnt wurde. Diese Startadresse merken Sie sich für die nächste Betriebsart der Pexec-Funktion: den Modus 4.

Im Modus 4 kann ein mit Modus 3 geladenes Programm gestartet werden. Dazu benötigt GEMDOS die Adresse der Basepage, die beim Laden ermittelt wurde. Diese Adresse übergeben Sie im Zeiger P2; P1 und P3 sind in diesem Modus ohne Bedeutung. Der Rückgabewert ist, wie schon im Modus 0, der Return-Code des Programms.

Zwei Programme sollen nun das gerade Gelernte demonstrieren, je eins für Modus 0 und Modus 3 und 4. Wir nehmen an, daß sich auf der Diskette ein Programm namens ZULADEN.TOS befindet. BASIC-Programmierer müssen, wie in Kapitel 1.6 erklärt, den GEMDOS-Speicher erhöhen und ihren eigenen etwas verkleinern. Die Assembler-Fans müssen erstmals ihren Speicherbedarf berechnen und außerdem einen großen Stack einrichten.

Eines ist noch wichtig für die Omikron- und GFA-BASIC-Besitzer: Das zweite Beispielprogramm (Laden und späteres Starten) läßt sich nur einmal problemlos starten - danach ist ein erneutes Laden des Interpreters nötig. Wenn Sie das Programm jedoch compilieren, so läuft es beliebig oft fehlerfrei.

GFA-BASIC

```
'
' Nachladen eines Programms und sofort starten mit Pexec
' GFA-BASIC             MP  08-10-88          NACHLAD.GFA
```

```
'
' 2000 Bytes für nachzuladendes Programm vom
' BASIC-Arbeitsspeicher abziehen:
'
RESERVE-2000
'
filename$="ZULADEN.TOS"+CHR$(0)
parameter$=CHR$(0)
environment$=CHR$(0)
'
filename%=VARPTR(filename$)
parameter%=VARPTR(parameter$)
environment%=VARPTR(environment$)
'
PRINT "Das ist das aufrufende Programm."
PRINT
'
IF GEMDOS(&H4B,0,L:filename%,L:parameter%,L:environment%)<0
  PRINT "Irgend ein Fehler ist aufgetaucht!!!"
  PRINT
ENDIF
'
PRINT "Hier ist wieder das aufrufende Programm!"
'
' Speicherveränderung wieder rückgängig machen:
'
RESERVE
END
```

```
'
' Nachladen eines Programms und Start zu späterem Zeitpunkt
' GFA-BASIC              MP  08-10-88          NACHLAD2.GFA
'
' 2000 Bytes für nachzuladendes Programm vom
' BASIC-Arbeitsspeicher abziehen:
'
RESERVE -2000
'
filename$="ZULADEN.TOS"+CHR$(0)
parameter$=CHR$(0)
environment$=CHR$(0)
'
filename%=VARPTR(filename$)
parameter%=VARPTR(parameter$)
environment%=VARPTR(environment$)
'
PRINT "Das ist das aufrufende Programm."
PRINT
```

```
'
' Modus 3: Programm laden
'
basepge%=GEMDOS(&H4B,3,L:filename%,L:parameter%,L:environment%)
IF basepge%<0
  PRINT "Irgend ein Fehler ist aufgetaucht!!!"
  PRINT
ELSE
  '
  PRINT "Programm wurde geladen. TASTE zum Start!"
  PRINT
  VOID GEMDOS(7)  ! Crawcin wartet auf Taste
  '
  ' Modus 4: Programm starten
  '
  VOID GEMDOS(&H4B,4,L:0,L:basepge%,L:0)
  '
ENDIF
'
PRINT "Hier ist wieder das aufrufende Programm!"
'
' Speicherveränderung wieder rückgängig machen:
' (funktioniert im Interpreter nicht so ganz...)
'
RESERVE
'
END
```

Omikron-BASIC

```
'
' Programm nachladen und sofort starten mit Pexec
' OMIKRON-BASIC      MP  25-09-88     NACHLAD.BAS
'
CLEAR 10000:' 10000 Bytes für nachzuladendes Programm
'
Nameptr= MEMORY("ZULADEN.TOS")
Parameter= MEMORY("")
Environment=Parameter
'
GEMDOS (,2,27)'        Umständlich, muß aber sein:
GEMDOS (,2, ASC("H"))' Der Cursor wird mit VT-52-Sequenzen
GEMDOS (,2,10)'        in die dritte Zeile gebracht (ESC+H
GEMDOS (,2,10)'        für Home und 2mal 10 für Zeilenvorschub)
'
CLS :' Bildschirm löschen
'
PRINT "Das ist das aufrufende Programm.": PRINT
'
' GEMDOS-Aufruf zum Laden und Starten:
```

```
'
GEMDOS (Ret%,$4B,0, HIGH(Nameptr), LOW(Nameptr), HIGH(Parameter),...
...LOW(Parameter), HIGH(Enrironment), LOW(Environment))
'
IF Ret%<0
  THEN PRINT "Irgendein Fehler ist aufgetaucht!!!"
ENDIF
'
PRINT : PRINT
PRINT "Hier ist wieder das aufrufende Programm!"
'
GEMDOS (,7):' Warten auf Taste (Crawcin)
END


'
' Programm nachladen und später starten mit Pexec
' OMIKRON-BASIC      MP 25-09-88     NACHLAD2.BAS
'
CLEAR 10000:' 10000 Bytes für nachzuladendes Programm
'
Nameptr= MEMORY("ZULADEN.TOS")
Parameter= MEMORY("")
Environment=Parameter
'
GEMDOS (,2,27)'        Wie gehabt, aber der Cursor wird in
GEMDOS (,2, ASC("H"))' Zeile 5 gesetzt
FOR I%=1 TO 4: GEMDOS (,2,10): NEXT I%
'
CLS :' Bildschirm löschen
'
PRINT "Das ist das aufrufende Programm.": PRINT
'
' GEMDOS-Aufruf zum Laden ohne Starten (Modus 3):
'
GEMDOS (Basepage,$4B,3, HIGH(Nameptr), LOW(Nameptr),...
...HIGH(Parameter), LOW(Parameter), HIGH(Environment),...
...LOW(Environment))
'
IF Basepage<0
  THEN PRINT "Irgendein Fehler ist aufgetaucht!!!"
  ELSE
  '
  PRINT "Programm wurde geladen. TASTE zum Start!": PRINT
  GEMDOS (,7):' Crawcin wartet auf Taste
  '
  ' Programm starten (Modus 4):
  '
  GEMDOS (,$4B,4,0,0, HIGH(Basepage), LOW(Basepage),0,0)
  '
```

```
ENDIF
'
PRINT : PRINT
PRINT "Hier ist wieder das aufrufende Programm!"
'
GEMDOS (,7):' Warten auf Taste (Crawcin)
END
```

C

```
/*********************************************************/
/*   Programm nachladen und sofort starten mit Pexec   */
/*   Megamax Laser C      MP  24-09-88     NACHLAD.C   */
/*********************************************************/

#include <osbind.h>

#define LADEN_UND_STARTEN 0
#define FILENAME "ZULADEN.TOS"
#define PARAMETER ""
#define ENVIRONMENT ""

main()
{
  Cconws ("Das ist das aufrufende Programm.\15\12\12");

  if (Pexec (LADEN_UND_STARTEN, FILENAME, PARAMETER, ENVIRONMENT) < 0)
    Cconws ("Irgendein Fehler ist aufgetaucht!!!\15\12\12");

  Cconws ("Hier ist wieder das aufrufende Programm!\15\12");

  Crawcin();     /* Ende mit Tastendruck */
}


/*****************************************************************/
/*   Nachladen eines Programms und Start zu späterem Zeitpunkt   */
/*   Megamax Laser C          MP  24-09-88          NACHLAD2.C   */
/*****************************************************************/

#include <osbind.h>

#define LADEN 3
#define STARTEN 4
#define FILENAME "ZULADEN.TOS"
#define PARAMETER ""
#define ENVIRONMENT ""

long  basepage;
```

```
main()
{
  Cconws ("Das ist das aufrufende Programm.\15\12\12");

  basepage = Pexec (LADEN, FILENAME, PARAMETER, ENVIRONMENT);
  if (basepage < 0)
    Cconws ("Irgendein Fehler ist aufgetaucht!\15\12\12");
  else
  {
    Cconws ("Programm wurde geladen. TASTE zum Start!\15\12\12");    Crawcin();

    Pexec (STARTEN, 0L, basepage, 0L);

    Cconws ("Jetzt ist wieder das alte Programmm da!\15\12\12");
  }

  Crawcin();
}
```

Assembler

```
          ;
          ;Programm nachladen und sofort starten mit Pexec
          ;Assembler        MP  25-09-88         NACHLAD.Q
          ;

gemdos    = 1

crawcin   = 7
cconws    = 9
mshrink   = $4a
pexec     = $4b

          TEXT

          movea.l   4(sp),a5          ;Speicherbedarf ermitteln
          move.l    12(a5),d0         ;Länge des Programmcodes
          add.l     20(a5),d0         ;+ Länge des Data-Segments
          add.l     28(a5),d0         ;+ Länge des BSS-Segments +
          addi.l    #$1100,d0         ;Basepage (256 Bytes)+Stack (4KB)

          move.l    d0,d1             ;Länge plus
          add.l     a5,d1             ;Startadresse
          andi.l    #-2,d1            ;(gerundet)
          movea.l   d1,sp             ;ergibt Stackpointer

          move.l    d0,-(sp)          ;Länge des benötigten Speichers
          move.l    a5,-(sp)          ;Startadresse des Bereichs
          clr.w     -(sp)             ;dummy-Byte ohne Bedeutung
          move.w    #mshrink,-(sp)
```

```
          trap      #gemdos
          adda.l    #12,sp

          pea       text1           ;Meldung am Bildschirm ausgeben
          move.w    #cconws,-(sp)
          trap      #gemdos
          addq.l    #6,sp

          pea       environ         ;Programm nachladen
          pea       params
          pea       filename
          clr.w     -(sp)           ;Null -> Laden und sofort starten
          move.w    #pexec,-(sp)
          trap      #gemdos
          adda.l    #16,sp

          tst.w     d0              ;Fehler aufgetreten?
          bmi.s     error

          pea       text2           ;noch eine Meldung...
          bra.s     ende

error:    pea       errtext         ;Fehlermeldung ausgeben
ende:     move.w    #cconws,-(sp)
          trap      #gemdos
          addq.l    #6,sp

          move.w    #crawcin,-(sp)  ;Warten auf Taste...
          trap      #gemdos
          addq.l    #2,sp

          clr.w     -(sp)           ;... und Programm beenden
          trap      #gemdos

          DATA

text1:    DC.b 'Das ist das aufrufende Programm.',13,10,10,0
text2:    DC.b 'Hier ist wieder das aufrufende Programm!',13,10,0
errtext:  DC.b 'Irgendein Fehler ist aufgetaucht!!!',13,10,10,0

filename: DC.b 'ZULADEN.TOS',0
params:   DC.b 0   ;keine Parameter und
environ:  DC.b 0   ;keine Umgebung

          END


          ;
          ;Programm nachladen und später starten mit Pexec
          ;Assembler        MP  25-09-88        NACHLAD2.Q
```

```
        ;

gemdos  = 1

crawcin = 7
cconws  = 9
mshrink = $4a
pexec   = $4b

        TEXT

        movea.l 4(sp),a5        ;Speicherbedarf ermitteln
        move.l  12(a5),d0       ;Länge des Programmcodes
        add.l   20(a5),d0       ;+ Länge des Data-Segments
        add.l   28(a5),d0       ;+ Länge des BSS-Segments +
        addi.l  #$1100,d0       ;Basepage (256 Bytes)+Stack (4KB)

        move.l  d0,d1           ;Länge plus
        add.l   a5,d1           ;Startadresse
        andi.l  #-2,d1          ;(gerundet)
        movea.l d1,sp           ;ergibt Stackpointer

        move.l  d0,-(sp)        ;Länge des benötigten Speichers
        move.l  a5,-(sp)        ;Startadresse des Bereichs
        clr.w   -(sp)           ;dummy-Byte ohne Bedeutung
        move.w  #mshrink,-(sp)
        trap    #gemdos
        adda.l  #12,sp

        pea     text1           ;Meldung am Bildschirm ausgeben
        move.w  #cconws,-(sp)
        trap    #gemdos
        addq.l  #6,sp

        pea     environ         ;Programm nachladen
        pea     params
        pea     filename
        move.w  #3,-(sp)        ;3 -> nur laden und Startadresse
        move.w  #pexec,-(sp)    ;zurückgeben
        trap    #gemdos
        adda.l  #16,sp

        tst.w   d0              ;Fehler aufgetreten?
        bmi.s   error           ;dann Meldung ausgeben...
        move.l  d0,start_adr    ;sonst Startadresse merken
        pea     text2           ;Melden: Programm ist
        move.w  #cconws,-(sp)   ;bereit zum Start
        trap    #gemdos
        addq.l  #6,sp
```

```
          move.w    #crawcin,-(sp)   ;Warten auf Taste...
          trap      #gemdos
          addq.l    #2,sp

          clr.l     -(sp)            ;Programm jetzt erst starten
          move.l    start_adr,-(sp)  ;Startadresse haben wir vom
          clr.l     -(sp)            ;ersten Pexec-Aufruf
          move.w    #4,-(sp)         ;4 -> Programm starten
          move.w    #pexec,-(sp)
          trap      #gemdos
          adda.l    #16,sp

          pea       text3            ;noch eine Meldung...
          bra.s     ende

error:    pea       errtext          ;Fehlermeldung ausgeben
ende:     move.w    #cconws,-(sp)
          trap      #gemdos
          addq.l    #6,sp

          move.w    #crawcin,-(sp)   ;Warten auf Taste...
          trap      #gemdos
          addq.l    #2,sp

          clr.w     -(sp)            ;... und Programm beenden
          trap      #gemdos

          DATA

text1:    DC.b 'Das ist das aufrufende Programm.',13,10,10,0
text2:    DC.b 'Programm wurde geladen. Taste zum Start!',13,10,10,0
text3:    DC.b 'Jetzt ist wieder das alte Programm da!',13,10,10,0
errtext:  DC.b 'Irgend ein Fehler ist aufgetaucht!!!',13,10,10,0

filename: DC.b 'ZULADEN.TOS',0
params:   DC.b 0   ;keine Parameter und
environ:  DC.b 0   ;keine Umgebung

          BSS

start_adr:DS.l 1

          END
```

1.8 Von Tracks und Sektoren

Ich glaube eigentlich nicht, daß Sie sich oft mit Tracks und Sektoren auseinandersetzen müssen (es sei denn, Sie wollten einen Disketten-Mo-

nitor programmieren). Trotzdem möchte ich Ihnen ein paar Dinge über die Diskettenverwaltung erzählen, weil dies erstens einen sehr schönen Überblick über das ganze TOS (also GEMDOS, BIOS und XBIOS) gibt, und zweitens, weil man es früher oder später doch einmal kennenlernen muß, z.B. um eine RAM-Disk oder ähnliches zu erstellen. Wahrscheinlich ist Ihnen vieles schon bekannt, doch in diesem Fall ist der Zusammenhang wichtig.

GEMDOS, das wissen Sie, kennt Dateien und Ordner (Unterverzeichnisse). Jede Datei wird durch einen Pfadnamen charakterisiert, der aus dem Laufwerksnamen, evtl. dem Weg durch verschiedene Ordner und schließlich aus dem Dateinamen selbst besteht. Wenn auf eine bestimmte Information zugegriffen werden soll, so benötigt GEMDOS nur diesen Pfadnamen und den Offset der Information, also deren Lage innerhalb der Datei. Was für Sie so selbstverständlich klingt, erfordert in Wirklichkeit eine sehr aufwendige Verwaltung - denken Sie nur daran, daß je nach Laufwerksname auf völlig unterschiedliche Peripheriegeräte zugegriffen werden muß (Diskette, Festplatte, RAM-Disk...). Mit dieser Verwaltung ist GEMDOS überfordert. Sie wird deshalb teilweise vom BIOS und auch vom XBIOS übernommen. Wie hat man sich das vorzustellen?

Nun, das Anwenderprogramm gibt GEMDOS wie eben beschrieben einen Pfadnamen. GEMDOS rechnet nun aus, in welchem logischen Sektor die gesuchte Information gespeichert ist. Ein logischer Sektor ist ein Block von 512 Bytes. Der ganze Massenspeicher besteht aus durchnumerierten logischen Sektoren; eine gewöhnliche einseitige Diskette besitzt z.B. 720 Sektoren mit den Nummern 0 bis 719. Und damit wären wir bereits beim BIOS. BIOS kennt keine Dateien mehr, auch keine Laufwerksnamen. Statt dessen arbeitet man mit Laufwerksnummern (0 = Laufwerk A, 1 = Laufwerk B usw.) und logischen Sektornummern. BIOS hat nun die Aufgabe, anhand der Laufwerksnummer den Gerätetreiber für das gewünschte Laufwerk (z.B. einen Harddisk-Treiber) zu aktivieren, der dann - endlich - die Daten auch wirklich liest. Nur, wenn ein Diskettenlaufwerk (A oder B) gefragt ist, muß BIOS noch etwas tun: Es muß die logische Sektornummer so verarbeiten, daß am Ende der Rechnung eine Tracknummer, eine Sektornummer und bei doppelseitigen Disketten eine Diskettenseite herauskommt. Diese drei Werte ergeben schließlich den physikalischen Sektor. Schließlich ruft BIOS noch eine XBIOS-Routine auf, die den physikalischen Sektor von der Diskette liest.

Ich hoffe, die Erklärungen waren nicht allzu trocken. Es gehört ganz einfach in den Bereich der Systemprogrammierung, sich mit solchen Din-

gen auseinanderzusetzen. Damit das Ganze nicht nur Theorie bleibt, sollten Sie sich im Intern 1 einmal die BIOS-Funktion Rwabs sowie die XBIOS-Funktionen Floprd, Flopwr und Flopfmt anschauen.

Nun möchte ich noch ein kleines Beispielprogramm geben. Es ist nicht nur für Demonstrationszwecke gedacht, sondern kann möglicherweise lebensrettend für Ihre Daten sein. Ich rede von einem Virus-Detektor. Halt, bevor Sie das Buch wieder zuschlagen, möchte ich gleich sagen, daß ich nicht das Programm geschrieben habe, das alle Viren vernichtet, wie es manch einer von seinen Programmen behauptet. Mein Virus-Detektor prüft lediglich den Bootsektor einer Diskette auf dessen Ausführbarkeit. Dazu müssen Sie wissen, daß jeder Bootsektor (das ist übrigens der erste Sektor auf dem ersten Track einer jeden Diskette) ein kleines Programm enthalten kann, das der Rechner beim Booten (daher der Name Boot-Sektor) automatisch ausführt, und zwar viel schneller als ein Auto-Ordner-Programm. Deshalb ist er auch bei den Virus-Programmiern so beliebt - die Zeit, die er braucht, um gestartet zu werden, ist minimal, und er benötigt weder einen Eintrag im Inhaltsverzeichnis noch Speicherplatz auf der Diskette.

Dieses Programm wird beim Booten jedoch nur ausgeführt, wenn die Checksumme des Bootsektors (das ist die Summe aller 256 Worte(!), was bekanntlich den 512 Bytes in einem Sektor entspricht) $1234 (dezimal: 4660) beträgt, wobei diese Zahl willkürlich von den Entwicklern des TOS gewählt wurde. Unser Virus-Detektor wird also den Bootsektor einer Diskette lesen und die Checksumme berechnen. Beträgt diese $1234, dann besteht akute Virus-Gefahr! Es könnte allerdings auch sein, daß es sich um ein freundlich gesinntes Bootsektor-Programm handelt, z.B. um den Loader eines Spiels, aber auch um das Ladeprogramm für Ihr Betriebssystem, falls Sie mit einem Disketten-TOS arbeiten (der Bootsektor auf einer Systemdiskette ist stets ausführbar). Bei Anwendersoftware oder gar Ihren eigenen Datendisketten ist das aber auszuschließen, so daß es sich wahrscheinlich um einen echten Virus handelt. Um ihn zu deaktivieren, müssen wir lediglich ein Byte des Bootsektors so abändern, daß die Checksumme nicht mehr $1234 beträgt. Dazu bietet sich das letzte Wort eines Bootsektors an; es wurde speziell dafür reserviert, die Checksumme falls gewünscht auf $1234 bringen zu können.

Folgendes Programm macht das automatisch (der Sektor wird mit den XBIOS-Funktionen Floprd und Flopwr gelesen und geschrieben; siehe Intern 1):

GFA-BASIC

```
'
' Virus-Detektor (sucht ausführbaren Bootsektor)
' GFA-BASIC         MP  07-10-88       VIRUS.BAS
'
puffer%=MALLOC(512)
'
PRINT "Bitte legen Sie eine Diskette in Laufwerk A ein!"
VOID GEMDOS(7)
'
' Track 0 Sektor 1 (Bootsektor) lesen
'
IF XBIOS(8,L:puffer%,L:0,0,1,0,0,1)<0
  PRINT "Diskette fehlerhaft!"
ELSE
  summe%=0
  '
  FOR i%=0 TO 510 STEP 2
    ADD summe%,DPEEK(puffer%+i%)
  NEXT i%
  '
  summe%=summe% AND 65535
  '
  IF summe%=&H1234
    PRINT "Der Bootsektor ist ausführbar!"
    PRINT "Soll ich das ändern?  ";
    '
    IF UPPER$(INPUT$(1))="J"
      PRINT "Ja"
      PRINT
      '
      ' Ein Byte ändern, so daß die Checksumme nicht mehr stimmt:      '
      POKE puffer%+511,(PEEK(puffer%+511)+1) AND 255
      '
      ' Puffer zurückschreiben:
      '
      ret%=XBIOS(9,L:puffer%,L:0,0,1,0,0,1)<0
      '
      IF ret%=-13
        PRINT "Disk war schreibgeschützt"
      ELSE
        IF ret%<0
          PRINT "Disk-Error"
        ENDIF
      ENDIF
    ENDIF
  ELSE
    PRINT "Der Bootsektor ist nicht ausführbar!"
```

```
  ENDIF
ENDIF
END
```

Omikron-BASIC

```
'
' Virus-Detektor (sucht ausführbaren Bootsektor)
' OMIKRON-BASIC      MP  28-09-88      VIRUS.BAS
'
Puffer$= STRING$(512,0)' Hierhin wird der Bootsektor geladen
Puffer= LPEEK( VARPTR(Puffer$))+ LPEEK( SEGPTR +28)' Adresse berechnen
'
CLS
PRINT "Bitte legen Sie eine Diskette in Laufwerk A: ein!"
REPEAT : UNTIL INKEY$ >""
'
' Track 0 Sektor 1 (Bootsektor) lesen
'
XBIOS (Ret,8, HIGH(Puffer), LOW(Puffer),0,0,0,1,0,0,1)
'
IF NOT (Ret<0) THEN
  '
  ' Bilden der Checksumme
  '
  Summe=0
  FOR I=0 TO 510 STEP 2
    Summe=Summe+ WPEEK(Puffer+I)
  NEXT I
  '
  Summe=Summe AND 65535' Nur 16 Bit berücksichtigen
  '
  IF Summe=$1234
    THEN PRINT "Der Bootsektor ist ausführbar!"
         PRINT "Soll ich das ändern?  ";
         IF UPPER$( INPUT$(1))="J"
           THEN PRINT "Ja": PRINT
                '
                ' Ein Byte ändern
                '
                POKE (Puffer+511),( PEEK(Puffer+511)+1) AND 255
                '
                ' Puffer zurückschreiben
                '
                XBIOS (Ret,9, HIGH(Puffer), LOW(Puffer),...
...0,0,0,1,0,0,1)
                IF Ret=-13 THEN
                  PRINT "Disk war schreibgeschützt" ELSE
                IF Ret<0 THEN PRINT "Disk-Error": ENDIF : ENDIF
           ENDIF
```

```
    ELSE PRINT "Der Bootsektor ist nicht ausführbar!"
  ENDIF
  '
ELSE PRINT "Diskette fehlerhaft!"
ENDIF
END
```

C

```
/*************************************************************/
/*   Virus-Detektor (prüft, ob Bootsektor ausführbar ist   */
/*   Megamax Laser C         MP  29-09-88        VIRUS.C   */
/*************************************************************/

#include <osbind.h>

#define  TRACK  0
#define  SEKTOR 1
#define  DEVICE 0
#define  SIDE   0

int  puffer[256];           /* 512 Bytes für einen Sektor */
int  summe = 0;
int  back;
int  i;
char janein;

main()
{
  Cconws ("Bitte legen Sie eine Diskette in Laufwerk A: ein!\15\12");
  Crawcin();                    /* Warten auf Tastendruck */

         /* Bootsektor lesen (1 = nur einen Sektor lesen) */
  back = Floprd (puffer, 0L, DEVICE, SEKTOR, TRACK, SIDE, 1);

  if (back < 0)
    Cconws ("Diskette fehlerhaft!\15\12");
  else
  {
    for (i = 0;  i < 256;  i++)
      summe += puffer[i];

    if (summe == 0x1234)          /* $1234 --> ausführbar */
    {
      Cconws ("Der Bootsektor ist ausführbar.\15\12");
      Cconws ("Soll ich das ändern?  ");

      do
        janein = Crawcin();
```

```
    while (!(janein=='j' || janein=='n'));

    if (janein=='j')
    {
      Cconws ("Ja\15\12");

      ++puffer[255];     /* letztes Wort um eins erhöhen */

                              /* Sektor zurückschreiben */
      back = Flopwr (puffer, 0L, DEVICE, SEKTOR, TRACK, SIDE, 1);

      if (back == -13)
        Cconws ("Disk war schreibgeschützt\15\12");
      else
        if (back < 0)
          Cconws ("Disk-Error\15\12");
    }
  }
  else
    Cconws ("Bootsektor nicht ausführbar!\15\12");
}

  Cconws ("\15\12\12Taste = Ende...");
  Crawcin();                  /* Nochmal warten auf Taste */
}
```

Assembler

```
          ;
          ; Virus-Detektor (prüft, ob Bootsektor ausführbar ist)
          ; Assembler          MP  29-09-88            VIRUS.Q
          ;

gemdos    = 1
xbios     = 14

crawcin   = 7
cconws    = 9
floprd    = 8
flopwr    = 9

          TEXT

          lea       text1,a0        ;Meldung: Bitte Diskette einlegen
          bsr       print
          bsr       taste

          ; Track 0 Sektor 1 lesen

          move.w    #1,-(sp)        ;einen Sektor lesen
```

```
        clr.w   -(sp)           ;Seite 0
        clr.w   -(sp)           ;Track 0
        move.w  #1,-(sp)        ;Sektor 1
        clr.w   -(sp)           ;Laufwerk A:
        clr.l   -(sp)           ;Filler, ohne Bedeutung
        pea     puffer          ;da soll er hin
        move.w  #floprd,-(sp)   ;Funktionsnummer
        trap    #xbios
        adda.l  #20,sp

        tst.w   d0              ;Fehler?
        bmi     readerr

        ; Checksumme berechnen

        clr.w   d0
        lea     puffer,a0
        move.w  #255,d1         ;256 Worte addieren
loop:   add.w   (a0)+,d0
        dbra    d1,loop

        cmpi.w  #$1234,d0       ;ausführbar?
        bne.s   not_ex

        lea     frage,a0        ;ja, dann fragen, ob Bootsektor
        bsr     print           ;nicht ausführbar werden soll

inloop: bsr.s   taste
        cmpi.b  #'n',d0         ;n gedrückt?
        beq.s   ende
        cmpi.b  #'j',d0         ;oder j?
        bne.s   inloop          ;beides nicht, dann nochmal

        lea     ok,a0
        bsr.s   print

        addi.b  #1,puffer+511   ;letztes Byte verändern

        ; Sektor zurückschreiben

        move.w  #1,-(sp)        ;einen Sektor schreiben
        clr.w   -(sp)           ;Seite 0
        clr.w   -(sp)           ;Track 0
        move.w  #1,-(sp)        ;Sektor 1
        clr.w   -(sp)           ;Laufwerk A:
        clr.l   -(sp)           ;Filler, ohne Bedeutung
        pea     puffer          ;da steht er im Speicher
        move.w  #flopwr,-(sp)   ;Funktionsnummer
        trap    #xbios
```

```
          adda.l    #20,sp

          cmpi.w    #-13,d0         ;Disk schreibgeschützt?
          beq.s     schutz

          tst.w     d0              ;sonst irgend ein Fehler?
          bpl.s     ende            ;nicht, dann Ende

          lea       disktxt,a0
          bra.s     errcont

schutz:   lea       sch_txt,a0
          bra.s     errcont

not_ex:   lea       text2,a0
          bra.s     errcont

readerr:  lea       retext,a0
errcont:  bsr       print

ende:     lea       endtext,a0
          bsr.s     print
          bsr       taste

          clr.w     -(sp)           ;Programm-Ende
          trap      #gemdos

taste:    move.w    #crawcin,-(sp)  ;Warten auf Tastendruck
          trap      #gemdos
          addq.l    #2,sp
          rts

print:    pea       (a0)            ;Text ausgeben, auf den
          move.w    #cconws,-(sp)   ;a0 zeigt
          trap      #gemdos
          addq.l    #6,sp
          rts

          DATA

text1:    DC.b 'Bitte legen Sie eine Diskette in Laufwerk A: ein!'
          DC.b 13,10,0

text2:    DC.b 'Bootsektor nicht ausführbar!',13,10,0

frage:    DC.b 'Der Bootsektor ist ausführbar.',13,10
          DC.b 'Soll ich das ändern?   ',0

ok:       DC.b 'Ja',13,10,0
```

```
disktxt:  DC.b 'Disk-Error',13,10,0

sch_txt:  DC.b 'Disk war schreibgeschützt!',13,10,0

retext:   DC.b 'Diskette fehlerhaft!',13,10,0

endtext:  DC.b 13,10,10,'Taste = Ende...',0

          BSS

puffer:   DS.b 512

          END
```

2. Grafik

Die Überschrift dieses Kapitels könnte vermuten lassen, ich wollte auf den folgenden Seiten erklären, wie Sie den Bildschirm Ihres Rechners mit Linien, Kreisen oder anderen Objekten füllen können. Falsche Vermutung! Erstens ist diese Grafik-Programmierung stark von der benutzen Programmiersprache abhängig, und zweitens hat sie - je nach Programmiersprache - mit unserem Thema, der Systemprogrammierung, nicht mehr viel zu tun.

Wir wollen vielmehr zwei ausgewählte Themen näher erläutern: das Arbeiten mit zwei verschiedenen Bildschirm-Speichern und die Programmierung von Sprites. Die Ausgabe von grafischen Objekten wird ohnehin im Kapitel 5 (GEM-Programmierung) ausführlich behandelt.

2.1 Zwei Bildschirmspeicher

Zunächst möchte ich einige grundsätzliche Dinge erklären, die man nicht unbedingt als allgemein bekannt voraussetzen kann. Eine für den Anwender selbstverständliche Sache, nämlich die Darstellung von Buchstaben, Ziffern und Bildern auf einem Monitor, ist bei näherer Betrachtung gar nicht mehr so selbstverständlich. Woher weiß denn dieser Monitor überhaupt, was er wo darstellen soll?

Betrachten wir die Sache doch von ihrem Ursprung her: vom Rechner. Ein Programm will z.B. ein bestimmtes Zeichen an einer bestimmten Stelle des Bildschirms ausgeben. Der Prozessor in unserem Rechner muß dieses Problem lösen; denn er allein ist für die Abarbeitung dieses Programms, also auch für die Ausgabe des Zeichens, verantwortlich. Nun gibt es aber weder einen Befehl des Prozessors, mit dem das Programm den Bildschirm steuern kann, noch sind an den Prozessor Leitungen angeschlossen, die einem Monitor die zur Darstellung eines Bildes nötigen Signale liefern können.

Deshalb gibt es in jedem Rechner einen Zwischenschritt: Das Bindeglied zwischen dem Prozessor und dem Monitor ist ein spezialisierter Chip, der sich ausschließlich um die Bildwiedergabe kümmert; er heißt deshalb Video-Chip. Im Atari ST trägt der Video-Chip zusätzlich noch den Namen Shifter. Der Shifter kann einen Monitor direkt ansteuern, ist also mit der Video-Buchse des Computers verbunden. Übrigens, das gilt für andere

Peripheriegeräte des Rechners gleichermaßen: Floppy, Hard-Disk, Drucker, Maus...alle sind über einen solchen Zwischenschritt mit dem Prozessor verbunden. Man nennt einen Chip wie den Shifter deshalb auch Peripherie-Baustein. Ich bleibe hier jedoch (stellvertretend für alle anderen) beim Shifter, schon alleine, weil er zu unserem Thema Grafik gehört.

Die Kommunikation zwischen dem Chip und dem Prozessor ist über sogenannte Register gewährleistet. Die können Sie sich als Mini-Speicher (1, 2 oder 4 Bytes) vorstellen, auf die der Prozessor etwa mit move-Befehlen (in BASIC: POKE) zugreifen kann, die aber nicht zum Hauptspeicher (RAM) des Rechners gehören, sondern im Chip gespeichert werden. Der hat nun ebenfalls Zugriff auf diese Speicher, und so kann über diese Register eine Kommunikation zwischen Chip und Prozessor erfolgen.

Die Register haben genau festgelegte Aufgaben. Aus diesen Aufgaben ergibt sich zum Teil auch, daß bestimmte Register nur gelesen, andere nur beschrieben werden dürfen, während einige gleichermaßen gelesen und beschrieben werden können. Das ergibt sich daraus, daß Register teilweise dazu verwendet werden, die Befehle des Prozessors an den Chip weiterzuleiten, teilweise aber auch dem Prozessor den Zustand des Peripheriebausteins mitteilen, also eventuelle Fehler melden (nicht beim Shifter, aber z.B. beim Floppy-Disk-Controller).

Die folgenden Informationen beziehen sich wieder nur auf den Shifter; alle anderen Peripheriebausteine sind natürlich im ersten Band des Intern dokumentiert. Nun könnte man annehmen, daß der Prozessor dem Shifter über diese Register die Aufgabe gibt, das gewünschte Zeichen an die gewünschte Stelle des Bildschirms zu schreiben. Diese Aufgabe ist jedoch viel zu komplex für den Chip. Der weiß nämlich gar nicht, wie Buchstaben oder Ziffern aussehen. Jeden einzelnen Punkt des Bildschirms möchte er vom Prozessor gemeldet bekommen, das heißt jedes Zeichen auf dem Bildschirm ist eigentlich nur eine Kombination von Punkten, die ihre Farbe, schwarz oder weiß, vom Prozessor erhalten. Der Shifter sorgt dann nur noch dafür, daß die entsprechenden Punkte auf dem Bildschirm ebenfalls schwarz bzw. weiß erscheinen.

Nun macht es die Technik eines Monitors allerdings erforderlich, daß der gesamte Bildschirminhalt in kurzen Abständen immer neu auf den Monitor geschrieben wird. Beim Monochrom-Monitor des ST geschieht das immerhin 71mal in der Sekunde. Nur deshalb steht das Bild ruhig und flackert nicht. Der Shifter hat nun die Aufgabe, jede Sekunde diese 71 Bilder nach den Wünschen des Prozessors herzustellen. Im Monochrom-

Betrieb hat jedes Bild eine Auflösung von 640 x 400 Punkten, besteht also aus 256.000 einzelnen Punkten. Schwarz und weiß läßt sich durch ein Bit ausdrücken (gesetzt = schwarz, nicht gesetzt = weiß), also können wir diese Bildinformation in 256.000 durch 8 = 32.000 Bytes ausdrücken. Statten wir also den Shifter mit 32.000 Registern aus, und er hat alle Informationen, die er braucht...

Allerdings existiert ein Peripheriebaustein mit 32.000 Registern noch nicht, jedenfalls nicht für Rechner, die die Dimensionen eines Atari ST aufweisen. Da man aber um die 32.000 Bytes nicht herumkommt, werden sie aus dem Video-Chip ausgelagert und vom normalen Hauptspeicher des Rechners "geliehen". Über ein paar Register kann der Prozessor dann dem Chip mitteilen, an welcher Adresse des Hauptspeichers dieser 32.000 Bytes große Block steht. Diesen Block bezeichnet man als Bild-Wiederhol-Speicher, der bequemere Ausdruck ist Bildschirmspeicher oder auch kurz: Video-RAM.

In diesem Video-RAM findet der Shifter also all die Informationen, aus denen er ein Bild für den Monitor machen kann. Die Startadresse für diesen Speicherbereich kann jedoch, das wurde eben schon einmal angedeutet, vom Prozessor, also von einem Programm, selbst bestimmt werden, indem diese Adresse in Registern des Shifters untergebracht wird. Die einzige Spielregel, die es dabei zu beachten gibt, ist, daß die 32.000 Bytes an einer sogenannten Pagegrenze beginnen muß. Dazu sollten Sie wissen, daß man den gesamten Speicher eines Rechners in Pages (engl. Seiten) aufteilt, wobei jede Seite eine Größe von 256 (= $100) Bytes hat. Eine Pagegrenze ist nun die Startadresse einer solchen Seite, also eine Adresse, die durch 256 ($100) teilbar ist. Im Hexadezimalsystem erkennt man solche Adressen ganz einfach daran, daß die letzten beiden Ziffern der Adresse 0 sind.

Nun könnte man auf die Idee kommen, einen zweiten Bildschirmspeicher einzurichten. Wir hätten dann immer einen Block, der vom Shifter auf den Monitor projiziert wird, und einen, auf den der Rechner bei der Ausgabe von Grafik und Text zugreift. So wird es beim Atari gemacht. Allerdings merken Sie in der Regel nichts davon, weil gewöhnlich beide Bildschirmspeicher identisch sind. Theoretisch gibt es aber auch dann einen Unterschied zwischen dem sog. physikalischem Video-RAM (das der Shifter zum Monitor schickt) und dem logischen Video-RAM (das der Prozessor für seine Ausgaben verwendet).

Logisches und physikalisches Video-RAM müssen aber nicht unbedingt übereinstimmen. Ein Programm kann jederzeit vom Betriebssystem einen Speicherbereich anfordern, der als zweites Video-RAM dienen kann. Ha-

ben wir also unterschiedliche Bereiche für den logischen und den physikalischen Bildschirm, dann macht der Rechner womöglich Ausgaben, die für den Benutzer überhaupt nicht sichtbar werden, weil der Shifter brav den physikalischen Bildschirm abbildet.

Für diese scheinbar sinnlos versteckte Ausgabe gibt es durchaus genügend Anwendungsmöglichkeiten, z.B. im Bereich der bewegten Computergrafik, also der Computer-Animation. Stellen Sie sich z.B. ein kleines Strichmännchen vor, das sich von links nach rechts über den Bildschirm bewegen soll. Das läßt sich in einem Programm in einer Schleife lösen: Bildschirm löschen, Männchen zeichnen, Bildschirm löschen, Männchen etwas weiter rechts zeigen, Bildschirm löschen, Männchen noch weiter rechts zeigen...

Wenn Sie das machen, so ist die gestellte Aufgabe zwar gelöst, doch werden Sie vom Ergebnis wahrscheinlich enttäuscht sein: das Bild flackert, weil unser Auge nicht nur die Bewegung der Figur, sondern auch das ständige Löschen und Neuzeichnen registriert. Hier hilft uns aber die Methode mit den beiden getrennten Bildschirmspeichern weiter: Wir löschen also Bildschirm A und zeichnen das Männchen hinein; anschließend wird dieser Bildschirm zum physikalischen Bildschirm ernannt - das Männchen erscheint auf dem Monitor. Sogleich definieren wir Bildschirmspeicher B als logischen Bildschirm und löschen diesen, ohne das Bild auf dem Monitor zu verändern (steht ja im Video-RAM A). Dann zeichnen wir noch das Männchen in den Speicher B, diesmal aber etwas weiter rechts. Schließlich sagen wir dem Shifter, daß ab sofort B der physikalische Bildschirmspeicher sein soll, und das Männchen erscheint in der zweiten Position auf dem Bildschirm. Bildschirm A wird nun zum logischen Bildschirm, und das Spiel beginnt von vorne.

Diesmal flackert das Bild nicht, weil der Anwender den leeren Bildschirm ohne Männchen nicht mehr sehen kann; dieses Bild existiert nur im logischen Bildschirmspeicher, nicht aber auf dem Monitor.

Prinzipiell läßt sich dieses Verfahren immer dann praktisch einsetzen, wenn der Rechner für den Bildaufbau viel Zeit benötigt, das Bild aber erst dann auf dem Monitor erscheinen soll, wenn es ganz fertig ist. Ein weiteres Beispiel wäre also eine Diashow, die mehrere Bilder nacheinander anzeigen soll, ohne ein neues Bild beim Laden Stück für Stück auf den Bildschirm zu bringen.

Um keine Mißverständnisse aufkommen zu lassen: Man könnte diese Probleme natürlich auch dadurch lösen, einfach einen logischen Bildschirmspeicher zu installieren und nach der Fertigstellung eines Bildes

den kompletten Inhalt dieses Speichers in das physikalische Video-RAM zu kopieren. Aber das ist nicht Sinn der Sache; wir wollen wirklich die Funktionen zweier Speicherbereiche (nämlich physikalischer bzw. logischer Bildschirm zu sein) austauschen.

Auf dem Atari gibt es mehrere Möglichkeiten, den Beginn des logischen und des physikalischen Bildschirmspeichers zu bestimmen. Die "legalste" Methode ist wohl die über die XBIOS-Routine Setscreen (Nummer 5). Sie bekommt von uns drei Parameter: zwei Langwörter, die die Startadresse des logischen/physikalischen Video-RAMs angeben, und ein Wort für die Bildschirmauflösung. Letztere sollten Sie allerdings nicht so ohne weiteres ändern... Übrigens, wenn Sie einen oder zwei der Werte nicht ändern wollen, so können Sie eine -1 übergeben. Bei den Langwörtern achten Sie aber bitte darauf, die -1 auch wirklich als 4-Byte-Wert zu übergeben. Sie können auch die Startadresse des logischen/physikalischen Bildschirmspeichers erfragen, und zwar mit den XBIOS-Funktionen Physbase (2) und Logbase (3).

Bevor wir zum Beispielprogramm kommen, habe ich leider eine schlechte Nachricht für GFA-BASIC-Programmierer: Der Interpreter weigerte sich strikt, Ausgaben auf unterschiedlichen logischen und physikalische Bildschirme bei allen Ausgaben zu akzeptieren. Einige Ausgabebefehl bezogen sich auf den logischen, andere auf den physikalischen Bildschirm. Anscheinend wurde das System manchmal aus Geschwindigkeitsgründen umgangen.

Das Programm soll einen kurzen Text über den Bildschirm bewegen lassen: "Hallo!" Das Wort soll sich von oben links nach unten rechts bewegen. Das Programm sollte nun nicht mehr schwer zu verstehen sein. Nur den Trick, wie man den zweiten Bildschirmspeicher an eine Pagegrenze bekommt, möchte ich Ihnen noch verraten: Dazu fordert man vom System einfach einen Speicherbereich an, der 256 ($100) Bytes größer ist als der tatsächlich benötigte. Innerhalb dieses Bereichs muß es dann zwangsläufig einen Bereich geben, der an einer Pagegrenze liegt und auch noch groß genug ist. Die Startadresse dieses Bereichs wird nun so verändert, daß die beiden letzten Ziffern (im Hexadezimalsystem geschrieben) 0 werden. Im Programm macht das die Verknüpfung AND $FFFFFF00. Weil die so erhaltene Adresse aber wahrscheinlich außerhalb unserer "Spielzone" liegt, addieren wir noch 256 ($100) und erhalten dabei natürlich wieder eine Pagegrenze (eine Page ist ja gerade 256 Bytes groß).

In Omikron-BASIC wird die Text-Ausgabe über die TEXT-Anweisung vorgenommen. In C mußte ich leider auf eine VDI-Funktion des GEM

zurückgreifen, die erst im Kapitel 5 erläutert wird. Ihr Name ist ebenfalls text. Sie benötigen für dieses Programm noch das Include-File GEM_INEX.C aus dem Ordner GEM der Diskette zum Buch; diese Datei wird ebenfalls im Kapitel 5 besprochen werden.

Die Grafik ist nun mit diesem Programm zwar flackerfrei, aber leider noch nicht ruckfrei. Das liegt daran, daß die Ausgabe von Text und das Löschen des ganzen Bildschirms ziemlich langsam ausgeführt werden. Deshalb habe ich in Assembler einmal in die Trickkiste gegriffen und die Text-Ausgabe "persönlich" vorgenommen. Trotzdem arbeitet auch dieses Programm natürlich mit zwei Bildschirmen. Das Assemblerprogramm arbeitet leider nur auf dem Monochrom-Monitor, ist dafür aber auch schnell und bewegt den Text mit 71 Bildern pro Sekunde auch wirklich ruckfrei. Zum Prinzip nur soviel: Die meiste Zeit wird bei der Ausgabe eines Textes gewöhnlich damit verbracht, die einzelnen Punkte der Buchstaben bitweise nach rechts oder links zu verschieben. Diese Arbeit macht mein Programm einmal zu Anfang, und kann dann später direkt auf 8 horizontal verschiedene Positionen des Strings zugreifen. Das ist aber wirklich eine Sache für Grafik-Profis; der BASIC-Fan ist mit seinem TEXT-Befehl besser bedient.

Spaßeshalber habe ich im Omikron-Programm zwei Zeilen innerhalb einer FOR-Schleife markiert, die Sie einmal entfernen sollten. Dann sehen Sie sehr deutlich den Unterschied zwischen nur-ruckend und flackernd-und-ruckend.

Omikron-BASIC

```
'
' Mehrere logische Bildschirme (flackerfreie Grafik)
' OMIKRON        MP 23-12-88          MULTISCR.BAS
'
CLEAR 33000'       Speicher an GEMDOS zurückgeben
'
DIM Scr(1)
'
' Adresse des Bildschirms erfragen.
' Platz für zweiten Bildschirm schaffen.
'
Scr(0)=FN Logbase
Free= MEMORY(32256)
'
IF Free=0 THEN
 FORM_ALERT (1,"[3][Zu wenig Speicher!][Schade!]")
ELSE
 '
```

```
' Scr(1) muß an Page-Grenze beginnen:
'
Scr(1)=(Free+256) AND $FFFF00
'
FOR I=20 TO 300
  Setphys(Scr(I MOD 2))'     *  Diese beiden Zeilen können Sie mal
  Setlog(Scr(1-(I MOD 2)))'  *  zum Spaß löschen; dann sehen Sie
  '                             den Unterschied!
  CLS
  TEXT I,I,"Hallo!"
  '
  XBIOS (,37)'  Warten auf Bildrücklauf
  '
NEXT I
'
GEMDOS (,7)'        Crawcin wartet auf Taste
'
Setlog(Scr(0))'     alten Zustand wiederherstellen
Setphys(Scr(0))
'
ENDIF
'
END
'
'
DEF FN Logbase
 XBIOS (Logbase,3)
RETURN (Logbase)
'
DEF PROC Setlog(X)
 XBIOS (,5, HIGH(X), LOW(X),-1,-1,-1)
RETURN
'
DEF PROC Setphys(X)
 XBIOS (,5,-1,-1, HIGH(X), LOW(X),-1)
RETURN
```

C

```
/*************************************************************/
/*   Mehrere logische Bildschirme (flackerfreie Grafik)   */
/*   Megamax Laser C      MP  23-12-88       MULTISCR.C   */
/*************************************************************/


#include <osbind.h>                  /* Betriebssystem-Definitionen */

#include "gem_inex.c"                      /* Erläutert im Kapitel 5 */

long screen1,
```

```
    screen2,
    memory,

    malloc ();


init_screens()
{
  screen1 = Logbase();                    /* Startadresse Video-RAM */
  if ((memory = malloc (32256)) == 0L)    /* Neuen Speicher anfordern */
  {
    printf ("Der Speicherplatz reicht nicht!\n");    /* Fehlermeldung */
    Crawcin();                                      /* Warten auf Taste */
    Pterm0();                                       /* Programm beenden */
  }
  screen2 = (memory & 0xffffff00) + 256;  /* muß an Pagegrenze liegen */
}


swap_screens()                                /* Umschalten: screen1/2 */
{
  if (Logbase () == screen1)
    Setscreen (screen2, screen1, -1);
  else
    Setscreen (screen1, screen2, -1);
}


main()
{
int i;
  init_screens ();         /* Initialisierung des zweiten Bildschirms */

  gem_init();                /* Steht in der Include-Datei GEM_INEX.C */

  v_hide_c (handle);                        /* Mauszeiger ausschalten */

  for (i=20; i<=300; i++)
  {
     Cconws ("\33E");                          /* Bildschirm löschen */
     v_gtext (handle, i, i, "Hallo!");            /* GEM-Text-Ausgabe */
     swap_screens ();              /* Wechsel log./phys. Bildschirme */

     Vsync();                                /* Warten auf Bildrücklauf */
  }

  Crawcin();                                 /* Warten auf Tastendruck */

  Setscreen (screen1, screen1, -1);       /* Bildschirm wieder normal */
  v_show_c (handle, 1);                                /* Mauszeiger an */
```

```
  gem_exit();
}
```

Assembler

```
( nur für Monochrom-Monitor)
;
; Arbeiten mit zwei logischen Bildschirmen (flackerfreie Grafik)
; Assembler                MP  23-12-88                MULTISCR.Q
;

gemdos      = 1
xbios       = 14
logbase     = 3
setscreen   = 5
cconws      = 9
vsync       = 37


            TEXT

            DC.w $a00a          ;Mauszeiger abschalten

            pea     clrscr      ;Bildschirm löschen
            move.w  #cconws,-(sp)
            trap    #gemdos
            addq.l  #6,sp

            move.w  #logbase,-(sp)     ;Bildschirmadresse holen
            trap    #xbios
            addq.l  #2,sp
            movea.l d0,a6
            move.l  d0,savescr  ;und retten

            move.l  #freemem,d0 ;2. Bildschirmspeicher löschen
            addi.l  #256,d0     ;muß an Pagegrenze beginnen
            andi.l  #$ffffff00,d0
            movea.l d0,a4

            lea     (a4),a3
            move.w  #32000/4-1,d0      ;Bildschirm löschen
clrloop:    clr.l   (a3)+
            dbra    d0,clrloop

; Die Zeichen, die ab char_pt definiert sind, müssen kopiert und
; horizontal verschoben werden:

            lea     char_pt,a0  ;Originalzeichen kopieren
            lea     characters,a1
```

```
            move.w  #16*8-1,d0
loop1:      move.b  (a0)+,(a1)+
            dbra    d0,loop1

            moveq.l #7-1,d0       ;7 Variationen bilden
            lea     characters,a0       ;'Original'-Zeile
loop2:      moveq.l #16-1,d2      ;16 Pixelzeilen pro Zeichenloop3:
            moveq.l #8-1,d1       ;8 Bytes pro Pixel-Zeile
loop4:      move.b  0(a0,d1.w),0(a1,d1.w)
            dbra    d1,loop4

; Diese neue Zeile um ein Pixel nach links schieben

            andi.b  #%11101111,ccr      ;X-Bit löschen
            roxr.w  (a1)          ;Bytes nach rechts schieben
            roxr.w  2(a1)         ;Überträge im X-Bit
            roxr.w  4(a1)
            roxr.w  6(a1)

            addq.l  #8,a0         ;nächste Pixelzeile
            addq.l  #8,a1
            dbra    d2,loop3
            dbra    d0,loop2

; Physikalischen und logischen Bildschirm angeben:

            move.w  #-1,-(sp)     ;Auflösung nicht ändern
            pea     (a4)          ;physikalisch
            pea     (a6)          ;logisch
            move.w  #setscreen,-(sp)
            trap    #xbios
            adda.l  #12,sp

; Text darstellen:

            move.w  #20,d2        ;Anfangs x- und y-Koordinate

mainloop:   lea     (a6),a5
            lea     characters,a0

; d2*80 ist Offset für Zeile / x*80 entspricht (x*4+1)*16

            clr.l   d3
            move.w  d2,d3
            asl.w   #2,d3         ;*4
            add.w   d2,d3         ;+1
            asl.w   #4,d3         ;*16
            adda.l  d3,a5         ;Startadresse der Bildschirmzeile
```

```
; Horizontal-Variation: d2 mod 8 (=d2 and 7)

            move.w  d2,d3
            andi.w  #7,d3

; *128 ist Offset auf Charcter-Tabelle

            asl.w   #7,d3       ;*128
            adda.l  d3,a0       ;Startadresse des Charactersatzes

; Spalte: d2 div 8

            move.w  d2,d3
            asr.w   #3,d3
            adda.l  d3,a5       ;Adresse im Video-RAM

; Schreiben der Zeichen:

            moveq.l #16-1,d1    ;16 Pixelzeilen
lop0:       moveq.l #8-1,d0
lop1:       move.b  0(a0,d0.w),0(a5,d0.w)
            dbra    d0,lop1
            adda.l  #80,a5      ;nächste Bildschirmzeile
            addq.l  #8,a0       ;nächste Characterzeile
            dbra    d1,lop0

; Umschalten der Bildschirme:

            exg.l   a4,a6
            move.w  #-1,-(sp)   ;Auflösung nicht ändern
            pea     (a4)        ;physikalisch
            pea     (a6)        ;logisch
            move.w  #setscreen,-(sp)
            trap    #xbios
            adda.l  #12,sp

            move.w  #vsync,-(sp) ;Warten auf Bildrücklauf
            trap    #xbios
            addq.l  #2,sp

            addq.w  #1,d2       ;nächste Position
            cmpi.w  #300,d2     ;Ende?
            ble     mainloop

            move.w  #7,-(sp)
            trap    #gemdos
            addq.l  #6,sp

            move.w  #-1,-(sp)   ;Auflösung nicht ändern
            move.l  savescr,-(sp)      ;alten Zustand wiederherstellen
```

```
                move.l  savescr,-(sp)
                move.w  #setscreen,-(sp)
                trap    #xbios
                adda.l  #12,sp


                DC.w $a00a          ;Mauszeiger wieder einschalten

                clr.w   -(sp)
                trap    #gemdos


                DATA

clrscr:         DC.b 27,'E',0

char_pt:        DC.b 0,0,0,0,0,0,0,0
                DC.b 0,0,0,0,0,0,0,0
                DC.b 0,102,0,56,56,0,24,0 ;Bildschirm-Codes für
                DC.b 0,102,0,56,56,0,24,0 ;den String 'Hallo!'
                DC.b 0,102,0,24,24,0,24,0
                DC.b 0,102,60,24,24,60,24,0
                DC.b 0,126,62,24,24,126,24,0
                DC.b 0,126,6,24,24,102,24,0
                DC.b 0,102,62,24,24,102,24,0
                DC.b 0,102,126,24,24,102,24,0
                DC.b 0,102,102,24,24,102,0,0
                DC.b 0,102,102,24,24,102,0,0
                DC.b 0,102,126,60,60,126,24,0
                DC.b 0,102,62,60,60,60,24,0
                DC.b 0,0,0,0,0,0,0,0
                DC.b 0,0,0,0,0,0,0,0


                BSS

savescr:        DS.l 1

freemem:        DS.b 32256          ;Platz für zweiten Bildschirm

; Platz für 8 Buchstaben mit je 12 Bytes
; in 8 verschiedenen horizontalen Ausrichtungen:

characters:     DS.b 8*16*8

                END
```

2.2 Sprites

Sprites sind grafische Objekte, die sich programmgesteuert über den Bildschirm bewegen können. Gewöhnlich werden Sprites von der Hardware, also vom Video-Chip eines Computers in das normale Bild eingeblendet. Der Shifter des Atari verfügt über diese Möglichkeit jedoch nicht, so daß die Sprites per Software auf den Bildschirm kommen und auch wieder verschwinden. Gewöhnlich nennt man solche Software-Sprites Shapes. Nun, beim Atari heißen sie weiterhin Sprites, und deshalb will ich mich an diesen Namen halten.

Ein Sprite wird immer dann benötigt, wenn kleine Objekte auf dem Bildschirm bewegt werden sollen. Bestes Beispiel: Der Mauszeiger ist ein solches Grafik-Objekt, das zwar nicht für ein Anwendungsprogramm, aber doch für das Betriebssystem nichts weiter als ein Sprite ist.

Jedes Sprite besteht auf dem Atari aus einer Matrix von 16*16 Punkten. Diese Größe ist von der Bildschirmauflösung unabhängig. Deshalb ist ein einzelnes Sprite, besonders bei Verwendung eines monochromen Monitors in der höchsten Auflösung, meist klein. Deshalb werden größere Objekte in mehrere neben- oder übereinanderliegende Sprites aufgeteilt, die koordiniert bewegt werden. Auf dem Bildschirm sehen sie nachher wie ein normales kleines Sprite aus.

Wenn sich so ein Sprite bewegen soll, dann wird es vom Programm gezeichnet, kurz danach wieder gelöscht und sofort an einer neuen Position erneut gezeichnet usw., bis der Zielpunkt erreicht ist. Wenn das Sprite gelöscht wird, dann ist es natürlich erforderlich, daß die darunterliegende Fläche nicht einfach weiß erscheint, sondern ihren alten Inhalt wiedererhält. Deshalb werden wir einen Speicherplatz bereithalten müssen, in dem der Hintergrund vor dem Zeichnen des Sprites gerettet werden kann. Zum Löschen des Objekts muß dann nur der gerettete Bereich wieder auf den Bildschirm zurück kopiert werden.

Dann benötigen wir noch einen Aktionspunkt für das Sprite. Darunter ist folgendes zu verstehen: Wenn das Sprite auf dem Bildschirm erscheinen soll, so müssen Sie als Programmierer natürlich die Koordinaten des Punktes angeben, an dem das Objekt gewünscht wird. Aber was heißt hier Punkt: Ein Sprite ist schließlich eine Fläche, und die soll auf einem Punkt liegen? Eben dazu gibt es den Aktionspunkt. Der gibt die Koordinate relativ zur linken oberen Ecke des Sprites an, auf den sich alle späteren Bildschirmkoordinaten beziehen. Diese Aktionspunkt-Koordinaten dürfen Werte zwischen 0 und 15 annehmen. Ein Aktionspunkt von 8,8 besagt also, daß die Mitte des Sprites der Bezugspunkt für das Sprite sein

soll. Zeichnen wir später an die Koordinate 100,100 dieses Sprite, dann wird genau an dieser Stelle des Bildschirms der Mittelpunkt des Objekts erscheinen.

Das Wichtigste fehlt noch: Wie soll es denn überhaupt aussehen, unser Sprite? Wir müssen das 16*16-Raster definieren. Das machen wir mit 16 Zahlen von je 16 Bit Breite. Jede Zeile des Rasters entspricht einer der 16 Zahlen, und jeder Punkt innerhalb einer Zeile entspricht einem Bit innerhalb der entsprechenden Zahl. Dabei ist das höchstwertige Bit (Nr. 15) der linke Punkt einer Zeile, das Bit 0 der rechte Punkt einer Zeile. Ein gesetztes Bit heißt: Punkt an. Da das Konstruieren eines Sprites mit diesen Zahlen aber nicht sehr angenehm und anfällig für Fehler ist, definieren wir lieber ein paar Strings im Programm mit Sternchen und Strichen. Die entsprechenden Zahlenwerte errechnet dann ein Unterprogramm zur Laufzeit. In Assembler können wir diese Werte auch im DATA-Segment als Konstanten ablegen und die Zahlen direkt im Binärsystem schreiben; dazu ist den Ziffern lediglich ein Prozentzeichen voranzustellen.

Zu diesen Sprite-Daten benötigen wir aber noch eine Maske, die ebenfalls in einem 16*16-Raster untergebracht wird. Diese ist erforderlich, wenn ein solches Sprite über einem Hintergrund gezeichnet wird: Alle Punkte des Hintergrundes, deren Bits in der Maske gesetzt sind, werden vor dem Zeichnen des eigentlichen Sprites gelöscht. Wenn die Maske immer etwas größer ist als das Sprite selbst, dann erscheint also selbst auf einem ganz schwarzen Hintergrund immer noch ein dünner Rand um das Objekt; andernfalls könnten Sie es ja gar nicht erkennen. Die Figur, die eine solche Maske darstellt, ist gewöhnlich ganz ausgefüllt, damit man später nicht nur die Umrisse, sondern auch das Muster des Sprites oder ein Bild erkennen kann.

Schließlich müssen Sie noch Vorder- und Hintergrundfarbe festlegen. Die Hintergrundfarbe ist die Farbe, mit der die Maske auf den Bildschirm geschrieben wird. Gewöhnlich wird man hierfür den Farbindex 0 wählen, um den Maskenbereich zu löschen. Die Vordergrundfarbe ist dann entsprechend die Farbe, mit der das eigentliche Sprite gezeichnet wird. Zuletzt ist noch die Angabe eines Modus-Wortes erforderlich. Man unterscheidet zwischen dem normalen Modus, auch VDI-Modus genannt (Modus-Wort 0) und dem XOR-Modus (1). Der Unterschied liegt in der Behandlung eines Sonderfalls: Wenn ein Bit im eigentlichen Sprite gesetzt, in der Maske aber gelöscht wird, dann wird im VDI-Modus die Vordergrundfarbe geschrieben, während der Bildschirmpunkt im XOR-Modus logisch mit der Hintergrundfarbe XOR-verknüpft wird. Fragen Sie mich nicht, wozu das gut sein sollte (die Hintergrundfarbe ist gewöhnlich 0);

eine sinnvolle Anwendung dieser Unterscheidung wäre höchstens im Farbbetrieb des Rechners denkbar. Wir arbeiten deshalb im VDI-Modus.

Die letzte Frage betrifft die Größe des Speicherbereichs, in dem der Hintergrund eines Sprites gerettet werden soll. In den müssen auch 16 Zeilen des Bildschirms (die Höhe eines Sprites) passen, aber nicht nur 16, sondern 32 Bildpunkte für jede Zeile. Aus Geschwindigkeitsgründen wird nämlich immer gleich ein Langwort pro Zeile gerettet, was umständliches und langsames Bitgeschiebe vermeidet. Allerdings müssen Sie auch noch beachten, daß im Farbbetrieb auf jeden Bildpunkt nicht mehr nur ein Bit im Speicher kommt; statt dessen benötigt der Rechner jetzt zwei (mittlere Auflösung) oder sogar vier Bits (niedrige Auflösung), um einen Bildpunkt mit seiner Farbinformation darzustellen. Und all diese Bits müssen selbstverständlich gerettet werden. Deshalb genügen in der hohen Auflösung 64 Bytes (16 Langworte) als Rettungsspeicher, während es in der mittleren Auflösung schon 128 Bytes und in der niedrigen gar 256 Bytes sein müssen. In C und Assembler müssen Sie dazu noch einmal 10 Bytes addieren, die für die interne Verwaltung benötigt werden.

So, das waren die allgemeinen Informationen. Jetzt betrachten wir noch, was es in den einzelnen Programmiersprachen zu beachten gibt.

GFA-BASIC

In GFA-BASIC müssen Sie alle Informationen über ein Sprite in einen String packen, der der Anweisung Sprite zu übergeben ist. Maske und Sprite-Daten werden 5 Angaben vorangestellt, die Sie am besten mit der Funktion MKI$ erzeugen, die aus einem Wort einen Zwei-Byte-String macht.

Der Sprite-Definitions-String ist folgendermaßen aufgebaut:

X-Koordinate des Aktionspunktes
Y-Koordinate des Aktionspunktes
Modus (0=VDI, 1=XOR)
Farbe der Maske (gewöhnlich 0=Hintergrund)
Farbe des Sprite-Vordergrundes

Daran schließen sich, immer abwechselnd, die je 16 Worte der Maske und des Sprites an. Insgesamt kommen wir also auf 37 Worte bzw. in BASIC auf einen String von 37*2=74 Bytes Länge.

Die Angabe eines Rettungsspeichers für den Hintergrund ist nicht erforderlich, das macht der Interpreter für uns. Wenn Sie das Sprite anzeigen

lassen wollen, dann geben Sie noch eine x- und eine y-Koordinate nach dem Definitions-String an; wenn Sie es aber löschen wollen, so lassen Sie die Koordinaten weg.

Omikron-BASIC

Omikron-BASIC bietet eine leicht abgewandelte Form der Sprite-Programmierung. Hier gibt es zwei Sprite-Größen (groß und klein), die jedoch in den verschiedenen Auflösungsstufen unterschiedlich sind. Ich empfehle Ihnen dringend, einmal im Handbuch zum BASIC unter Sprite nachzuschlagen. Grundsätzlich gelten die oben angegebenen Informationen jedoch auch für Omikron.

C

Im Standard-C gibt es keine Sprites, doch im Megamax Laser C sind sie enthalten. Dort gibt es eine Line-A-Bibliothek mit einem entsprechenden Header-File. Line-A bezeichnet Grafik-Routinen, die auf einer sehr niedrigen Ebene im System angeordnet sind. Darunter befinden sich auch zwei Routinen für Sprites: a_drawsprite und a_undrawsprite. Unter anderem bietet das Header-File auch einen struct namens sprite (kleingeschrieben!) an, der alle Informationen über ein Sprite aufnimmt und an die Line-A-Routinen übergeben werden kann. Bei der mir vorliegenden Version waren jedoch die Einträge für die Vordergrund- und die Hintergrundfarbe vertauscht (Fehler im Header-File). Das können Sie leicht selbst überprüfen: Im Header-File sollte der Eintrag der Hintergrundfarbe vor der Sprite-Farbe (Vordergrund) stehen; bei mir war es andersherum. Ansonsten entspricht dieser struct genau den Daten, die schon unter GFA-BASIC als Sprite-Definitions-String beschrieben wurden.

Um ein Rechteck auf den Bildschirm zu bringen, mußte ich (wie schon im vorigen Abschnitt 2.1) Gebrauch von ein paar VDI-Funktionen des GEM machen. Dazu gehört auch das Include-File GEM_INEX.C, das Sie im Ordner GEM.5 auf der Diskette im Buch finden. Alle Erklärungen dazu und zu den verwendeten Funktionen in Kapitel 5.

Assembler

Hier geht im Prinzip alles so wie in C,nur können wir die beiden Routinen drawsprite und undrawsprite nicht über Namen aufrufen. Statt dessen benutzen wir den sogenannten Line-A-Emulator. Der 68000 hat nämlich die Eigenschaft, bei einem Opcode, dessen ersten vier Bits die Hex-Ziffer $A ergeben (also bei allen Codes $Axxx), eine Ausnahmebehandlung auszulösen und dem Betriebssystem die Kontrolle zu übergeben.

Dieses prüft nun das untere Byte dieses Opcdes, mit dem es zwischen 16 verschiedenen grafischen Operationen unterscheidet. Darunter sind nun auch unsere beiden Routinen: drawsprite wird über den Opcode $A00D aufgerufen, undrawsprite über $A00C. Parameter: Bei undrawsprite ist die Adresse des Hintergrund-Sicherungsbereichs in A2 zu übergeben. Bei drawsprite sind zusätzlich in d0 und d1 die Koordinaten des Punktes anzugeben, an dem das Sprite erscheinen soll, sowie in a0 die Adresse eines Sprite-Definitions-Blocks, der genauso aussieht wie unter GFA-BASIC bechrieben.

Nach soviel Theorie nun endlich ein Beispielprogramm. Es zeichnet ein Rechteck auf den Bildschirm, um die Wirkung der Maske zu demonstrieren. Dann bewegt sich ein Sprite von links nach rechts über den Bildschirm. Das Sprite soll eine Diskette darstellen:

GFA-BASIC

```
'
' Sprites                SPRITE.GFA
' GFA-BASIC              MP 27-12-88
'
' Einlesen der Sprite-Daten aus Data-Zeilen:
'
READ x_aktion,y_aktion,modus,maskenfarbe,spritefarbe
'
sprite$=MKI$(x_aktion)+MKI$(y_aktion)+MKI$(modus)
sprite$=sprite$+MKI$(maskenfarbe)+MKI$(spritefarbe)
'
FOR i=1 TO 16
  READ m$,s$     ! Je ein Wort für Maske und Sprite
  sprite$=sprite$+MKI$(FN zahl(m$))+MKI$(FN zahl(s$))
NEXT i
'
' Jetzt soll sich das Sprite über den Bildschirm bewegen:
'
PRBOX 140,100,500,200    ! Rahmen als Hintergrund zeichnen
'
FOR x=20 TO 620
  SPRITE sprite$,x,150
  VSYNC   ! Nächste Zeichnung beim Bildrücklauf, da stört's niemanden
NEXT x
'
' Sprite vom Bildschirm entfernen:
'
SPRITE sprite$
'
'
' Sprite-Daten:
```

```
'
DATA 8,8,0,0,1
'
DATA ****************,----------------
DATA ****************,-**************-
DATA ****************,-*--*------*--*-
DATA ****************,-*--*------*--*-
DATA ****************,-*--*------*--*-
DATA ****************,-*--********--*-
DATA ****************,-*------------*-
DATA ****************,-*------------*-
DATA ****************,-*-**********-*-
DATA ****************,-*-*--------*-*-
DATA ****************,-*-*-*-***--*-*-
DATA ****************,-***--------*-*-
DATA ****************,-***--**-*--*-*-
DATA ****************,-*-*--------*-*-
DATA ****************,-**************-
DATA ****************,----------------
'
'
FUNCTION zahl(a$)   ! Macht aus 16 Nullen/Einsen eine Zahl
  LOCAL i,z
  z=0
  FOR i=0 TO 15
    IF MID$(a$,16-i,1)<>"-"  !  'Bit' gesetzt?
      ADD z,2^i              !  Dann entsprechenden Wert addieren
    ENDIF
  NEXT i
  RETURN z
ENDFUNC
'
```

Omikron-BASIC

```
'
' Sprites                SPRITE.BAS
' Omikron               MP 27-12-88
'
' Einlesen der Sprite-Daten aus Data-Zeilen:
'
FOR I=1 TO 16
 READ M$,S$'      Je ein Wort für Maske und Sprite
 Sprite$=Sprite$+FN Zahl$(M$)+FN Zahl$(S$)
NEXT I
'
DEF SPRITE 1,5, MEMORY(64)'    Nummer 1, Typ 5
'
' Jetzt soll sich das Sprite über den Bildschirm bewegen:
'
```

```
PRBOX 140,100 TO 500,200'       Rahmen als Hintergrund zeichnen'
FOR X=20 TO 620
 SPRITE 1,X,150, LPEEK( VARPTR(Sprite$))+ LPEEK( SEGPTR +28),0,1 XBIOS (,37)'
Nächste Zeichnung beim Bildrücklauf, da stört's niemanden
NEXT X
'
' Sprite vom Bildschirm entfernen:
'
DEF SPRITE 1,0
'
END
'
'
' Sprite-Daten:
'
DATA "****************","----------------"
DATA "****************","-**************-"
DATA "****************","-*--*------*--*-"
DATA "****************","-*--*------*--*-"
DATA "****************","-*--*------*--*-"
DATA "****************","-*--********--*-"
DATA "****************","-*------------*-"
DATA "****************","-*------------*-"
DATA "****************","-*-**********-*-"
DATA "****************","-*-*--------*-*-"
DATA "****************","-*-*-*-***--*-*-"
DATA "****************","-***--------*-*-"
DATA "****************","-***--**-*--*-*-"
DATA "****************","-*-*--------*-*-"
DATA "****************","-**************-"
DATA "****************","----------------"
'
'
DEF FN Zahl$(A$)'     Macht aus 16 Nullen/Einsen einen String
 LOCAL I,Z
 Z=0
 FOR I=0 TO 15
   IF MID$(A$,16-I,1)<>"-"'  'Bit' gesetzt?
     THEN Z=Z+2^I'       Dann entsprechenden Wert addieren
   ENDIF
 NEXT I
RETURN RIGHT$( MKIL$(Z),2)
```

C

```
/************************************/
/*   Sprites             SPRITE.C   */
/*   Laser C         MP  27-12-88   */
/************************************/
```

```
#include <linea.h>

#include "gem_inex.c"   /* Siehe Beschreibung im Buch */

char  *daten[] = {
  "****************","*--------------",
  "****************","*--------------*",
  "****************","*--*------*--*-*",
  "****************","*--*------*--*-*",
  "****************","*--*------*--*-*",
  "****************","*--********--*-*",
  "****************","*-*----------*-*",
  "****************","*-*----------*-*",
  "****************","*-*----------*-*",
  "****************","*-*-*--------*-*",
  "****************","*-*-*-*-***--*-*",
  "****************","*-***--------*-*",
  "****************","*-***--**-*--*-*",
  "****************","*-*-*--------*-*",
  "****************","*--------------*",
  "****************","*--------------"
} ;

int  sprite_data[32];
int  i;
int  pxyarray[4];
int  save_area[74];   /* (alles für Mono-Monitor) */
sprite sprite_info;   /* struct aus linea.h */


get_sprite_data()
{
int i;

  for (i=0; i<32; i++)
    sprite_data[i] = bin (daten[i]);
}


int bin (zeile)
char *zeile;
{
int i, z;

  z=0;

  for (i=0; i<16; i++)
    z = ((z << 1) | ((*(zeile+i) == '*') & 1));
```

```
  return (z);
}


main()
{
  gem_init();  /* Siehe Beschreibung... */

  v_hide_c (handle);  /* Mauszeiger ausschalten */
  v_clrwk (handle);   /* Bildschirm löschen */

  pxyarray[0] = 140;  pxyarray[1]= 50;     /* Rechteck zeichnen */
  pxyarray[2] = 500;  pxyarray[3]= 150;
  vsf_interior (handle, 1);    /* ganz ausfüllen */
  v_rfbox (handle, pxyarray);  /* "Rundes Rechteck" zeichnen */

  v_show_c (handle, 1);       /* Mauszeiger wieder zeigen */

  get_sprite_data();

  sprite_info.x = 8;                /* Aktionspunkt */
  sprite_info.y = 8;
  sprite_info.format = 0;           /* Normales Sprite */
  sprite_info.forecolor = 0;        /* Hintergrundfarbe */
  sprite_info.backcolor = 1;        /* Sprite-Farbe */

  for (i=0; i<32; i++)
    sprite_info.image[i] = sprite_data[i];   /* Maske und Sprite */

  for (i=20; i<620; i++)
  {
    if (i>20) a_undrawsprite (save_area);
    a_drawsprite (i, 100, &sprite_info, save_area);
    xbios (37);   /* Bildrücklauf abwarten */
  }

  a_undrawsprite (save_area);

  gem_exit();
}
```

Assembler

```
;
; Sprites          SPRITES.Q
; Assembler     MP  27-12-88
;
```

```
gemdos      = 1
xbios       = 14
vsync       = 37

            INCLUDE 'GEM_INEX.Q'

            TEXT

main:       jsr     gem_init      ;Wir brauchen GEM für ein Rechteck

            move.w  #123,contrl   ;v_hide_c, versteckt Mauszeiger
            clr.w   contrl+2
            clr.w   contrl+4
            clr.w   contrl+6
            clr.w   contrl+8
            move.w  handle,contrl+12
            jsr     vdi

            move.w  #3,contrl     ;VDI Clear Workstation
            clr.w   contrl+2      ;Löscht den Bildschirm
            clr.w   contrl+4      ;(näheres in Kapitel 5)
            clr.w   contrl+6
            clr.w   contrl+8
            move.w  handle,contrl+12
            jsr     vdi

            move.w  #23,contrl    ;vsf_interior (setzt Füllmuster)
            clr.w   contrl+2
            clr.w   contrl+4
            move.w  #1,contrl+6
            move.w  #1,contrl+8
            move.w  #1,intin      ;ganz ausfüllen mit Vordergrundfarbe
            jsr     vdi

            move.w  #11,contrl    ;Gefülltes, rundes Rechteck
            move.w  #2,contrl+2   ;(v_rfbox)
            clr.w   contrl+4
            clr.w   contrl+6
            clr.w   contrl+8
            move.w  #9,contrl+10
            move.w  handle,contrl+12
            move.w  #140,ptsin    ;Koordinaten
            move.w  #50,ptsin+2
            move.w  #500,ptsin+4
            move.w  #150,ptsin+6
            jsr     vdi
```

```
            move.w  #122,contrl  ;v_show_c, Mauszeiger wieder zeigen
            clr.w   contrl+2
            clr.w   contrl+4
            move.w  #1,contrl+6
            clr.w   contrl+8
            move.w  handle,contrl+12
            move.w  #1,intin
            jsr     vdi

            moveq.l #20,d5       ;Laufvariable
            bra     entry

loop:       lea     save_area,a2
            DC.w    $a00c        ;Undraw Sprite -> Sprite löschen

            cmpi.w  #620,d5      ;rechts angekommen?
            beq     ende
            addq.w  #1,d5        ;nein, dann weiter

entry:      move.w  d5,d0        ;x-Koordinate
            move.w  #100,d1      ;y-Koordinate
            lea     sprite,a0
            lea     save_area,a2
            DC.w    $a00d        ;Draw Sprite

            move.w  #vsync,-(sp) ;Warten auf Bildrücklauf
            trap    #xbios
            addq.l  #2,sp

            bra     loop

ende:       jsr     gem_exit

            clr.w   -(sp)
            trap    #gemdos


            DATA

sprite:     DC.w 8,8             ;Aktionspunkt
            DC.w 0 ;normales Format
            DC.w 0,1             ;Hinter-/Vordergrundfarbe
```

```
            DC.w %1111111111111111,%0000000000000000
            DC.w %1111111111111111,%0111111111111110
            DC.w %1111111111111111,%0100100000010010
            DC.w %1111111111111111,%0100100000010010
            DC.w %1111111111111111,%0100100000010010
            DC.w %1111111111111111,%0100111111110010
            DC.w %1111111111111111,%0100000000000010
            DC.w %1111111111111111,%0100000000000010
            DC.w %1111111111111111,%0101111111111010
            DC.w %1111111111111111,%0101000000001010
            DC.w %1111111111111111,%0101010111001010
            DC.w %1111111111111111,%0111000000001010
            DC.w %1111111111111111,%0111001101001010
            DC.w %1111111111111111,%0101000000001010
            DC.w %1111111111111111,%0111111111111110
            DC.w %1111111111111111,%0000000000000000

            BSS

save_area:  DS.w 32             ;Platz, um den Hintergrund zu retten

            END
```

3. Interrupts

Wohl kaum ein Gebiet der Systemprogrammierung ist ähnlich reizvoll wie das der Interrupts. Es gibt eigentlich nichts in unserem Rechner, was ohne Interrupts funktionieren könnte. Gerade deshalb bieten sich mit Eingriffen in das Interrupt-System so viele Möglichkeiten, den Rechner zu beeinflussen.

Die schlechte Nachricht vorweg: BASIC-Fans werden auf Interrupts verzichten müssen; die Struktur der Sprache erlaubt es einfach nicht. Sie können höchstens auf Interrupt-ähnliche Routinen wie EVERY x GOSUB (GFA-BASIC) und ON TIMER x GOSUB (Omrikon-BASIC) zurückgreifen. Das gehört jedoch nicht zum Thema dieses Buches. Aber vielleicht besteht darin auch ein Anreiz, sich mit den maschinennahen Programmiersprachen (C und Assembler) zu beschäftigen. Und auch die C-Programmierer werden in Sachen Interrupts nicht bis zum Ende mithalten können; dieses Gebiet ist nun einmal eine Domäne der Assembler-Programmierung.

Nachdem ich Ihnen nun soviel vorgeschwärmt habe, sollten Sie jetzt erfahren, was Interrupts eigentlich sind. Als Interrupt bezeichnet man das Auslösen einer Unterbrechnung des Programms, das gerade in einem Rechner läuft. Dieser Auslöser kann entweder durch die Hardware hervorgerufen werden, oder das Programm leitet per Software-Befehl ganz bewußt eine Unterbrechung ein. Während die Unterbrechnung durch die Hardware dazu dient, das Betriebssystem auf wichtige oder kritische Ereignisse im Zusammenhang mit der Peripherie eines Rechners aufmerksam zu machen, wird die Ausnahme durch Software dazu benutzt, Routinen des Betriebssystems aufzurufen. Ein TRAP-Befehl ist nämlich nichts weiter als eine programmgesteuerte Ausnahme-Anforderung.

Bleiben wir aber zunächst bei den Hardware-Interrupts. Da unterscheidet man ganz grob drei Klassen: MFP, VBL und HBL. Dabei hat MFP die höchste Priorität, HBL die niedrigste (keine Angst, wir werden noch sehen, was das bedeutet). MFP ist die Abkürzung für Multi Functional Peripheral. Das ist ein Chip im ST, der 16 verschiedene Hardware-Interrupts überwachen kann und sie dem Prozessor meldet. Der 68000 alleine wäre nämlich nicht in der Lage, so viele verschiedene Interrupts zu unterscheiden. Welche Arten von Interrupts der MFP nun im einzelnen verwaltet, müssen wir später noch klären.

Betrachten wir zunächst noch die beiden anderen Hardware-Interrupts: VBL und HBL. Die Abkürzungen bedeutet Vertical Blank und Horizontal Blanc; damit bezeichnet man den Bildrücklauf (VBL) bzw. den Zeilenrücklauf (HBL) des Elektronenstrahls im Monitor. Das heißt im Klartext, daß jedesmal, wenn der Monitor ein komplettes Bild fertiggeschrieben hat, ein VBL-Interrupt ausgelöst wird, während ein HBL nach Fertigstellung jeder einzelnen Pixelzeile, also viel öfter, stattfindet.

Zwei Vektoren des Rechners sind nun für uns wichtig. Über sie springt nämlich der Rechner in eine Routine, die den Interrupt behandeln soll. Diese Routine heißt auch Ausnahmebehandlung. Bei jedem VBL springt der Rechner über den Vektor $70, beim HBL über den Vektor $68; in diesen Adressen stehen also die Startadressen der Interrupt-Routinen. Wenn Sie eigene Routinen zur Interruptbehandlung installieren wollen, so brauchen Sie also nur deren Startadressen in diese Vektoren zu schreiben; die Routinen selbst müssen mit dem Assembler-Befehl RTE (Return from Exception = Rückkehr von einer Ausnahmebehandlung) enden. Zum Verändern des Vektors ist es übrigens erforderlich, daß der Rechner sich im Supervisor-Modus befindet (notfalls mit GEMDOS-Funktion $20, Super, einschalten).

Nur: Was für Gründe gibt es überhaupt, eine Routine gerade dann aufrufen zu lassen, wenn der Elektronenstrahl gerade mal eine Zeile oder auch ein ganzes Bild zu Ende geschrieben hat? Meistens keine. Es ist nur so, daß in jedem Computersystem gewisse Aufgaben periodisch erledigt werden müssen; dazu sind Routinen des Betriebssystems in kurzen Zeitabständen immer wieder zu aktivieren. Das macht der VBL. Er ist eine Art Abfallprodukt des Video-Chips, der dem Rechner eine Uhr erspart; genausogut könnte ja auch per Stoppuhr (Timer) alle paar Hundertstelsekunden eine solche Routine aufgerufen werden. Der VBL liefert für diese Aufgabe eigentlich sehr gute Zeiten: Je nach angeschlossenem Monitor wird dieser Interrupt 71 (Monochrom-Monitor) oder 50 bis 60 (Farb-Monitor) Mal in der Sekunde ausgelöst.

Was den HBL betrifft, so kann ich Ihnen von dessen Benutzung nur sehr abraten. Er tritt furchtbar oft auf, und schon kurze Routinen zur Ausnahmebehandlung würden das ganze Computersystem stark bremsen (was natürlich auch beabsichtigt sein kann -> Software-Zeitlupe). Deshalb schaltet ihn das Betriebssystem auch ganz einfach ab; alle ankommenden HBLs werden vom Prozessor ignoriert. Also: Sie müssen den HBL eigentlich nie verwenden und sollten nur wissen, daß es ihn gibt und wann er auftritt.

3.1 Die VBL-Queue

Zurück zum VBL: Er ist bei allen Programmierern sehr beliebt, und damit bei der Benutzung des VBL-Vektors ($70) kein Streit zwischen den verschiedenen Programmen auftreten kann, verewigt sich das Betriebssystem beim Start des Rechners dort; es benutzt den VBL also für eigene Zwecke. Das heißt aber nicht, daß Sie als Programmierer auf diesen Interrupt verzichten müssen. Eine dieser System-Zwecke ist nämlich die Abarbeitung der sogenannten VBL-Queue (Queue = Schlange).

In dieser VBL-Queue befinden sich Startadressen von maximal 8 Routinen, die darauf warten, bei einem VBL-Interrupt aufgerufen zu werden, diesmal allerdings nicht direkt vom Prozessor über den Vektor $70, sondern von der VBL-Routine des Betriebssystems, die ihrerseits über den Vektor $70 aufgerufen wird. Jedes Programm darf sich in dieser Liste eintragen und wird fortan 50 bis 71mal in der Sekunde gestartet. Es dürfte klar sein, daß dadurch keine Textverarbeitung aufgerufen wird; auch ein Parkprogramm für die Harddisk (SHIP.PRG) 71mal in der Sekunde ausführen zu lassen wäre wenig sinnvoll. Aber Routinesachen wie das Blinken des Cursors (findet halt nicht bei jedem, sondern nur bei jedem dreißigsten VBL statt) oder die Anzeige einer Digitaluhr in der ersten Bildschirmzeile sind Dinge, für die der VBL-Interrupt durchaus geeignet ist. Zu weiteren Anwendungsbeispielen komme ich noch.

Für Sie ist es jetzt natürlich wichtig zu wissen, wie sich ein Programm in die Liste der VBL-Routinen eintragen kann. Dazu hat unser Rechner eine Liste mit gewöhnlich acht Langworten, in denen Startadressen von Routinen stehen dürfen. Die Routinen müssen mit einem RTS-Maschinenbefehl abgeschlossen sein und dürfen alle Register (Ausnahme: User-Stackpointer) benutzen. (C-Funktionen werden stets mit einem RTS beendet.) Außerdem dürfte es Sie interessieren, daß solche VBL-Queue-Routinen im Supervisor-Modus ablaufen; das gesamte System liegt Ihnen also ungeschützt zu Füßen.

Nun zur Liste der Startadressen: Hier sind gleich zwei Dinge variabel; denn die Liste kann an verschiedenen Adressen im Speicher beginnen und auch unterschiedliche Länge haben. Damit ist auch die Zahl der möglichen VBL-Interrupt-Routinen unterschiedlich. Um die Adresse der Liste und ihre Länge herauszufinden, gibt es im Atari zwei Systemvariablen. Das sind Adressen im Speicher, deren Bedeutung unabhängig von der TOS-Version ist. Diese Systemvariablen haben auch Namen. Für uns sind folgende wichtig:

Name	Adresse	Größe	Bedeutung
nvbls	$454	Wort	Zahl der möglichen VBL-Routinen.
_vbl_queue	$456	Langwort	Startadresse einer Liste mit Langworten, die die Adressen der VBL-Routinen enthalten.

Wir gehen nun folgendermaßen vor: Erst besorgen wir uns die Adresse, ab der die Startadressen der VBL-Routinen eingetragen sind. Dann überprüfen wir ab dieser Adresse die ersten nvbls Langworte. Ist ein Langwort Null, dann bedeutet das, daß dieser Eintrag noch frei ist; wir schreiben die Startadresse unserer eigenen Routine in dieses Langwort und beenden die Suche. Finden wir aber in den nvbls Langworten keine Null, dann ist die Liste voll; wir müssen die Suche erfolglos abbrechen. Übrigens: Auf die Systemvariablen kann ein Programm nur zugreifen, wenn es sich im Supervisor-Modus befindet. Das Programm kann mit den Funktionen Super (GEMDOS, Nr. $20) und Supexec (XBIOS, 38) in diese Betriebsart umschalten.

Wenn wir uns erfolgreich in die Liste eingetragen haben, dann müssen wir das Initialisierungsprogramm beenden, damit der Desktop wieder erscheint und der Anwender normal weiterarbeiten kann; das Interrupt-Programm soll ja nur im Hintergrund mitlaufen. Allerdings darf das Programm dabei nicht aus dem Speicher entfernt werden, weil die Interrupt-Routine ja bei jedem VBL aufgerufen werden soll. Die übliche GEMDOS-Funktion Pterm0 (Nr. 0), die bis jetzt all unsere Programme beendet hat (auch in C, allerdings fügt der Compiler den Funktionsaufruf automatisch ein), hilft uns also nicht, weil unser Programm gelöscht und der belegte Speicherplatz wieder freigegeben würde - das nächste zu ladende Programm belegt also den gleichen Speicherplatz.Statt dessen müssen wir auf die Funktion Ptermres zurückgreifen (ebenfalls GEMDOS, Nr. $31). Das res steht für resident und bedeutet, daß das Programm zwar beendet, aber nicht gelöscht werden soll. Damit nicht das nächste Programm unsere Interrupt-Routine wieder löscht, müssen wir beim Aufruf von Ptermres noch angeben, wie viele Bytes ab Programmstart wir gerne reservieren möchten. Dieser Speicherplatz wird dann für alle später geladenen Programme tabu sein.

Dabei stellt sich natürlich die Frage, wie groß denn dieser Speicherbereich sein muß. In Assembler können wir den Bedarf leicht errechnen: Er setzt sich aus der Größe der Basepage (konstant 256 Bytes), dem TEXT-, DATA- und aus dem BSS-Segment zusammen, wobei die Größe der drei letzten Bestandteile aus der Basepage entnommen werden kann. Wie das geht, ist dem Assembler-Listing des folgenden Beispiel-Programms zu entnehmen. In C ist die Sache nicht so einfach; am besten ist es wohl, die

Summe aus Programmlänge und dem Bedarf für Strings und globale Variablen zu nehmen und diese Zahl großzügig aufzurunden. Probieren geht hier über studieren.

Kommen wir zum Beispielprogramm: Ein Programm soll sich in die VBL-Queue einhängen und resident im Speicher bleiben. Die VBL-Routine überprüft nun, ob der Benutzer die beiden Shift-Tasten und Alternate gedrückt hat. Wenn ja, so soll der Bildschirminhalt gerettet und ein Text ausgegeben werden. Drückt der Benutzer dann die Taste Control, so wird der Bildschirm wiederhergestellt und die Interrupt-Routine beendet. Während der Text auf dem Bildschirm steht, soll das Hauptprogramm nicht weiterlaufen. Der Text, den das Programm ausgeben soll, heißt: "Ihr Auftrag wird bearbeitet - Bitte etwas Geduld!" Das ist sehr praktisch, wenn man im Büro Schach spielt und der Chef kommt gerade herrein...

Bei der Programmierung dieses Programms gibt es allerdings noch ein ganz schwerwiegendes Problem: Die Benutzung von Systemfunktionen (GEMDOS, BIOS und XBIOS) aus einem Interrupt heraus. So ganz legal ist deren Verwendung in einer VBL-Routine eigentlich nie. Man sagt zwar, daß BIOS und XBIOS-Routinen gefahrlos aufgerufen werden können, doch habe ich auch schon das Gegenteil feststellen müssen. Im Zweifelsfall sollten Sie also immer ausprobieren, wie der Rechner auf Ihre VBL-Routine reagiert, wenn diese Systemfunktionen benutzt.

Ich kann Ihnen aber einige Spielregeln geben, bei deren Beachtung ich bisher keine Probleme hatte. Zunächst: Auf keinen Fall bei jedem VBL Systemroutinen aufrufen! Wenn Ihr Programm also auf eine Tastenkombination warten muß, wie das in meinem Beispielprogramm der Fall ist, dann müssen Sie die BIOS-Funktion Kbshift (Nr. 11), die diese Aufgabe normalerweise übernimmt, umgehen; wie, das zeige ich gleich noch. Auf keinen Fall dürfen Sie auch mit Funktionen wie Cconin (GEMDOS) oder Bconin (BIOS) auf den Druck einer Taste warten; denn wenn dieser Funktionsaufruf ohne Bomben durchkommt, dann wartet die VBL-Routine wie befohlen auf einen Tastendruck; das Hauptprogramm steht still.

Wenn Sie nun also die gewünschte Tastenkombination vorfinden, dann ist alles erlaubt: GEMDOS, BIOS und XBIOS. Allerdings nur dann, wenn der Benutzer Ihr residentes Interrupt-Programm nicht gerade dann aufgerufen hat, wenn schon das Hauptprogramm in einer dieser Funktionen hing. (Das ganze Problem entsteht ja nur dadurch, daß nicht zwei Programme gleichzeitig Systemfunktionen benutzen dürfen.) Weil aber TOS-Programme in der Regel in GEMDOS- oder anderen Routinen hängen

bzw. auf diese warten, ist die Verwendung von TOS-Routinen in einer VBL-Routine immer dann verboten, wenn gerade ein TOS-Programm im Vordergrund läuft.

Anders bei GEM-Programmen: Immer, wenn ein GEM-Programm auf etwas wartet (Mausklick, Tastendruck...), können Sie aus dem VBL heraus beliebige TOS-Funktionen aufrufen. Und da fast alle Programme GEM-Applikationen sind und diese Programme meist auf eine Benutzeraktion warten, ist die Einschränkung bei den Systemaufrufen aus einem Interrupt-Programm eigentlich belanglos. Das einzige Problem ist wirklich das Erkennen des Aufrufs, also die Abfrage bestimmter Tasten.

Zu diesem Zweck müssen meist die Sondertasten wie Shift, Control und Alternate herhalten. Es ist möglich, daß ein Programm diese Tasten abfragt. Man erhält dann einen Wert zurück, dessen Bits folgende Bedeutung haben:

Bit	Taste
0	Shift (rechts)
1	Shift (links)
2	Control
3	Alternate

Die anderen Bits sind für unser Beispiel nicht interessant; bei der Abfrage im Programm werden sie durch eine bitweise AND-Verknüpfung ignoriert. Aber wie kommen wir überhaupt an den Wert? Nun, die eine Möglichkeit bietet die BIOS-Funktion Nr. 11 (Kbshift). Da wir aber keine TOS-Funktionen anwenden wollen, müssen wir die zweite Möglichkeit benutzen: Den direkten Zugriff über eine Adresse. Es gibt nämlich ein Byte (!!!) im Rechner, das genau den Zustand der vier Sondertasten anzeigt. Leider hängt die Adresse dieses Bytes von der Betriebssystem-Version ab: Bevor es das Blitter-TOS gab, konnte man den Tastenstatus an der Adresse $e1b auslesen. Mit dem Blitter-TOS änderte sich diese Adresse, und sie wird sich möglicherweise mit dem neuen TOS 1.4 abermals ändern. Doch gibt es dann einen Zeiger im ROM auf dieses Byte, und der steht an einer festgelegten Adresse: $fc0024. Die TOS-Version selbst entnehmen wir ebenfalls dem ROM: Steht an der Adresse $FC0002 (Wort) ein Wert größer als $100, dann haben wir es mindestens mit dem Blitter-TOS zu tun.

Um den Bildschirm zu retten, müssen wir natürlich die Startadresse des Video-RAM herausfinden. Wieder führen zwei Wege zum Ziel: die XBIOS-Funktionen Physbase und Logbase (2 und 3) und die Adresse $44e. Letztere ist eine Systemvariable, die auf das logische Video-RAM

(siehe auch 2.1) zeigt; ihre Lage im Speicher ist festgelegt, also von der TOS-Version nicht abhängig. Die eigentliche Text-Ausgabe wird übrigens mit der GEMDOS-Funktion Cconws (Nummer 9) vorgenommen.

Nun die Listings: Das C-Programm sieht schlimmer aus als es ist, weil ich sehr viel mit Pointern arbeiten mußte (das fällt in Assembler nicht so auf). Im C-Programm wird der Bildschirm in einem Array von 32000 Bytes (= 8000 Langworte) gerettet, und in Assembler sichern wir ihn im BSS-Segment. Wie ich schon sagte, ist es in C-Programmen nicht möglich, den Speicherbedarf genau zu bestimmen. Deshalb habe ich für das Programm mit dem 32000-Byte-Array großzügig 64 KByte reservieren lassen:

C

```
/*******************************************/
/*   Interrupt-Programm in der VBL-Queue   */
/*   Laser C    MP 30-12-88   VBLQUEUE.C   */
/*******************************************/

#include <osbind.h>

char *shiftptr;
int  *nvbls;
long *queue_ptr;
long *vbl_queue;

long save_screen[8000];   /* 32000 Bytes */

routine()       /* Funktion, die Interrupt-Routine sein soll */
{
int  i;
long *screen;

  if ((*shiftptr & 15) == 11)    /* Alt + 2x Shift? */
  {
    screen = (long *) 0x44e;     /* Zeiger auf Video-RAM */
    screen = (long *) *screen;

    for (i=0; i<8000; i++)       /* Bildschirm retten */
      save_screen[i] = *(screen+i);

    Cconws ("\33E\12\12");       /* ClrScr und 2 mal LineFeed */
    Cconws ("  Ihr Auftrag wird bearbeitet.\15\12\12");
    Cconws ("  Bitte etwas Geduld...");

    while ((*shiftptr & 15) != 4);   /* Warten auf Control */
```

```
    for (i=0; i<8000; i++)       /* Bildschirm zurückholen */
      *(screen+i) = save_screen[i];
  }
}


int init_vbl()
{
long *long_ptr,
     save_ssp;
int  i,
     *int_ptr,
     gefunden;

  save_ssp = Super (0L);  /* Supervisor einschalten */

  /* Adresse Sondertasten-Status ermitteln: */

  shiftptr = (char *) 0xe1bL;  /* Adresse bei altem TOS */
  int_ptr = (int *) 0xfc0002;  /* Welche TOS-Version? */
  if (*int_ptr > 0x100)        /* größer als altes TOS? */
  {
    long_ptr = (long *) 0xfc0024;  /* Hier steht die Adresse */
    shiftptr = (char *) *long_ptr;
  }

  /* Eintrag in die VBL-Queue */

  nvbls = (int *) 0x454L;
  queue_ptr = (long *) 0x456L;
  vbl_queue = (long *) *queue_ptr;

  i=0; gefunden=0;

  do
  {
    if (*(vbl_queue+i) == 0L)
    {
      *(vbl_queue+i) = (long) routine;  /* Routine installieren */
      gefunden = -1;
    }
    i++;
  }
  while (!gefunden && i < *nvbls);

  Super (save_ssp);       /* Wieder USER-Modus */
  return (gefunden);
}
```

```
main()
{
  if (init_vbl())
    Ptermres (65536L, 0);
}
```

Assembler

```
;
; Interrupt-Programm in der VBL-Queue (Zusatzbildschirm)
; Assembler            MP  30-12-88          VBLQUEUE.Q
;


gemdos      = 1
xbios       = 14
cconws      = 9
supexec     = 38
ptermres    = $31

_v_bas_ad   = $44e
nvbls       = $454
_vblqueue   = $456


            TEXT

            movea.l 4(sp),a0     ;Speicherbedarf ausrechnen
            move.l  #$100,d6     ;feste Größe der Basepage
            add.l   12(a0),d6    ;+ Größe des TEXT-Segments
            add.l   20(a0),d6    ;+ Größe des DATA-Segments
            add.l   28(a0),d6    ;+ Größe des BSS-Segments

            pea     init_vbl     ;Initialisierung muß im
            move.w  #supexec,-(sp)     ;Supervisor-Modus
            trap    #xbios       ;durchgeführt werden
            addq.l  #6,sp

            tst.w   err_flag     ;Initialisierung erfolgreich?
            beq.s   keep         ;ja, dann resident halten

            clr.w   -(sp)        ;sonst normal beenden
            trap    #gemdos

keep:       clr.w   -(sp)        ;keine Fehlermeldung
            move.l  d6,-(sp)     ;Speicherbedarf
            move.w  #ptermres,-(sp)    ;resident halten
            trap    #gemdos      ;Ende Hauptprogramm
```

```
; Installierung der Interrupt-Routine in der VBL-Queue:

init_vbl:     move.w  nvbls,d0      ;max. Zahl der VBL-Queue-Einträge
              lsl.w   #2,d0         ;* 4
              movea.l _vblqueue,a0  ;Liste der Startadressen
              clr.w   d1

vbl_search:   tst.l   0(a0,d1.w)    ;Eintrag überprüfen
              beq     vbl_found     ;0, dann ist der Eintrag frei
              addq.w  #4,d1         ;sonst nächstes Langwort überprüfen
              cmp.w   d0,d1         ;Ende der Liste?
              bne.s   vbl_search    ;nein, weitersuchen

              move.w  #-1,err_flag  ;Fehler merken
              rts                   ;und ins Hauptprogramm zurück

vbl_found:    bsr     get_shiftptr
              move.l  #routine,0(a0,d1.w)       ;Startadresse eintragen
              rts

get_shiftptr:
; Adresse für Sondertasten-Status ermitteln:

              cmpi.w  #$100,$fc0002     ;TOS-Version
              ble     altes_tos
              move.l  $fc0024,shiftptr  ;BlitterTOS: Adresse im ROM
              rts
altes_tos:    move.l  #$e1b,shiftptr    ;sonst feste Adresse
              rts

; Interrupt-Routine, wird ab sofort bei jedem VBL aufgerufen

routine:      movea.l shiftptr,a0  ;Sondertasten abfragen
              move.b  (a0),d0
              andi.b  #%1111,d0    ;nur Sondertasten, kein Caps Lock
              cmpi.b  #%1011,d0    ;Alternate und beide Shift-Tasten?
              bne     quit_vbl     ;nein, dann weiter im Hauptprogramm

; Ab hier legt das Hauptprogramm eine Pause ein...

              clr.l   d0           ;physikalischer Bildschirmstart
              move.b  $ffff8201,d0 ;High-Byte des Video-RAM
              lsl.w   #8,d0        ;* $100
              move.b  $ffff8203,d0 ;+ Mid-Byte des Video-RAM
              lsl.l   #8,d0        ;* $100
              move.l  d0,d7        ;merken für später

              movea.l d0,a6        ;Bildschirm retten
              lea     buffer,a1
```

```
            move.w  #32000/4-1,d0

save_loop:  move.l  (a6)+,(a1)+
            dbra    d0,save_loop

            pea     meldung       ;Text auf dem Bildschirm ausgeben
            move.w  #cconws,-(sp)
            trap    #gemdos
            addq.l  #6,sp

            movea.l shiftptr,a0   ;Warten auf Control
wait_loop:  move.b  (a0),d0
            andi.b  #%1111,d0
            cmpi.b  #%100,d0      ;Control?
            bne.s   wait_loop

            movea.l _v_bas_ad,a6
            lea     buffer,a1     ;Bildschirm wiederherstellen
            move.w  #32000/4-1,d0
rest_loop:  move.l  (a1)+,(a6)+
            dbra    d0,rest_loop

quit_vbl:   rts                   ;Ende VBL-Queue-Routine (kein RTE)


            DATA

err_flag:   DC.w 0

meldung:    DC.b 27,'E',10,10     ;Bildschirm löschen
            DC.b '  Ihr Auftrag wird bearbeitet.',13,10,10,10
            DC.b '  Bitte etwas Geduld...',0

            BSS

shiftptr:   DS.l 1
buffer:     DS.w 16000            ;Platz für den ganzen Bildschirm

            END
```

3.2 Der Timer A des MFP

Die VBL-Queue ist eine sehr praktische Einrichtung, die es vielen Programmen erlaubt, sich bequem in einen Interrupt einzuhängen. Der Nachteil des VBLs ist aber, daß er je nach angeschlossenem Monitor unterschiedlich oft auftritt (zwischen 50 und 71mal pro Sekunde). Außer-

dem kann es durchaus erforderlich sein, daß eine Routine in wesentlich kürzeren Zeitabständen aufgerufen wird. Diese Flexibilität kann der VBL nicht liefern.

Da müssen wir uns auf der nächsthöheren Interrupt-Ebene umschauen, nämlich beim MFP. Zur Erinnerung: Der MFP ist ein Chip, der alle Interrupts des Atari außer VBL und HBL verwaltet. Für den Prozessor ist dieser Baustein dem VBL und dem HBL übergeordnet, das heißt eine durch den MFP aktivierte Interrupt-Routine hat höchste Priorität und kann daher nicht durch einen VBL oder HBL unterbrochen werden.

Dieser MFP besitzt unter anderem auch vier programmierbare Timer. Die können Sie sich als Wecker vorstellen, die dem Prozessor in bestimmten Zeitabständen mitteilen, er möge dieses oder jenes Interrupt-Programm starten. Die Zeitabstände kann das Programm frei bestimmen. Von den insgesamt vier Timern, die der MFP besitzt, ist allerdings nur einer dafür gedacht, einem Anwenderprogramm als Wecker zu dienen. Es handelt sich um den Timer A. Die Timer C und D haben feste Aufgaben im System; der Timer B wird gewöhnlich für einen anderen Zweck eingesetzt, den wir noch kennenlernen werden.

Der MFP gehört mit zu den kompliziertesten Bausteinen im Atari ST. Er ist auch entsprechend schwierig zu programmieren. Aus diesem Grund gibt es auch im TOS eine Reihe von Routinen, die die einfache Verwendung der Timer ermöglichen sollen. Nur eine einzige XBIOS-Funktion benötigen wir, um den Timer in Gang zu setzen: Xbtimer (Nr. 31).

Es werden vier Parameter übergeben: Die Nummer des Timers, auf den sich der Funktionsaufruf bezieht, ein Kontrollregister, ein Datenregister und die Adresse der Routine, die beim Interrupt aufgerufen werden soll. Am einfachsten ist die Timer-Nummer zu erklären: Die Timer A bis D haben die Nummern 0 bis 3. Für unseren Timer A benötigen wir also die Null.

Das Kontrollregister hat gleich mehrere Aufgaben: Es schaltet den Timer ein und aus, es bestimmt die Betriebsart des Timers und regelt den Vorteiler. Wir machen uns die Sache etwas einfacher, als sie ist: Ist das Kontrollregister 0, so ist der Timer abgeschaltet. Betriebsarten kennen wir nicht. Aber mit dem Vorteiler müssen wir uns schon beschäftigen: Wir wollen ja in bestimmten Zeitabständen einen Interrupt auslösen. Dazu muß im MFP eine Uhr vorhanden sein. Diese existiert in Form eines Schwingquarzes, der am MFP angeschlossen und mit 2,4576 MHz getaktet ist. Jede Sekunde liefert dieser Quarz also 2.457.600 Impulse. Soviele Interrupts kann unser Prozessor aber gar nicht verarbeiten, und

deshalb gibt es den Vorteiler. Wenn wir den nämlich auf 1/64 stellen, dann beachtet der MFP nur jeden 64. Impuls des Quarzes. Die folgende Tabelle zeigt, welche Vorteilereinstellungen es gibt und durch welche Kontrollregister-Werte sie eingestellt werden können:

Vorteiler	Kontrollregister	Ergebnisfrequenz
1/4	1	614.400 Hz
1/10	2	245.760 Hz
1/16	3	153.600 Hz
1/50	4	49.152 Hz
1/64	5	38.400 Hz
1/100	6	24.576 Hz
1/200	7	12.288 Hz

Mit diesem Kontrollregister wäre es also schon möglich, die Zahl der Interrupts auf gut zwölftausend pro Sekunde zu begrenzen. Dagegen wirken selbst die 71 Hz des VBLs (im Monochrom-Modus) winzig. Und auch die 12.288 Interrupts werden Sie wahrscheinlich nie benötigen. Bedenken Sie: Interrupt-Routinen schlucken desto mehr Prozessorzeit, je öfter sie aufgerufen werden.

Nun, der Vorteiler hieße sicher nicht Vorteiler, wenn nicht noch ein anderer Teiler existierte, nämlich das Datenregister. Wenn wir da beispielsweise die Zahl 100 hineinschreiben, dann wird der eigentliche Interrupt erst bei jedem 100. Impuls des Vorteilers geliefert. Das Datenregister steuert also die zweite Teilerstufe. Der Wert dieses Registers, das eine Breite von nur 8 Bit aufweist, darf zwischen 1 und 256 liegen, wobei die 256 durch eine Null dargestellt wird. Damit können wir ab sofort Interrupts mit Aufruf-Frequenzen zwischen 48 Hz (Kontrollregister 7, Datenregister 0=256) und 61Z4.400 Hz (Kontrollregister 1, Datenregister 1) erzeugen.

Nun zur eigentlichen Interrupt-Routine. In der VBL-Queue wurde uns sämtlicher Verwaltungskram abgenommen; beim Timer müssen wir alles selbst machen. Aber so schlimm ist es gar nicht: Beim Eintritt in die Routine sind sämtliche Register zu retten, die wir möglicherweise verändern könnten. Bevor wir den Interrupt beenden, müssen diese Register natürlich wiederhergestellt werden. Als letzte Amtshandlung müssen wir uns beim MFP verabschieden, indem wir ein ganz bestimmtes Bit eines ganz bestimmten Registers des MFP löschen. Es würde jedoch Seiten füllen, die Funktion dieses Bits genau zu beschreiben. Deshalb bitte ich Sie hier nur, sich den bclr-Befehl im folgenden Beispielprogramm anzusehen und mir zu glauben, daß es so funktioniert. Wenn Sie sich näher für den MFP oder für dieses spezielle Register interessieren, dann sollten

Sie sich die Hardware-Beschreibung des Chips im ersten Intern-Band durchlesen. Wichtig ist noch, daß die Routine mit einem RTE (Return from Exception, Rückkehr von einer Ausnahmebehandlung) beendet wird.

Nun zur Implementierung: In Assembler ist die Sache recht einfach zu lösen. In C kommen wir nicht ohne fünf Zeilen Assembler aus, um die Register zu retten, eine beliebige C-Funktion aufzurufen, die Register wieder zurückzuholen, das geheimnisvolle Bit zu löschen und um den Interrupt mit RTE zu beenden. Die Startadresse der Funktion müssen wir im Hauptprogramm an Xbtimer übergeben; deshalb ist es erforderlich, das Label, das den Anfang des Assembler-Teils kennzeichnet, als Funktion zu deklarieren ('static entry()' im C-Listing).

Das Programm stellt einen Timer auf die niedrigste der möglichen Frequenzen ein: 48 Hz. 48mal in der Sekunde wird dann ein Zeichen aus einem String mit der BIOS-Funktion Bconout auf dem Bildschirm (Console) ausgegeben, und zwar im Interrupt. Während das passiert,wartet das Hauptprogramm gemütlich in einer Warteschleife:

C

```
/*********************************************/
/*   Demo für Benutzung der Timer des MFP    */
/*   Laser C     MP 02-01-89    TIMER_A.C    */
/*********************************************/

#include <osbind.h>
#define  CONSOLE 2

long i;


routine()
{
static char *str = "das ist eine wichtige meldung\15\12\12\12...
                    ... der timer a des mfp im atari st ist...
                    ... viel\15\12leichter anzuwenden, als...
                    ... die meisten Leute\15\12glauben      ...
                    ... \15\12\12\12          streng geheim ...
                    ... \15\12          ============= \15...
                    ... \12\12\12\12\12";  /* Riesenstring, was? */

  if (*str)    /* wenn noch nicht Stringende, dann... */
  {
    Bconout (CONSOLE, *str);   /* Zeichen ausgeben */
    str++;                     /* Zeiger auf nächstes Zeichen */
```

```
  }
}


static entry();
asm {       /* Einsprung für Interrupt */

entry:      movem.l D0-D7/A0-A6,-(A7)
            jsr     routine
            movem.l (A7)+,D0-D7/A0-A6
            bclr.b  #5,0xfffffa0f
            rte
    }


main()
{
  Cursconf (3,0);   /* Cursor soll nicht blinken */

  /* Timer anschmeißen: */
  /* Timer A (0) mit 48 Hz --> Vorteiler durch 200 (7) */
  /* und Datenregister 0=256 */

  Xbtimer (0, 7, 0, entry);

  /* Verzögerungsschleife: */
  for (i=0; i<1100000; i++);

  /* Timer wieder abschalten: */
  Xbtimer (0, 0, 0, entry);
}
```

Assembler

```
;
; Demo für die Benutzung der Timer des MFP
; Assembler     MP  02-01-89     TIMER_A.Q
;


gemdos        = 1
bios          = 13
xbios         = 14
console       = 2
bconout       = 3
cursconf      = 21
xbtimer       = 31
```

```
            TEXT

; Cursor auf 'nicht blinken' stellen:

            move.w  #3,-(sp)     ;Funktion: nicht blinken
            move.w  #cursconf,-(sp)
            trap    #xbios
            addq.l  #4,sp

; Installieren der Interrupt-Routine im Timer A (0).
; Frequenz: 48 Hz. Wird gebildet aus Vorteiler durch 200 (7)
; und Datenregister 256 (=0).

            pea     routine      ;Adresse der Interrupt-Routine
            clr.w   -(sp)        ;Datenregister ist 0=256
            move.w  #7,-(sp)     ;Kontrollregister mit Vorteiler
            clr.w   -(sp)        ;0 für Timer A
            move.w  #xbtimer,-(sp)
            trap    #xbios
            adda.l  #12,sp

; Warteschleife

            moveq.l #100,d0      ;reicht für gut 10 Sekundenouter:
            moveq.l #-1,d1
inner:      dbra    d1,inner
            dbra    d0,outer

; Timer wieder abstellen

            pea     routine
            clr.w   -(sp)        ;Datenregister
            clr.w   -(sp)        ;Kontrollregister (0 = ausschalten)
            clr.w   -(sp)        ;0 für Timer A
            move.w  #xbtimer,-(sp)
            trap    #xbios
            adda.l  #12,sp

            move.w  #2,-(sp)     ;Cursor soll wieder blinken
            move.w  #cursconf,-(sp)
            trap    #xbios
            addq.l  #4,sp

            clr.w   -(sp)        ;Programmende
            trap    #gemdos


routine:
; Einsprung Interrupt:
```

```
            movem.l d0-d7/a0-a6,-(sp) ;rette sich wer kann!

            clr.w   d0
            movea.l pointer,a0   ;Zeiger auf nächstes Zeichen
            move.b  (a0),d0      ;Zeichen holen
            beq     schluss      ;Null? Dann sind wir fertig...

            move.w  d0,-(sp)     ;sonst Zeichen ausgeben
            move.w  #console,-(sp)
            move.w  #bconout,-(sp)
            trap    #bios
            addq.l  #6,sp

            addq.l  #1,pointer

schluss:    movem.l (sp)+,d0-d7/a0-a6 ;Register zurück
            bclr    #5,$fffffa0f ;Interrupt in Service löschen
            rte                  ;End of Exception


            DATA

ausgabe:    DC.b 'das ist eine wichtige meldung',13,10,10,10
            DC.b 'der timer a des mfp im atari st ist viel',13,10
            DC.b 'leichter anzuwenden, als die meisten Leute',13,10
            DC.b 'glauben                                  '
            DC.b 13,10,10,10,'         streng geheim ',13,10
            DC.b '         ============= ',13,10,10,10,10,0

pointer:    DC.l ausgabe

            END
```

Der Timer A ist natürlich nur eine der vielen Möglichkeiten des MFP. Vielleicht hat das Beispielprogramm Ihr Interesse für diesen Baustein geweckt, so daß Sie mehr über ihn wissen möchten. Sie sollten in diesem Fall die Hardware-Beschreibung des MFP (Intern I) durchlesen; außerdem empfehle ich Ihnen, sich die Beschreibung der XBIOS-Routinen Mfpint (Nr. 13), Jdisint (Nr. 26) und Jenabint (Nr. 27) anzuschauen, um die direkte Programmierung des Chips zu vermeiden. Xbtimer (Nr. 31) kennen Sie ja bereits.

3.3 Rasterzeileninterrupt

Der letzte Hardware-Interrupt, den ich Ihnen vorstellen möchte, ist der Rasterzeileninterrupt. Er nutzt eine besondere Eigenschaft des MFP aus:

Die Timer A und B können nicht nur nach einer bestimmten Zeit einen Interrupt auslösen, sondern auch, wenn eine bestimmte Anzahl von Impulsen an einem der Pins des MFP-Gehäuses angekommen sind. Für den Timer A ist dieser Pin nicht angeschlossen, wohl aber für den Timer B. Hier meldet sich nämlich der Video-Chip mit jedem Zeilenrücklauf des Elektronenstrahls im Monitor. Dieser Interrupt ist also dem normalen HBL-Interrupt ähnlich.

Der Unterschied zum HBL liegt darin, daß wir mit dem Datenregister des Timers einstellen können, daß der Interrupt nicht bei jedem HBL, sondern erst nach einer bestimmten Anzahl von Zeilenrückläufen ausgelöst wird. Außerdem liegt der MFP, das wissen Sie bereits, auf der höchsten Interruptebene und hat die größte Priorität. Während also ein simpler HBL oft keine Chance hat, zum Prozessor "durchzukommen", haben die Timer Vorrang vor den meisten anderen Dingen.

Jetzt mögen Sie fragen, wozu dieser Interrupt überhaupt nützlicher sein könnte als ein normaler Timer. Antwort: Immer dann, wenn auf einem Teil des Bildschirms eine andere Farbpalette (Zuordnung der 512 möglichen Farben zu den max. 16 gleichzeitig darstellbaren) gelten soll als auf einem zweiten Teil. So kann man mit Hilfe des Rasterzeileninterrupts nicht nur alle 512 Farben gleichzeitig auf den Bildschirm bringen; man kann auch die Farben im Randbereich verändern (nützlich für Spiele), etwa um den Horizont einer Landschaft über die ganze Monitorbreite erkennen zu lassen.

Alles läuft darauf hinaus, die Farbpalette genau dann zu verändern, wenn der Elektronenstrahl des Monitors gerade eine ganz bestimmte Zeile schreibt (bzw. zu Ende geschrieben hat). Wenn das ganze Bild fertig geschrieben wurde (also beim VBL), muß die Änderung wieder rückgängig gemacht werden, damit der obere Teil des Bildschirms wieder normal erscheint. Damit ist auch klar, daß die Farben nur in vertikaler Richtung geändert werden können: Es kann einen oberen und einen unteren Bereich geben, aber nicht einen linken und einen rechten!

Wie sieht das nun im Programm aus? Wir initialisieren den Timer B mit der Funktion Xbtimer (XBIOS, Nr. 31), lassen ihn allerdings noch ausgechaltet (Kontrollregister ist 0). Dann hängen wir eine Interrupt-Routine in den VBL, die sich immer dann meldet, wenn das Bild ganz fertig ist. Allerdings benutzen wir nicht die VBL-Queue, weil möglicherweise schon ein oder zwei Rasterzeilen geschrieben wären, bis unser VBL-Programm aufgerufen wird (wenn andere VBL-Routinen schon vor uns eingetragen waren, werden diese natürlich zuerst bearbeitet). Deshalb hängen wir uns vor die Standard-VBL-Bearbeitung des Betriebssystems, in-

dem wir den Vektor $70 auf unsere Ersatz-VBL-Routine lenken, an deren Ende das Original-VBL-Programm gestartet wird (JMP im Listing; die Zieladresse wird erst im Programm eingetragen).

Wenn nun ein VBL-Interrupt ausgelöst wird, dann wählen wir zunächst die normale Hintergrundfarbe (weiß) für den oberen Bereich des Bildschirms, indem wir das Hardwareregister für die erste Farbe der Farbpalette mit dem Wert $777 füllen. Anschließend aktivieren wir den Timer B: Ins Datenregister schreiben wir, nach wie vielen Zeilen der Interrupt ausgelöst werden soll, und ins Kontrollregister schreiben wir die Zahl 8, die den Event-Count-Mode kennzeichnet (es werden Ereignisse, HBLs, gezählt). Einen Vorteiler gibt es in dieser Betriebsart des Timers nicht.

Wenn der Rasterzeileninterrupt nun ausgelöst wird, dann wechseln wir die Farbe 0, so daß diese z.B. blau statt weiß wird. Schließlich schalten wir den Interrupt wieder ab (0 ins Kontrollregister) und beenden den Interrupt (den Interrupt des Timers B müssen wir durch Löschen des Bits 0 im Register $FFFFFA0F offiziell beenden; beim Timer A war es das Bit 5). Beim nächsten VBL wird der Timer ja wieder aktiviert; der Bildschirmhintergrund wird bis dahin in der geänderten Farbe gezeichnet.

Zum Beispielprogramm: Der Rasterzeileninterrupt sollte für grafische Zwecke nur auf dem Farbmonitor genutzt werden, weil die sehr hohe Zeilenfrequenz im Monochrom-Betrieb zu Ungenauigkeiten führt. So ist der Elektronenstrahl nach dem Interrupt zu schnell wieder auf dem Weg, so daß die Umschaltung der Farben (??? Wir könnten den Bildschirm ja invertieren...) nicht ganz links am Bildrand geschieht. In der hohen Auflösung kann der Rasterzeileninterrupt höchstens dazu benutzt werden, festzustellen, wo ungefähr auf dem Bildschirm der Elektronenstrahl gerade steht. Auch dafür werden Sie noch ein Beispielprogramm finden.Ich habe jedoch ein Programm für den Farbmonitor geschrieben, das resident im Speicher bleibt. Es teilt den Bildschirm in zwei verschiedenfarbige Bereiche auf, die ihre Größe ändern, d.h. die Trennlinie zwischen den Farbbereichen wandert von oben nach unten und zurück. Am besten lassen Sie das Programm laufen; es behält seine Wirkung auch dann, wenn Sie ein anderes Programm starten.;

```
; Rasterzeileninterrupt mit dem Timer B des MFP
; Assembler        MP  02-01-89       TIMER_A.Q
;


gemdos      = 1
xbios       = 14
xbtimer     = 31
```

```
super       = $20
ptermres    = $31

vbl_vec     = $70
farbe0      = $ffff8240
bcontrol    = $fffffa1b         ;Kontrollregister Timer B
bdata       = $fffffa21         ;Datenregister Timer B


            TEXT

; Installieren der Interrupt-Routine im Timer B (1).
; Event-Count-Mode (16) --> kein Vorteiler.

            movea.l 4(sp),a0    ;Speicherbedarf errechnen
            move.l  #$100,d6
            add.l   12(a0),d6
            add.l   20(a0),d6
            add.l   28(a0),d6

            clr.l   -(sp)       ;Supervisor einschalten
            move.w  #super,-(sp)
            trap    #1
            addq.l  #6,sp
            move.l  d0,d7       ;SSP merken

            pea     routine     ;Adresse der Interrupt-Routine
            move.w  #50,-(sp)   ;Datenregister
            move.w  #0,-(sp)    ;Kontrollregister (Timer noch aus)
            move.w  #1,-(sp)    ;1 für Timer B
            move.w  #xbtimer,-(sp)
            trap    #xbios
            adda.l  #12,sp

            move.l  vbl_vec,jump+2    ;VBL-Routine in Sprungbefehl
            move.l  #new_vbl,vbl_vec  ;eigene VBL-Routine davor

            move.l  d7,-(sp)    ;zurück in den User-Modus
            move.w  #super,-(sp)
            trap    #gemdos
            addq.l  #6,sp

            clr.w   -(sp)       ;kein Fehler
            move.l  d6,-(sp)    ;Programmlänge
            move.w  #ptermres,-(sp)   ;Programm resident halten
            trap    #gemdos


; Routine zur Farbumschaltung:
```

```
routine:      move.w  #$700,farbe0 ;Rot als Hintergrundfarbe
              clr.b   bcontrol     ;Interrupt aus
              bclr    #0,$fffffa0f ;Interrupt in Service löschen
              rte                  ;End of Exception

; Neue VBL-Routine:

new_vbl:      move.w  #$777,farbe0 ;Bildschirmhintergrund weiß
              move.w  d0,-(sp)     ;Register retten
              move.b  richtung,d0
              add.b   d0,hoch      ;Wechsel 1 Zeile später als bisher
              tst.b   d0           ;Wandert nach oben oder unten?
              bmi.s   oben
              cmpi.b  #200,hoch    ;schon bei Zeile 200?
              beq.s   vbl_change   ;dann Richtung wechseln
              bra.s   vbl_cont
oben:         cmpi.b  #50,hoch     ;oder wieder ganz oben?
              bne.s   vbl_cont
vbl_change:   eori.b  #$ff,richtung     ;Zahl negieren
              addq.b  #1,richtung  ;(Zweierkomplement)
              subq.w  #1,flag      ;nächste Runde für Hauptprogrammvbl_cont:
              move.b  hoch,bdata   ;Zeilenzahl für Wechsel
              move.b  #8,bcontrol  ;Kontrollregister: Timer wieder an
              move.w  (sp)+,d0     ;Register zurück
jump:         jmp     $12345678    ;zurück in normale VBL-Routine


              DATA

flag:         DC.w 2
richtung:     DC.b 1
hoch:         DS.b 50

              END
```

Daß man den Rasterzeileninterrupt auch auf dem Monochrom-Monitor sinnvoll einsetzen kann, zeigt folgendes Programm. Es blendet einen Text in den Bildschirm eines laufenden Programms ein. Das Besondere dabei ist, daß der Rechner davon nichts merkt. Der Text stört also das Hauptprogramm nicht, und auch auf einer Hardcopy würde er nicht erscheinen. Wie ist das möglich?

Beim VBL programmieren wir den Timer B auf 100 Zeilen. Wenn der entsprechende Interrupt ausgelöst wird, dann schreiben wir unseren Text ab Zeile 110 auf den Bildschirm; vorher müssen wir den Bereich allerdings irgendwo retten. Sofort danach wird der Bildschirm wieder restauriert, d.h. der alte Inhalt erscheint wieder; dann beenden wir den Interrupt.

Zur Erklärung: Im Monochrom-Betrieb ist der Elektronenstrahl so schnell, daß wir schon 10 Zeilen vor der eigentlichen Text-Ausgabe den Interrupt auslösen lassen müssen. Beim Schreiben und Retten des Textes (das erfolgt zeilenweise) überholt uns der Elektronenstrahl. Die Folge: Wenn wir die letzte Pixelzeile des Textes geschrieben haben, dann hat der Video-Chip schon die oberste Zeile abgetastet. Es stört also nicht, wenn wir sofort nach dem Schreiben den alten Zustand wiederherstellen; denn auch dabei ist der Elektronenstrahl schneller als wir.

Der eingeblendete Text ist für das Hauptprogramm übrigens deshalb nicht vorhanden, weil er ja nur während unseres Interrupts überhaupt im Bildschirmspeicher steht. Übrigens: Das Programm besorgt sich die Adresse des physikalischen (!) Video-RAM durch die Adressen $FFFF8201 und $FFFF8203 (beides Bytes). Es sind Hardwareregister des Shifters. Die Register enthalten immer die Startadresse des gerade dargestellten Speicherbereichs. Dabei stehen im Register $FFFF8201 die Bits 16-23 dieser Adresse, im anderen die Bits 8-15. Die restlichen Bits sind 0. Das ist auch der Grund dafür, daß das Video-RAM nur an einer Pagegrenze beginnen darf (siehe auch 2.1).

Um die Text-Ausgabe zu beschleunigen und die Benutzung von Systemroutinen während des Interrupts zu vermeiden, habe ich folgenden Trick benutzt: Vor der Initialisierung der Interrupts wird der Bildschirm gelöscht und der String in der obersten Bildschirmzeile ausgegeben. Dann kopieren wir pro Pixelzeile 20 Bytes (in meinem Beispielprogramm ist der String so lang) bzw. fünf Langworte in einen sicheren Bereich. Im Interrupt müssen dann nur diese Langworte an die gewünschte Stelle kopiert werden.

```
;
; Rasterzeilen-Interrupt: Demo für Monochrom-Monitor
; Assembler          MP  02-01-88        RASTER2.Q
;


gemdos      = 1
xbios       = 14
cconws      = 9
xbtimer     = 31
super       = $20
ptermres    = $31

vbl_vec     = $70
_v_bas_ad   = $44e
bcontrol    = $fffffa1b          ;Kontrollregister Timer B
```

```
bdata       = $fffffa21         ;Datenregister Timer B


            TEXT

            movea.l 4(sp),a0    ;Speicherbedarf errechnen
            move.l  #$100,d6
            add.l   12(a0),d6
            add.l   20(a0),d6
            add.l   28(a0),d6

; Ausgabe des Textes auf dem Bildschirm:

            pea     ausgabe
            move.w  #cconws,-(sp)
            trap    #gemdos
            addq.l  #6,sp

            clr.l   -(sp)       ;Supervisor einschalten
            move.w  #super,-(sp)
            trap    #1
            addq.l  #6,sp
            move.l  d0,d7       ;SSP merken

; Kopieren dieses Textes (Pixel, keine ASCII-Codes mehr):

            movea.l _v_bas_ad,a0 ;logisches Video-RAM
            lea     characters,a1

            moveq.l #15,d0      ;16 Pixelzeilen
l1:         moveq.l #20/4-1,d1  ;20 Zeichen = 5 Langworte
l2:         move.l  (a0)+,(a1)+
            dbra    d1,l2
            lea     60(a0),a0   ;nächste Zeile (+80-20)
            dbra    d0,l1

; Installieren der Interrupt-Routine im Timer B (1).
; Event-Count-Mode --> kein Vorteiler. Noch ausgeschaltet.


            pea     routine     ;Adresse der Interrupt-Routine
            move.w  #0,-(sp)    ;Datenregister
            move.w  #0,-(sp)    ;Kontrollregister (Timer noch aus)
            move.w  #1,-(sp)    ;1 für Timer B
            move.w  #xbtimer,-(sp)
            trap    #xbios
            adda.l  #12,sp

            move.l  vbl_vec,jump+2   ;VBL-Routine in Sprungbefehl
```

```
            move.l  #new_vbl,vbl_vec  ;eigene VBL-Routine davor

            move.l  d7,-(sp)      ;zurück in den User-Modus
            move.w  #super,-(sp)
            trap    #gemdos
            addq.l  #6,sp

            clr.w   -(sp)         ;kein Fehler
            move.l  d6,-(sp)      ;Programmlänge
            move.w  #ptermres,-(sp)   ;Programm resident halten
            trap    #gemdos


routine:    movem.l d0-d7/a0-a6,-(sp)

            clr.l   d0            ;Bildschirmstart (physikalisch)
            move.b  $ffff8201,d0
            asl.w   #8,d0
            move.b  $ffff8203,d0
            asl.l   #8,d0

            movea.l d0,a0
            lea     110*80(a0),a0     ;auf Zeile 110 setzen
            movea.l a0,a3         ;merken

            lea     characters,a1
            lea     save,a2
            moveq.l #15,d0        ;16 Pixelzeilen

lop:        move.l  30(a0),(a2)+ ;retten
            move.l  34(a0),(a2)+
            move.l  38(a0),(a2)+
            move.l  42(a0),(a2)+
            move.l  46(a0),(a2)+

            move.l  (a1)+,30(a0) ;Text schreiben
            move.l  (a1)+,34(a0)
            move.l  (a1)+,38(a0)
            move.l  (a1)+,42(a0)
            move.l  (a1)+,46(a0)

            lea     80(a0),a0     ;nächste Zeile
            dbra    d0,lop

            lea     save,a1
            moveq.l #15,d0        ;16 Pixelzeilen

lop2:       move.l  (a1)+,30(a3) ;Bild zurückschreiben
            move.l  (a1)+,34(a3)
            move.l  (a1)+,38(a3)
```

```
                move.l  (a1)+,42(a3)
                move.l  (a1)+,46(a3)
                lea     80(a3),a3     ;nächste Zeile
                dbra    d0,lop2

                movem.l (sp)+,d0-d7/a0-a6
                clr.b   bcontrol      ;Timer abstellen

                bclr    #0,$fffffa0f
                rte


; Neue VBL-Routine:

new_vbl:        move.b  #100,bdata    ;Timer anwerfen (100 Zeilen)
                move.b  #8,bcontrol   ;Kontrollregister: Timer wieder an
jump:           jmp     $12345678     ;zurück in normale VBL-Routine


                DATA

ausgabe:        DC.b 27,'E I LIKE DATA BECKER ',0

                BSS

                ALIGN

characters:     DS.b 20*16
save:           DS.b 20*16

                END
```

4. Sound-Programmierung

Der Sound-Chip des Atari ST heißt YM-2149 und stammt von YAMAHA (daher das YM). Bis zur Serienproduktion des Computers war allerdings unklar, ob dieser Chip von YAMAHA oder ein ähnlicher von General Instruments verwendet werden sollte. Ich sage Ihnen das nur deshalb, weil die Routine des Betriebssystems, die den Soundchip bedient, mit den Buchstaben GI (General Instruments) beginnt. So kommt wenigstens keine Verwirrung auf.

Der Sound-Chip ist recht leistungsfähig. Auch die Programmierung des Chips ist an sich nicht schwierig, doch können die wenigsten Programmierer die Fähigkeiten voll ausschöpfen. Ich will Ihnen deshalb im folgenden sowohl die technischen Möglichkeiten des Chips als auch etwas Hintergrundwissen über die Tonerzeugung näherbringen.

4.1 Grundlagen

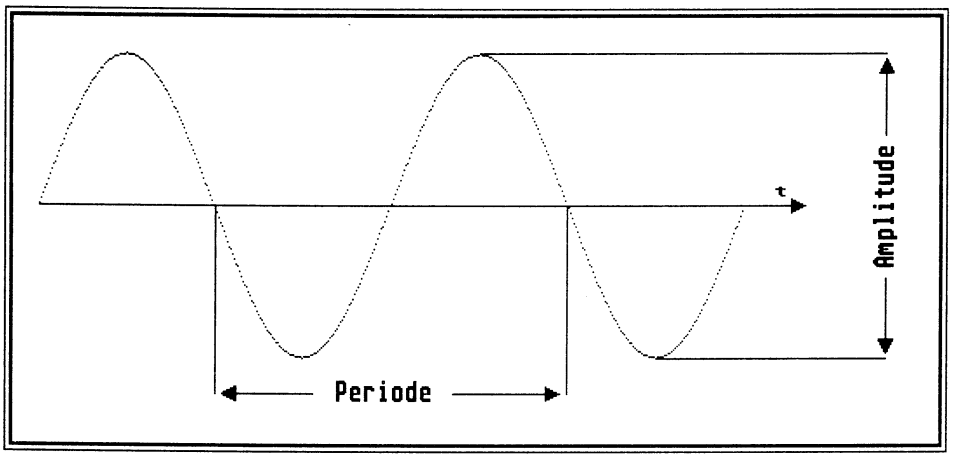
Einige Leser werden die folgenden Bemekungen für überflüssig halten. Ich wollte sie dennoch nicht als bekannt voraussetzen.

Betrachten wir zunächst einmal die Töne, die Sie z.B. auf einem Klavier spielen können. Die Klaviatur wird in Oktaven eingeteilt. Eine Oktave ist ein Bereich von einem C bis zum nächsthöheren H. Die Noten einer Oktave heißen C, D, E, F, G, A und H. Wenn Sie diese Noten nacheinander spielen, dann hören Sie das, was im allgemeinen einfach mit Tonleiter bezeichnet wird. Diese Töne entsprechen den weißen Tasten auf einem Klavier.

Nun gibt es aber auch noch die schwarzen Tasten, und zwar fünf in jeder Oktave. Durch die Anordnung dieser schwarzen Tasten kann man übrigens überhaupt erst erkennen, wo die verschiedenen Noten auf einer Klaviatur liegen. Spielt man nun eine Tonleiter, die auch die schwarzen Tasten enthält, dann bezeichnet man das als chromatische Tonleiter.

Töne lassen sich aber noch auf eine andere Weise bestimmen als über Oktave und Note, nämlich über die Frequenz. Ein Ton, den Sie hören, ist nämlich zunächst einmal eine Schwingung des Mediums, das den Ton weiterleitet (gewöhnlich Luft). Um eine solche Schwingung zu erzeugen,

muß sich die Membran eines Lautsprechers sehr schnell hin und her bewegen. Das könnte bei einem vom Computer generierten Ton so aussehen:



Auf der Abzisse ist die Zeit eingetragen, auf der Ordinate der Ausschlag der Membran relativ zur normalen Lage. Verändert man die Größe der Membranausschläge, also die Amplitude, dann verändert sich auch die Lautstärke des Tons. Wichtiger ist aber das, was in der Zeichnung mit Periode bezeichnet ist. Das ist ein Teil des Graphen, der von einer Nullstelle über den Tiefpunkt, noch eine Nullstelle und einen Hochpunkt wieder zur Nullstelle führt (oder umgekehrt). Leger formuliert ist es die Kombination von Berg und Tal des Graphen.

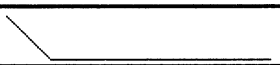
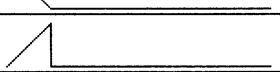
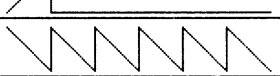
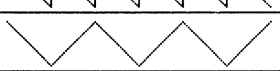
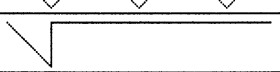
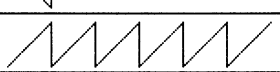
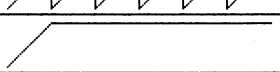
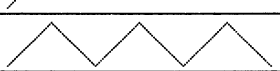
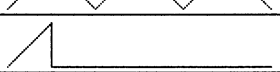
Ein Ton ist nun umso höher, je mehr solcher Perioden in der gleichen Zeit vorkommen. Um einen höheren Ton zu erreichen, muß sich die Membran des Lautsprechers also bei gleicher Amplitude schneller bewegen. Im gleichen Maße wie die Zahl der Perioden pro Zeit zunimmt, nimmt die zeitliche Ausdehnung einer einzelnen Periode, also die Periodendauer, ab. Die Zahl der Perioden pro Sekunde bezeichnet man als Frequenz (Maßeinheit: Hz). Für uns wird jedoch die Periodendauer wichtiger sein als die Frequenz. Merken Sie sich also diesen Begriff.

Nun müssen wir uns noch um den Lautstärke-Verlauf eines Tones kümmern. Stellen Sie sich ein Glockenspiel vor: Wenn hier ein Ton angeschlagen wird, dann erklingt er zunächst ganz laut, um dann langsam leiser zu werden. Oder eine Trompete: Bis der Trompeter den Ton sauber gefunden hat, spielt er etwas leiser. Dafür beendet er den Ton abrupt. Dieser Lautstärkeverlauf ist also einer der Charakteristika eines Instruments.

Wir können diese Änderung der Lautstärke auch durch den Sound-Chip unseres Computers durchführen lassen, um bestimmte Effekte zu erzielen. Dazu müssen wir eine Hüllkurve angeben. Diese bestimmt den Lautstärkeverlauf eines Tones mit der Zeit. Es gibt Hüllkurven, deren Verlauf periodisch ist, und solche, deren Lautstärke am Ende der Hüllkurve auf einem bestimmten Punkt bleibt: entweder Null oder ganz laut.

Es ist schwierig, den Verlauf der möglichen Hüllkurven unseres Sound-Chips in Worte zu fassen. Sehen Sie deshalb folgende Grafik, die auch schon die Nummern der einzelnen Kurven enthält (brauchen wir erst später):

Hüllkurven

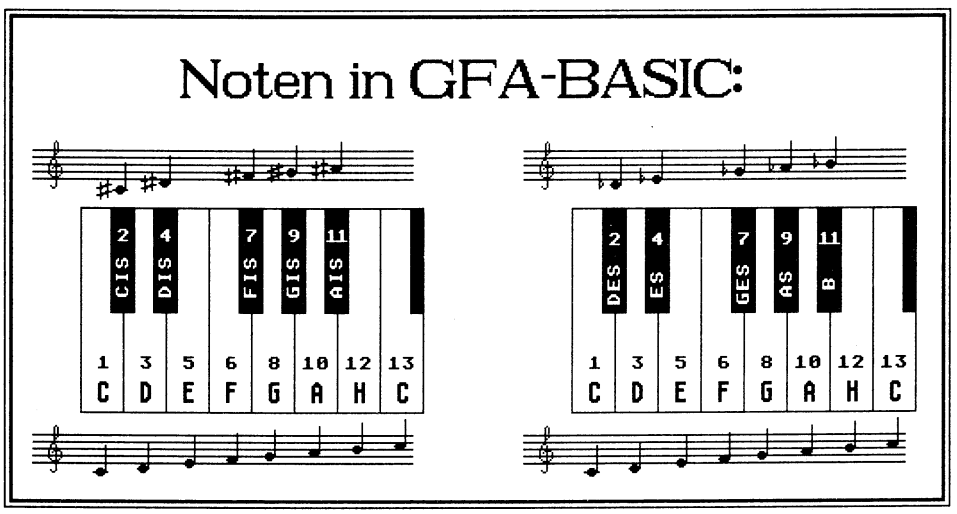
Nummer	Grafik	Beschreibung
1-3/9		linear fallend
4-7		linear steigend, dann 0
8		Sägezahn fallend
10		Dreieck, anfangs fallend
11		linear fallend, dann laut
12		Sägezahn steigend
13		linear steigend und laut
14		Dreieck, anfangs steigend
15		linear steigend, dann 0

4.2 SOUND in GFA-BASIC

SOUND ist die einfachste Anweisung, um dem Rechner unter GFA-BASIC Töne zu entlocken. Als Parameter müssen Sie den Sound-Kanal, die Lautstärke (0-15), die Note und die Oktave (1-8) angeben. Optional

können Sie eine Wartezeit angeben (in 50stel-Sekunden). Es ist nicht die Haltedauer des Tons, sondern die Zeit, die gewartet wird, bevor der nächste BASIC-Befehl ausgeführt wird.

Der Wert für die Note kann zwischen 1 und 12 liegen:



Das hohe C (ganz rechts auf den Klaviaturen) gehört eigentlich schon zur nächsten Oktave. Die höheren Oktaven haben übrigens die größeren Nummern. Der höchste Ton wäre also Oktave 8, Ton 12.

Der Sound-Kanal darf zwischen 1 und 3 liegen. Damit bestimmen Sie, über welchen der drei Tongeneratoren des Sound-Chips (A, B und C) der Ton gespielt werden soll. Mehrstimmig können Sie damit allerdings nicht spielen; denn wenn Sie für SOUND den Generator A (1) angeben, dann werden B und C automatisch abgeschaltet. Apropos: Abgeschaltet wird der Ton dadurch, daß Sie als Lautstärke eine Null angeben (die anderen Parameter entfallen dann).

Damit können wir schon ein erstes kleines Stück spielen lassen:

```
'
' Sound in GFA-BASIC - Der Sound-Befehl
' GFA-BASIC    MP 03-01-89   SOUND1.GFA
'
ganze=80              ! Notenwerte
halbe=ganze/2
viertel=ganze/4
```

```
'
c=1     ! Noten
d=3
e=5
f=6
g=8
a=10
h=12
'
' Wir spielen:
'
SOUND 1,15,e,4,viertel
SOUND 1,15,c,4,viertel
SOUND 1,15,d,4,viertel
SOUND 1,15,g,3,halbe+viertel     ! 3 = eine Oktave tiefer als 4SOUND 1,0,0,0,2
! kurze Pause
SOUND 1,15,g,3,viertel
SOUND 1,15,d,4,viertel
SOUND 1,15,e,4,viertel
SOUND 1,15,c,4,halbe+viertel
SOUND 1,0,0,0                    ! Ton abschalten
'
```

Die kurze Pause (siehe Kommentar im Listing) ist übrigens deshalb nötig, weil sonst die beiden gleichen Töne zu einem langen Ton verschmelzen würden!

4.3 Sound im allgemeinen

In Omrikon-BASIC, C und Assembler kennt der Rechner keine Noten und Oktaven mehr. Hier werden Töne über die Periodendauer angegeben. Eine Liste der Frequenzen und der entsprechenden Periodendauern finden Sie im Anhang C.

Zunächst möchte ich die Hardware-Register des Soundchips vorstellen.

Register 0 und 1:

Diese Register steuern die Periodendauer des Tones, der vom Kanal A erzeugt werden soll. Dabei stehen im Register 0 die unteren acht Bits dieses Wertes, während sich im unteren Nibble (4 Bits) des Registers 1 die oberen 4 Bits befinden. Wir haben also zur Steuerung der Tonhöhe 8+4=12 Bits und können folglich Periodendauern zwischen 0 und 4095 angeben. Natürlich ist die Einheit dieser Zahl nicht Sekunden; Sie müssen

vielmehr die Konstante 125000 durch die gewünschte Frequenz teilen. Den gerundeten Wert schreiben Sie in die Register 0 und 1, und der Ton wird erzeugt.

Damit Sie nicht soviel rechnen müssen, sind die Periodendauern zusätzlich zu den Frequenzen der wichtigen Töne im Anhang C abgedruckt. Dort finden Sie auch gleich Low- und High-Byte dieses Wertes.

Register 2 und 3:

Wie 0/1, aber für Tongenerator B.

Register 4 und 5:

Wie 0/1 bzw. 2/3, aber für Generator C.

Register 6:

Zusätzlich zu den Tongeneratoren verfügt der Soundchip noch über einen Rauschgenerator. Das Register 6 beeinflußt das Rauschen. Je kleiner der Wert, den Sie hier hineinschreiben, desto höher klingt das Rauschen (man kann nicht wirklich von einer Tonhöhe sprechen). Es werden nur die unteren 5 Bits benutzt, d.h. es sind Werte zwischen 0 und 31 sinnvoll.

Register 7:

Dieses Register heißt Multifunktionsregister. Die Bits 0 bis 5 sind für uns von Bedeutung; die Bits 6 und 7 müssen immer auf Eins gesetzt sein!

Mit den Bits 0, 1 und 2 werden die drei Tongeneratoren eingeschaltet (Bit 0 ist Generator A usw.). Merkwürdigerweise bedeutet ein gesetztes Bit, daß der entsprechende Kanal abgeschaltet ist. Die Bits 3, 4 und 5 schalten den Rauschgenerator (siehe Register 6) auf die Kanäle A, B und C auf. Auch hier gilt: Bit gesetzt = Rauschen aus. Ein Wert von 252 (Binär: 11111100) in diesem Register würde also bedeuten, daß nur die Generatoren A und B angeschaltet sind.

Register 8:

Hiermit wird die Lautstärke des Kanals A eingestellt; es sind Werte zwischen 0 und 15 erlaubt. Wenn Sie hier eine 16 hineinschreiben, so bedeutet dies, daß die Lautstärke von Kanal A nicht über dieses Register 8, sondern über die gerade eingestellte Hüllkurve (siehe Register 11 bis 13) gesteuert wird.

Register 9:

Wie 8, aber für Kanal B.

Register 10:

Wie 8 bzw. 9, aber für Kanal C.

Register 11/12:

Erinnern Sie sich an die grafische Darstellung der Hüllkurven? Mit diesem Registerpaar können Sie bestimmen, wie steil oder flach die Lautstärke, wenn sie durch die Hüllkurve gesteuert wird, ansteigt oder fällt. Mit anderen Worten: Sie regeln hiermit die Geschwindigkeit der Lautstärke-Änderung. So kann ein Ton z.B. sehr schnell oder auch langsam ausklingen. Das Register steuert also die zeitliche Ausdehnung einer Hüllkurvenperiode und wird deshalb ebenfalls Periodendauer genannt. Passen Sie also auf, daß Sie es nicht mit der Periodendauer aus den Registern 0 bis 5 verwechseln.

Register 11 enthält das Low-Byte, Register 12 das High-Byte. Es werden alle 16 Bits genutzt; damit sind Werte zwischen 0 und 65535 möglich.

Register 13:

Hiermit stellen Sie die Hüllkurve ein, die den Lautstärkeverlauf steuert (allerdings nur, wenn das Lautstärkeregister des entsprechenden Tonkanals auf 16 steht; siehe Register 8 bis 10). Die Werte entnehmen Sie bitte der obenstehenden Abbildung der verschiedenen Hüllkurven. Beachten Sie, daß einigen Hüllkurven mehr als eine Nummer zugewiesen wurde. Das ist kein Fehler, sondern hängt von der Codierung der Bits dieses Registers ab. Näheres finden Sie im Intern Band 1.

Schon jetzt ein Hinweis: Ein Beschreiben dieses Registers führt immer dazu, daß die Hüllkurve neu begonnen wird (egal ob Sie eine neue oder die gleiche Hüllkurve noch einmal angegeben haben). Wenn Sie also später zwei Töne nacheinander spielen möchten, so müssen Sie, damit der zweite Ton auch wirklich neu angeschlagen wird, das Register 13 beschreiben und nicht die Register, die die Periodendauer des Tons bestimmen.

Die Register 14 und 15 sind für uns nicht von Interesse, weil Sie nicht für die Sounderzeugung bestimmt sind.

Nachdem nun die Bedeutung der Register geklärt ist, benötigen wir eine Routine, um die Register auslesen und beschreiben zu können. Das ist die XBIOS-Funktion Giaccess (Nummer 28). Sie wird wie folgt aufgerufen:

```
x=XBIOS(28,daten,register)     !     GFA-BASIC

XBIOS(x,28,daten,register)'          OMIKRON-BASIC

x = Giaccess (daten, register);  /* C */

move.w #register,-(sp)             ; Assembler
move.w #daten,-(sp)
move.w #28,-(sp)
trap   #14
addq.l #6,sp                       ; x in D0
```

Mit register bestimmen Sie, wie der Name schon sagt, auf welches Register im Sound-Chip Sie zugreifen möchten, wobei die Registernummer zwischen 0 und 13 liegen sollte. Wenn zusätzlich das Bit 7 gesetzt ist (oder zur Registernummer die Zahl 128 addiert wird, was dasselbe ist), dann wird der Wert daten in dieses Register geschrieben. Bei gelöschtem Bit 7 wird statt dessen das entsprechende Register ausgelesen und dessen Inhalt als Funktionswert zurückgegeben.

In BASIC ist die Verwendung dieser Funktion jedoch oft nicht erforderlich, weil man mit den Anweisungen SOUND und WAVE (GFA-BASIC) bzw. TUNE, NOISE und VOLUME die meisten Dinge regeln kann. Das habe ich dann auch in den beiden folgenden Beispielprogrammen gemacht. Schauen Sie bitte im Handbuch Ihrer Programmiersprache nach oder sehen Sie sich die Programmlistings an, aus denen die Verwendung eigentlich deutlich hervorgeht.

Zunächst ein Programm, das eine Art Sirenenton erzeugt. Es benutzt dazu (auch in GFA-BASIC) die Periodendauer eines Tones:

GFA-BASIC

```
'
' Sound mit Angabe der Periodendauer
' GFA-BASIC  MP 03-01-89  SOUND2.GFA
'
'
FOR a=1 TO 5
  FOR i=4000 DOWNTO 500   ! Ton wird höher
    SOUND 1,15,#i
  NEXT i
```

```
  '
  FOR i=500 TO 4000       ! und wieder niedriger
    SOUND 1,15,#i
  NEXT i
NEXT a
'
SOUND 1,0   ! Ton aus
'
```

Omikron-BASIC

```
'
' Sound mit Angabe der Periodendauer
' OMIKRON    MP 03-01-89  SOUND2.BAS
'
FOR A=1 TO 5
  FOR I=4000 TO 500 STEP -1
    TUNE 1,I
  NEXT I
  '
  FOR I=500 TO 4000
    TUNE 1,I
  NEXT I
NEXT A
'
TUNE 1,0'   Schaltet Kanal ab
END
```

C

```
/*************************************************/
/*   Sound unter Verwendung der Periodendauer  */
/*   Laser C      MP  03-01-89      SOUND2.C   */
/*************************************************/

#include <osbind.h>

int  i,a;


sound (kanal, periode)
int kanal, periode;
{
  Giaccess (periode & 255, 128 + (kanal-1)*2);    /* Low-Byte */
  Giaccess (periode >> 8,  128 + (kanal-1)*2+1);  /* High-Byte */
}


main()
{
```

```
  Giaccess (254, 128 + 7);   /* nur Tonkanal A an */
  Giaccess (15,  128 + 8);   /* Lautstärke A = 15 */

  for (a = 0;  a < 5;  a++)
  {
    for (i = 4000;  i > 500;  i--)
      sound (1, i);

    for (i = 500;  i < 4000;  i++)
      sound (1,i);
  }

  sound (1, 0);   /* Ton abschalten */
}
```

Assembler

```
;
; Sound unter Verwendung der Periodendauer eines Tons
; Assembler            MP  03-01-89          SOUND2.Q
;

gemdos      = 1
xbios       = 14
giaccess    = 28


            TEXT

            move.w  #128+7,-(sp) ;Kanal A einschalten
            move.w  #254,-(sp)
            move.w  #giaccess,-(sp)
            trap    #xbios
            addq.l  #6,sp

            move.w  #128+8,-(sp) ;Lautstärke Kanal A
            move.w  #15,-(sp)    ;aufdrehen
            move.w  #giaccess,-(sp)
            trap    #xbios
            addq.l  #6,sp

            moveq.l #4,d7        ;5 Durchgänge

loop:       move.w  #4000,d6     ;Anfangswert für Periodendauer
l1:         bsr     play         ;Ton spielen
            subq.w  #1,d6        ;um eins erniedrigen
            cmpi.w  #500,d6      ;Endwert?
            bne.s   l1

l2:         bsr     play         ;spielen
```

```
                addq.w  #1,d6
                cmpi.w  #4000,d6      ;Wieder ganz oben?
                bne.s   l2

                dbra    d7,loop       ;fünfmal, bitte

                clr.w   d6            ;Ton abschalten
                bsr     play

                clr.w   -(sp)         ;Ende
                trap    #gemdos


play:           move.w  d6,d5         ;Ton in d6 über Kanal A spielen

                move.w  #128,-(sp)    ;Kanal A low
                move.w  d5,-(sp)
                andi.w  #$ff,(sp)     ;nur Bits 0 bis 7
                move.w  #giaccess,-(sp)
                trap    #xbios
                addq.l  #6,sp

                move.w  #128+1,-(sp) ;Kanal A high
                asr.w   #8,d5
                move.w  d5,-(sp)
                move.w  #giaccess,-(sp)
                trap    #xbios
                addq.l  #6,sp

                rts

                END
```

Im Assembler- und C-Listing finden Sie auch jeweils ein kleines Unterprogramm zum Einstellen einer bestimmten Periodendauer, das Sie sicher auch gut in Ihren eigenen Programmen verwenden können.

Ein weiteres Beispielprogramm soll nun demonstrieren, in wie vielfältiger Weise man ein und denselben Ton durch einen Wechsel der Hüllkurve und durch verschiedene Hüllkurven-Periodendauern verändern kann. Hören Sie selbst:

GFA-BASIC

```
'
' Hüllkurven         SOUND3.GFA
' GFA-BASIC       MP  03-01-89
'
SOUND 1,15,1,4          ! Ton einschalten (C)
```

```
'
WAVE 1,1,14,15,200    ! Dreieck (14) / Periodendauer: 15
WAVE 1,1,14,400,200   ! gleicher Ton, aber längere Periodendauer
WAVE 1,1,14,3000,200  ! nochmal länger...
'
WAVE 1,1,8,3000,200   ! Neue Hüllkurve: Sägezahn fallend (8)
WAVE 1,1,1,40000,200  ! linear fallend (1), lange Periodendauer
'
SOUND 1,0             ! Ton abschalten
'
```

Omikron-BASIC

```
'
' Hüllkurven     SOUND3.BAS
' OMIKRON      MP  03-01-89
'
TUNE 1,478'    Irgend ein Ton für unsere Versuche...
'
VOLUME 1,14,15: WAIT 4'    Dreieck (14) / Periodendauer: 15
VOLUME 1,14,400: WAIT 4'   gleicher Ton, aber längere Dauer
VOLUME 1,14,3000: WAIT 4'  nochmal länger...
'
VOLUME 1,8,3000: WAIT 4'   Sägezahn fallend (8)
VOLUME 1,1,40000: WAIT 4'  Linear fallend (1)
'
TUNE 1,0
END
```

C

```
/*********************************/
/*   Hüllkurven       SOUND3.C   */
/*   Laser C      MP  03-01-89   */
/*********************************/

#include <osbind.h>


sound (kanal, periode)
int kanal, periode;
{
  Giaccess (periode & 255, 128 + (kanal-1)*2);    /* Low-Byte */
  Giaccess (periode >> 8,  128 + (kanal-1)*2+1);  /* High-Byte */
}


wave (kurve, dauer)
int kurve, dauer;
{
```

```
  Giaccess (kurve,       128 + 13);
  Giaccess (dauer & 255, 128 + 11);
  Giaccess (dauer >> 8,  128 + 12);
}


warte()         /* Verzögerungsschleife */
{
long i;

  for (i = 0;  i < 500000;  i++);
}


main()
{
  Giaccess (254, 128 + 7);  /* Kanal A einschalten */
  Giaccess (16,  128 + 8);  /* Lautstärke durch Hüllkurve gesteuert */

  sound (1, 478);     /* 'Versuchston' anschalten */

  wave (14, 15);
  warte();

  wave (14, 400);
  warte();

  wave (14, 3000);
  warte();

  wave (8, 3000);
  warte();

  wave (1, 40000);
  warte();

  sound (1, 0);   /* Ton abschalten */
}
```

Assembler

```
;
; Hüllkurven        SOUND3.Q
; Assembler      MP 03-01-89
;

gemdos        = 1
xbios         = 14
giaccess      = 28
```

```
          TEXT

          move.w  #128+7,-(sp) ;Kanal A einschalten
          move.w  #254,-(sp)
          move.w  #giaccess,-(sp)
          trap    #xbios
          addq.l  #6,sp

          move.w  #128+8,-(sp) ;Lautstärke Kanal A
          move.w  #16,-(sp)    ;gesteuert durch Hüllkurve
          move.w  #giaccess,-(sp)
          trap    #xbios
          addq.l  #6,sp

          move.w  #478,d6      ;Test-Ton
          bsr     play

          move.w  #14,d7       ;Dreieck
          move.w  #15,d6       ;Periodendauer
          bsr     kurve
          bsr     wait

          move.w  #400,d6      ;neue Periodendauer
          bsr     kurve
          bsr     wait

          move.w  #3000,d6
          bsr     kurve
          bsr     wait

          move.w  #8,d7        ;Sägezahn fallend
          bsr     kurve        ;P.-dauer immer noch 3000
          bsr     wait

          move.w  #1,d7        ;linear fallend
          move.w  #40000,d6    ;ganz schön lang!
          bsr     kurve
          bsr     wait

          clr.w   d6           ;Ton abschalten
          bsr     play

          clr.w   -(sp)        ;Ende
          trap    #gemdos


play:     move.w  d6,d5        ;Ton in d6 über Kanal A spielen

          move.w  #128,-(sp)   ;Kanal A low
```

```
            move.w  d5,-(sp)
            andi.w  #$ff,(sp)    ;nur Bits 0 bis 7
            move.w  #giaccess,-(sp)
            trap    #xbios
            addq.l  #6,sp

            move.w  #128+1,-(sp) ;Kanal A high
            asr.w   #8,d5
            move.w  d5,-(sp)
            move.w  #giaccess,-(sp)
            trap    #xbios
            addq.l  #6,sp

            rts


kurve:      move.w  #128+13,-(sp)     ;Hüllkurve
            move.w  d7,-(sp)
            move.w  #giaccess,-(sp)
            trap    #xbios
            addq.l  #6,sp

            move.w  d6,d5         ;Periodendauer in d6

            move.w  #128+11,-(sp)     ;low
            move.w  d5,-(sp)
            andi.w  #$ff,(sp)    ;nur Bits 0 bis 7
            move.w  #giaccess,-(sp)
            trap    #xbios
            addq.l  #6,sp

            move.w  #128+12,-(sp)     ;high
            asr.w   #8,d5
            move.w  d5,-(sp)
            move.w  #giaccess,-(sp)
            trap    #xbios
            addq.l  #6,sp

            rts

wait:       moveq.l #-1,d0
l1:         moveq.l #30,d1
l2:         dbra    d1,l2
            dbra    d0,l1
            rts


            END
```

Es sei noch einmal wiederholt: Wenn Sie Musik machen und dabei eine Hüllkurve verwenden, die etwas langsamer ist, etwa um den Ton langsam ausklingen zu lassen, dann müssen Sie für jeden neuen Ton die entsprechende Hüllkurvennummer in das Register 13 schreiben!

4.4 Sound im Interrupt

Unser ST besitzt eine außergewöhnliche Routine im XBIOS, die, einmal aufgerufen, beliebig lange und komplizierte Befehlssequenzen an den Sound-Chip sendet. Das geschieht während eines Systeminterrupts, so daß ein Programm normal weiterarbeiten kann, während die Musik läuft. So wird in den meisten Spielen das Hauptprogramm geladen, während gleichzeitig irgendein Titellied gespielt wird. Nur eins sollten Sie beachten: Der Tastenklick beendet die ganze Aktion (Systemvariable $484 (Byte) zum Abschalten des Tastenklicks, siehe Intern 1).

Diese XBIOS-Routine heißt Dosound. Sie hat die Funktionsnummer 32. Als Parameter ist ein Zeiger auf eine Liste mit Soundbefehlen zu übergeben.

Diese Soundbefehle sind alle ein Byte lang. Es bedeuten:

0-15: Das folgende Byte wird in das durch das Befehlsbyte bezeichnete Register des Sound Chips geschrieben.

255: Das folgende Byte gibt die Wartezeit in 50stel Sekunden an, die der Rechner warten soll, bevor der nächste Sound-Befehl ausgeführt werden soll. Damit wird jedoch nicht das laufende Programm angehalten! Ist das der 255 folgende Byte eine Null, dann wird die Interrupt-Soundverarbeitung beendet.

Es gibt zwar noch zwei weitere Kommandos, doch die werden so gut wie nie gebraucht und können immer durch die anderen Kommandos ersetzt werden. Näheres auch hierzu im Intern 1.

Ein Musikstück läßt sich nun dadurch schreiben, daß die Register des Soundchips mit bestimmten Werten geladen werden, was einen Ton ergibt. Dann wird solange gewartet, bis die erste Änderung eines der Register erforderlich ist usw. Sicher können Sie sich vorstellen, daß diese Art der Komposition nicht gerade komfortabel ist. Deshalb gibt es eine Reihe von Musikprogrammen, die ein Musikstück nicht nur spielen können, sondern es optional in Zahlenform für den Dosound-Befehl ausgeben.

Daß es dennoch ohne ein solches Programm geht, beweist folgende Demo. Die C- und Assembler-Programmierer beachten bitte, daß das Programm eigentlich resident im Speicher gehalten werden müßte, um die Sound-Daten vor Überschreiben zu schützen. Der Einfachheit halber habe ich im Demoprogramm darauf verzichtet. Falls Sie das Programm verändern möchten, so beachten Sie bitte, daß aus diesem Grund die Extension der Programmdatei .TOS heißen muß (nicht .PRG), wenn Sie das Programm vom Desktop aus starten wollen.

GFA-BASIC

```
'
' Interrupt-Musik mit Dosound (XBIOS, Nr. 32)
' GFA-BASIC      MP  03-01-89      SOUND4.GFA
'
s$=""   ! Befehlsstring basteln
'
DO
  READ befehl
  EXIT IF befehl=-1
  s$=s$+CHR$(befehl)
LOOP
'
VOID XBIOS(32,L:VARPTR(s$))
'
END
'
'
' Musik-Daten:
'
' Alle Tongeneratoren auf 0:
'
DATA 0,0,1,0,2,0,3,0,4,0,5,0
'
' Lautstärke: A und B durch Hüllkurve gesteuert, C 0
'
DATA 8,16,9,16,10,0
'
' Hüllkurve: linear fallend, Periodendauer: 8000
'
DATA 13,1,11,64,12,31
'
' Generatoren A und B einschalten:
'
DATA 7,252
'
' Jetzt kommen die Töne:
'
DATA 0,222,1,1,255,16
```

```
DATA 0,170,13,1,255,16
DATA 0,123,2,222,3,1,13,1,255,32
DATA 13,1,255,46
DATA 0,102,2,170,13,1,255,16
DATA 0,28,2,123,13,1,255,24
DATA 0,63,13,1,255,8
DATA 13,1,255,32
DATA 0,123,2,222,13,1,255,16
DATA 0,63,2,123,13,1,255,16
DATA 2,170,13,1,255,24
DATA 0,102,13,1,255,8
DATA 13,1,255,32
DATA 0,63,2,250,13,1,255,16
DATA 0,102,2,0,3,0,13,1,255,16
DATA 0,123,2,222,3,1,13,1,255,64
'
'
'
' Ende: Alles wieder auf Null/normal:
'
DATA 0,0,1,0,2,0,3,0,4,0,5,0,7,255,8,15,9,15,10,15
'
' Bearbeitung abbrechen:
'
DATA 255,0
'
DATA -1
'
```

Omikron-BASIC

```
'
' Interrupt-Musik mit Dosound (XBIOS, Nr. 32)
' OMIKRON-BASIC    MP  03-01-89    SOUND4.BAS
'
S$=""'      Befehlsstring basteln
'
WHILE 1'   Endlosschleife
  READ Befehl
  IF Befehl=-1 THEN EXIT
  S$=S$+ CHR$(Befehl)
WEND
'
Strptr= LPEEK( VARPTR(S$))+ LPEEK( SEGPTR +28)
XBIOS (,32, HIGH(Strptr), LOW(Strptr))
'
END
'
'
' Musik-Daten:
```

```
'
' Alle Tongeneratoren auf 0:
'
DATA 0,0,1,0,2,0,3,0,4,0,5,0
'
' Lautstärke: A und B durch Hüllkurve gesteuert, C 0
'
DATA 8,16,9,16,10,0
'
' Hüllkurve: linear fallend, Periodendauer: 8000
'
DATA 13,1,11,64,12,31
'
' Generatoren A und B einschalten:
'
DATA 7,252
'
' Jetzt kommen die Töne:
'
DATA 0,222,1,1,255,16
DATA 0,170,13,1,255,16
DATA 0,123,2,222,3,1,13,1,255,32
DATA 13,1,255,46
DATA 0,102,2,170,13,1,255,16
DATA 0,28,2,123,13,1,255,24
DATA 0,63,13,1,255,8
DATA 13,1,255,32
DATA 0,123,2,222,13,1,255,16
DATA 0,63,2,123,13,1,255,16
DATA 2,170,13,1,255,24
DATA 0,102,13,1,255,8
DATA 13,1,255,32
DATA 0,63,2,250,13,1,255,16
DATA 0,102,2,0,3,0,13,1,255,16
DATA 0,123,2,222,3,1,13,1,255,64
'
'
'
' Ende: Alles wieder auf Null/normal:
'
DATA 0,0,1,0,2,0,3,0,4,0,5,0,7,255,8,15,9,15,10,15
'
' Bearbeitung abbrechen:
'
DATA 255,0
'
DATA -1
'
```

C

```
/*******************************************/
/*   Interrupt-Musik mit Dosound-Funktion  */
/*   Laser C     MP  03-01-89    SOUND4.C  */
/*******************************************/

#include <osbind.h>

char daten[] = {

   0,0,1,0,2,0,3,0,4,0,5,0,        /* A, B und C auf 0 */
   8,16,9,16,10,0,                 /* Lautstärke: A und B Hüllk. */
   13,1,11,64,12,31,               /* linear fallend, P.: 8000 */
   7,252,                          /* A und B anstellen */

   0,222,1,1,255,16,               /* Hier spielt die Musik... */
   0,170,13,1,255,16,
   0,123,2,222,3,1,13,1,255,32,
   13,1,255,46,
   0,102,2,170,13,1,255,16,
   0,28,2,123,13,1,255,24,
   0,63,13,1,255,8,
   13,1,255,32,
   0,123,2,222,13,1,255,16,
   0,63,2,123,13,1,255,16,
   2,170,13,1,255,24,
   0,102,13,1,255,8,
   13,1,255,32,
   0,63,2,250,13,1,255,16,
   0,102,2,0,3,0,13,1,255,16,
   0,123,2,222,3,1,13,1,255,64,

   0,0,1,0,2,0,3,0,4,0,5,0,7,255,8,15,9,15,10,15,  /* alles aus */
   255,0                           /* Soundverarbeitung beenden */
};

main()
{
  Dosound (daten);
}
```

Assembler

```
;
; Interrupt-Musik mit Dosound (XBIOS, Nr. 32)
; Assembler       MP  03-01-88       SOUND4.Q
;

gemdos        = 1
```

```
xbios       = 14
dosound     = 32


            TEXT

            pea     daten
            move.w  #dosound,-(sp)
            trap    #xbios
            addq.l  #6,sp

            clr.w   -(sp)
            trap    #gemdos


            DATA

daten:      DC.b 0,0,1,0,2,0,3,0,4,0,5,0
            DC.b 8,16,9,16,10,0
            DC.b 13,1,11,64,12,31
            DC.b 7,252

            DC.b 0,222,1,1,255,16
            DC.b 0,170,13,1,255,16
            DC.b 0,123,2,222,3,1,13,1,255,32
            DC.b 13,1,255,46
            DC.b 0,102,2,170,13,1,255,16
            DC.b 0,28,2,123,13,1,255,24
            DC.b 0,63,13,1,255,8
            DC.b 13,1,255,32
            DC.b 0,123,2,222,13,1,255,16
            DC.b 0,63,2,123,13,1,255,16
            DC.b 2,170,13,1,255,24
            DC.b 0,102,13,1,255,8
            DC.b 13,1,255,32
            DC.b 0,63,2,250,13,1,255,16
            DC.b 0,102,2,0,3,0,13,1,255,16
            DC.b 0,123,2,222,3,1,13,1,255,64

            DC.b 0,0,1,0,2,0,3,0,4,0,5,0,7,255,8,15,9,15,10,15
            DC.b 255,0

            END
```

5. GEM-Programmierung

GEM hat sich zu einem richtigen Schlagwort entwickelt: Jeder kennt es, jeder hat es und jeder will damit umgehen können. Nur: Kaum jemand kann sagen, was GEM wirklich ist. Bevor Sie also lernen, mit GEM umzugehen, möchte ich Ihnen erst ein wenig "Allgemeinwissen" darüber vermitteln.

Ein Blick ins Englisch-Wörterbuch verrät: GEM heißt Schmuckstück oder Juwel, was aber eher Zufall sein dürfte. GEM ist vielmehr eine Abkürzung für den Begriff Graphics Environment Manager, auf Deutsch etwa: Verwalter einer grafischen Umgebung. Halten Sie die Übersetzung irgendwo im Hinterkopf fest.

Manchmal hört man, GEM sei ein Betriebssystem. Das ist falsch; Sie wissen bereits, daß das Betriebssystem des Atari TOS heißt. Schwieriger wird es mit der zweiten Behauptung: GEM sei eine Benutzeroberfläche. Klären wir kurz diesen Begriff: Eine Benutzeroberfläche ist die Verbindung zwischen dem Benutzer eines Programms und den Funktionen, die das Programm dem Benutzer bieten möchte. So kann ein Programm durchaus sehr leistungsfähig sein, doch wenn der Benutzer nicht weiß, wie er all diese Funktionen verwenden kann oder die Anwendung zu umständlich ist, so sind sie für ihn nutzlos - es fehlt eine gute Benutzeroberfläche. Die Benutzeroberfläche wird praktisch als Schale (englisch: Shell) um ein Programm herumgebaut. Der Anwender sagt nun dem Programm nicht direkt, was er will, sondern teilt der Schale seine Wünsche mit, die dann ihrerseits das Programm aktiviert. Das gleiche System findet man oft auch bei Compilern: Ein Tastendruck oder Mausklick genügt, um einen Quelltext zu compilieren und anschließend gleich zu linken. Dabei geben Sie den Tastendruck an die Schale oder Shell, die dann ihrerseits alle benötigten Teilprogramme mit den notwendigen Parametern oder Kommandozeilen versorgt und aufruft.

GEM ist zwar keine Benutzeroberfläche, aber der Begriff geht schon in die richtige Richtung. Erinnern wir uns an die deutsche Übersetzung: Verwalter einer grafischen Umgebung. Die Umgebung ist in diesem Fall nichts anderes als eine Benutzeroberfläche, nämlich die Schale (Shell), die ein Programm umgibt. Grafisch will sagen, daß der Benutzer nicht, wie etwa bei PCs üblich, seine Befehle über die Tastatur (also als Text) eingibt, sondern, Sie kennen es, mit der Maus einen Pfeil auf dem Bildschirm bewegt, auf Bildchen zeigt und diese dann schließlich anklickt. So bleibt nur noch das Wort Verwalter zu klären. Damit ist gemeint, daß

GEM selbst keine grafische Benutzeroberfläche ist, sondern nur die Programmierung einheitlicher grafischer Benutzeroberflächen unterstützt.

Vor allem das Wort einheitlich ist von Bedeutung, denn GEM sorgt dafür, daß die Bedienung von Programmen zu einem gewissen Grad genormt ist. Es ist zum Beispiel kein Zufall, daß ein Druck auf die Taste Esc innerhalb einer Dialogbox stets das aktuelle Eingabefeld löscht, daß die Fenster in jedem Programm fast gleich aussehen und daß Sie in der obersten Bildschirmzeile fast immer eine Menüzeile finden.

Ein Hinweis noch, bevor es richtig losgeht: GEM ist ein sehr leistungsfähiges und daher auch umfangreiches System. Der Umgang damit ist zwar nicht Experten vorbehalten, doch benötigen Sie einige Grundkenntnisse, bevor ich Ihnen die ersten Beispielprogramme anbieten kann - das wird zwar nicht schwer werden, aber solange müssen Sie wohl oder übel durchhalten. Ja, und noch etwas: Sie werden bald eine Beschreibung der GEM-Funktionen benötigen, die des großen Umfangs wegen natürlich nicht in diesem Buch untergebracht werden konnte. Sie finden Sie in der neuesten Auflage des ersten Intern-Bandes und im GEM-Buch von DATA BECKER, aber auch in einigen anderen Büchern. Legen Sie Ihre Dokumentation ruhig schon einmal bereit!

5.1 TOS - und GEM-Programme

Nachdem ich bisher die Bedeutung von GEM für den Anwender aufgezeigt habe, möchte ich dieses System jetzt aus Sicht des Programmierers darstellen. GEM besteht aus einer sehr umfangreichen Bibliothek von nützlichen Unterprogrammen, den GEM-Funktionen. Diese Funktionen nehmen dem eigentlichen Programm lästige Routinearbeiten ab. So kann die Position des Mauszeigers abgefragt, und mit einem einzigen Funktionsaufruf die File-Selector-Box auf den Bildschirm gezaubert werden. Wie diese Unterprogramme aufgerufen werden, erfahren Sie etwas später. Hier sollen zunächst einmal die prinzipiellen Unterschiede dargelegt werden, die zwischen der reinen TOS-Programmierung und der unter GEM zu beachten sind. Programme, die unter GEM laufen, nennt man übrigens Applikationen (= Anwendungen).

GEM erlaubt einen eingeschränkten Multitasking-Betrieb, das heißt, mehrere Programme dürfen sich gleichzeitig im Rechner befinden und auch gleichzeitig laufen. Eingeschränkt ist dieses System deshalb, weil ein Programm eine wichtigere Stellung einnimmt als alle anderen: die gerade laufende Applikation. Alle anderen Programme sind Accessories. In ei-

nem solchen System ist es natürlich sehr wichtig, daß sowohl Speicherplatz als auch Rechenzeit optimal auf die einzelnen Programme verteilt werden. Während alle Speicherplatz-Fragen von TOS und nicht von GEM geklärt werden (Kapitel 1.5), hilft uns GEM bei der Rechenzeit. Wie hat man sich das vorzustellen?

Nehmen wir an, eine Applikation soll auf einen Tastendruck warten. Wenn Sie das Kapitel 1 gelesen haben, so könnten Sie auf die Idee kommen, die GEMDOS-Funktion Cconin() zu bemühen. Doch unterbricht diese Funktion (wie übrigens alle TOS-Funktionen) das ganze System (also alle im Multitasking laufenden Programme), bis das gewünschte Ereignis, nämlich das Drücken einer Taste, eingetreten ist. Ein Drucker-Spooler, der vielleicht als Accessroy installiert ist, kann seinen Text in dieser Zeit also nicht loswerden. Da in GEM-Applikationen auch auf ganz andere Ereignisse gewartet wird (z.B. Mausklick oder Anklicken eines Menüpunktes) und eigentlich ständig auf irgendetwas gewartet werden muß, gehört die Ereignisverwaltung zu den wichtigsten Aufgaben des GEM, mit der wir uns noch sehr ausführlich beschäftigen werden. Hier gibt es nämlich z.B. eine Funktion, die wie Cconin() auf einen Tastendruck wartet. Nur ist die GEM-Funktion so schlau und läßt in der Zeit, die die Applikation auf den Benutzer wartet, ein Accessory arbeiten.

Doch bevor ich weiter auf einzelne Funktionen eingehe, sollten Sie zunächst einmal die grobe Struktur des GEM kennenlernen.

Da gilt es, zwei große Teile zu unterscheiden:

AES: (Application Environment Services)

Hier findet sich alles, was ein Programm zu einer richtigen GEM-Applikation macht: Fenster, Menüzeilen, Alert- und Dialogboxen. Außerdem ist die Ereignisverwaltung hier untergebracht.

VDI: (Virtual Device Interface)

Hier sind die Grafik-Funktionen des GEM versammelt. Alle Text- und Grafik-Ausgaben einer Applikation sollten mit VDI-Routinen vorgenommen werden. Auch das AES bedient sich dieser Funktionen.

Bevor ich Ihnen ein Beispiel geben kann, muß ich erst noch erklären, wie die ganzen Funktionen aufgerufen werden.

5.2 Funktionsaufrufe

Ich könnte es mir leicht machen und sagen, daß eine GEM-Funktion über einen Namen aufgerufen wird und Parameter in Klammern erhält, ähnlich wie das bei den TOS-Funktionen in C der Fall ist. Dann könnten Sie zwar damit arbeiten, doch sollten Sie (bzw. müssen Sie, falls Sie in Assembler programmieren) schon etwas mehr über die Aufrufe erfahren.

Wollen Sie eine der vielen GEM-Funktionen aufrufen, so müssen Sie zunächst wissen, ob es sich um eine AES- oder eine VDI-Funktion handelt, was mit der GEM-Dokumentation leicht festzustellen ist. Bevor GEM aktiviert wird, wird diese Information irgendwo vermerkt. Alle anderen Informationen werden in sogenannten Parameter-Arrays hinterlegt. Das sind verschieden große Speicherbereiche, in denen vor dem Aufruf Parameter abgelegt werden können und nachher eventuelle Rückgabewerte der Funktion zu finden sind. Schauen wir uns einmal diese Parameter-Arrays bei einem AES-Aufruf an:

control-Array:

Hier wird die Nummer der gewünschten AES-Funktion eingetragen. Außerdem muß hier vermerkt werden, wie viele Parameter in den anderen Arrays übergeben werden sollen.

global-Array:

(Streng genommen kein Array, sondern ein Feld.) Hier werden keine Parameter übergeben, sondern nur verschiedene Informationen untergebracht, die aber zum größten Teil uninteressant sind. Ich werde bei Gelegenheit auf dieses Feld eingehen.

intin-Array:

Wie die Abkürzung vermuten läßt, werden hier Parameter verstaut, die zwei Bytes groß sind (Integer = Worte). Die Bedeutung der einzelnen Worte hängt von der jeweiligen Funktion ab und ist deren Dokumentation zu entnehmen.

intout-Array:

Hier findet der Programmierer nach Abschluß der Funktion alle Worte, die von der Funktion zurückgegeben wurden, z.B. wenn Daten wie die Mauskoordinaten abgefragt wurden.

addrin-Array:

Wie intin, aber für Parameter, die vier Bytes umfassen (Adressen = Langworte).

addrout-Array:

Wie intout, aber ebenfalls für Adressen. Dieser Aufstellung können Sie also entnehmen, daß die gewünschte Funktion durch das control-Array bestimmt wird, weil dort die Funktionsnummer eingetragen wird. Die anderen Einträge im control-Array (die Zahl der Parameter in den anderen Arrays) ist wie die Funktionsnummer in der Regel durch die gewünschte Funktion vorbestimmt, da eine Funktion fast immer gleich viele Parameter erwartet. Die eigentlichen Parameter an die Funktion werden dann je nach Größe entweder im intin- oder im addrin-Array abgelegt. Funktionswerte bzw. -ergebnisse (GEM-Funktionen geben oft eine Menge von Werten zurück) findet man nach dem Aufruf in den Arrays intout und addrout. Das global-Array schließlich dient hauptsächlich internen Zwecken und wird nur ganz selten von der Applikation selbst benutzt.

Die Sache hat einen Haken: Das ganze Felder-System ist furchtbar umständlich für den Programmierer. Denken Sie nur daran, wie viele Zuweisungen an Array-Elemente im Programmtext stehen müssen, bevor eine Funktion ordnungsgemäß aufgerufen werden kann. Außerdem müssen Sie jedesmal nachschlagen, in welches Element eines Arrays welcher Parameter gehört; behalten kann das nämlich niemand. Viel einfacher wäre es doch, die Funktionen statt über eine Nummer und mit Parameter-Arrays über ihren Namen aufzurufen, dem Parameter folgen können - Hochsprachen-Unterprogramme werden schließlich auch so aufgerufen! Die Lösung heißt: Bindings. Was verbirgt sich hinter diesem Begriff? Bindings (in unserem Fall spricht man von GEM-Bindings) sind kleine Unterprogramme. Sie haben Namen, die mit denen der GEM-Funktionen identisch sind. Und ihnen werden Parameter übergeben, die die betreffende GEM-Funktion auch benötigt. Die Unterprogramme (die Bindings also) füllen nun die GEM-Parameter-Arrays mit den übergebenen Parametern; außerdem setzen sie die Funktionsnummer und die Zahl der Parameter im control-Array. Dann rufen Sie das GEM auf, und wenn dieses seine Arbeit beendet hat, werden, falls vorhanden, Rückgabewerte an den Aufrufer zurückgegeben.

Dank dieser Bindings ist auch der Aufruf von GEM-Funktionen in den beiden BASIC-Interpretern und in C fast gleich. Wer sich allerdings für Assembler entschieden hat, muß auf diese Annehmlichkeit gewöhnlich verzichten; die wenigsten Assembler bieten serienmäßig Bindings an, was wohl einen wichtigen Grund auch in der Sprache selbst hat: In Assembler gibt es nun mal keine bequemen Unterprogrammaufrufe mit Parametern und Funktionswert, wie man es von Hochsprachen gewöhnt ist. Nur mit Makros kann platzsparend gearbeitet werden, doch dann sind Programme wiederum nicht auf verschiedenen Assemblern assemblierbar.

Betrachten wir noch schnell die Parameter-Arrays, die für einen VDI-Aufruf erforderlich sind. Da wäre zunächst das contrl-Array (jawohl, das o fehlt). Es ist praktisch mit dem control-Array vom AES gleichzusetzen. Intin und intout gibt es auch im VDI. Addrin und addrout entfallen, statt dessen gibt es die Arrays ptsin und ptsout. Pts steht für Points, also Punkte, oder besser gesagt: Koordinaten. Jeder Eintrag in diesen beiden Arrays ist vier Bytes groß: je zwei Bytes für die x- und die y-Koordinate eines Bildschirmpunktes.

Jetzt möchte ich Ihnen noch zeigen, welche Unterschiede in den verschiedenen Programmiersprachen zu beachten sind. Danach - ich verspreche es - bekommen Sie dann endlich ein Beispielprogramm.

5.3 GEM in verschiedenen Sprachen

Grundsätzlich sind die GEM-Aufrufe genormt. Im Klartext heißt das, daß die C-Aufrufe der Funktionen als Standard gelten. Das schließt jedoch nicht aus, daß es hier und da einmal Abweichungen von diesem Standard geben kann. Das gilt besonders für die Parameter der BASIC-Bindings. Deshalb sollten Sie bei jeder neu vorgestellten Funktion (falls Sie in BASIC arbeiten) unbedingt im Handbuch nachschauen, ob es Änderungen gegenüber dem C-Standard gibt, an den ich mich bei der Beschreibung halten werde. Das wird jedoch nur selten der Fall sein.

Einige grundsätzliche Unterschiede sollen hier jedoch kurz angesprochen werden:

5.3.1 GFA-BASIC

In GFA-BASIC sind alle AES-Funktionen über ihren Namen zu erreichen. Parameter werden in Klammern übergeben. Bei Zahlen ist zu be-

achten, daß je nach Bedarf 2-Byte- oder 4-Byte-Integervariablen bzw. Konstanten übergeben werden: Adressen benötigen 4 Bytes, während Integerwerte (z.B. Indizes) mit 2 Bytes auskommen. Es ist aber durchaus zulässig, statt einer 2-Byte-Variablen eine mit 4 Bytes anzugeben, wenn der Inhalt dieser Variablen auch in eine kleine Variable paßt. Wo Strings benötigt werden, sollten auch solche übergeben werden und nicht, wie in C und Assembler, Zeiger auf solche.

Die VDI-Funktionen sind nicht implementiert, jedenfalls nicht direkt. Doch alles, was Sie mit dem VDI in anderen Sprachen machen können, ist auch mit den speziellen Grafik- und Text-Ausgaberoutinen des GFA-BASIC möglich - Sie erinnern Sich daran, daß das AES ein Programm zu einer GEM-Applikation macht, während das VDI "nur" einige sehr allgemeine Ausgabefunktionen bereitstellt. Auch ohne direkten Zugang zu den VDI-Routinen können Sie also in GFA-BASIC alle Register ziehen.

5.3.2 Omikron-BASIC

Omrikon-BASIC enthält gar keine GEM-Funktionen wie GFA-BASIC. Statt dessen können Sie die nötigen Bindings bei Bedarf von Diskette laden. Jede GEM-Funktion ist dann ein kleines Unterprogramm in Ihrer Applikation.

Zum Laden geben Sie ein:

```
LOAD "GEMLIB.BAS"
```

Anschließend können Sie alle AES- und VDI-Funktionen innerhalb eines Programms, aber auch im Direktmodus benutzen. Für Parameter gilt das gleiche wie unter 5.3.1 für GFA-BASIC gesagt.

Das hat einen Nachteil: Speichert man diese Programme wieder ab, so werden jedesmal die kompletten Bindings mit abgespeichert. Da jedoch meist nur ein kleiner Teil der Bindings benötigt werden wird, befindet sich auf der Omikron-Diskette ein Programm namens GEMSEL und eine Anleitung dazu. Dieses nützliche Programm ist eine Art Linker für BASIC - es schaut nach, welche GEM-Funktionen in einem anzugebenden Programm aufgerufen werden, und hängt diesem Programm die benötigten Binding-Unterprogramme aus der GEMLIB-Datei an.

In diesem Buch werden aus Platzgründen keine Bindings mit abgedruckt und auch nicht in die BASIC-Programme auf der Diskette geschrieben.

Bevor Sie die Omikron-Beispiele laufen lassen können, müssen Sie also entweder im Interpreter das jeweilige Programm mit der Datei "GEM-LIB.BAS" mischen oder aber GEMSEL vor dem Aufruf des Programms starten.

Übrigens weichen die Funktionsaufrufe in Omikron-BASIC etwas vom Standard ab. Zwar heißen die Funktionen genauso wie in C, doch ist es üblich, daß ein Ergebnis oder Rückgabewert nicht als Funktionswert zurückgegeben wird, sondern an eine Variable, die als Parameter übergeben wird. Streng genommen sind es also keine Funktionen, sondern nur noch Prozeduren. Auch sind für verschiedene Anwendungsmöglichkeiten einer Funktion manchmal verschiedene Formen verwendet, die den Aufruf wesentlich vereinfachen können. Für die Funktion dieser Routinen hat das jedoch keine Bedeutung, es ist lediglich eine andere Syntax, so daß Sie Ihr BASIC-Handbuch am besten neben Ihr GEM-Buch legen.

5.3.3 C

Jeder gute C-Compiler besitzt eine Library, in der neben den Standard-C-Funktionen (printf, scanf...) auch GEM-Bindings für AES und VDI enthalten sind. Beim Linken werden diese Bindings automatisch in das fertige Programm mit eingebunden. Sie können sie daher benutzen, als wären sie fester Bestandteil des C-Sprachumfangs.

Wenn Ihr Programm VDI-Funktionen benutzen möchte, so müssen Sie jedoch die VDI-Parameter-Arrays innerhalb Ihrer Applikation selbst deklarieren. Die Bindings greifen dann automatisch darauf zu. Das sieht so aus:

```
int contrl[12],
    intin[128],
    ptsin[128],
    intout[128],
ptsout[128];
```

Im Buch "C-Know-How", ebenfalls bei DATA BECKER erschienen, finden Sie alle Library-Funktionen der wichtigsten C-Compiler und einiges mehr beschrieben.

5.3.4 Assembler

Um mit der Tür ins Haus zu fallen: In Assembler müssen Sie alles von Hand machen. Das ist nicht schwer, aber umständlich, leider auch langweilig und deshalb ideal für schwer zu findende Flüchtigkeitsfehler geeignet. Dafür werden Sie natürlich durch besonders schnelle und kurze Applikationen entschädigt, was oft sehr wichtig ist.

Kennen Sie noch die Parameter-Arrays aus Kapitel 5.2? Die brauchen Sie, um in Assembler GEM-Aufrufe durchführen zu können. Parameter, Funktionsnummer und weitere schöne Dinge müssen Sie mit move-Befehlen in die Arrays kopieren. Der eigentliche Aufruf der GEM-Funktionen ist dagegen ganz einfach; er besteht nur aus einem Befehl:

```
trap #2
```

GEM wird also genauso wie GEMDOS, BIOS und XBIOS (Kapitel 1) über einen Trap aufgerufen. Der Unterschied zu den TOS-Funktionen ist dabei nur, daß Parameter und Funktionsnummer nicht auf dem Stack, sondern in den Arrays übergeben werden.

Zwischen dem Füllen der Arrays und dem Trap-Befehl sind allerdings noch zwei Dinge zu tun:

1. GEM muß wissen, ob die gewünschte Funktion (deren Nummer im control-Array (AES) bzw. contrl-Array (VDI) steht) zum AES oder zum VDI gehört. Das ist deshalb nötig, weil VDI und AES nicht über getrennte Funktionsnummern verfügen. Diese Information schreibt man ins Register D0 (Wortlänge). Dabei gilt: $C8 (=dezimal 200) für AES und $73 (=dezimal 115) für VDI.

2. GEM muß außerdem wissen, wo (d.h. an welcher Adresse im Speicher) die Parameter-Arrays überhaupt liegen. Dazu gibt es zwei sogenannte Parameterblöcke: den AES- und den VDI-Parameterblock, kurz AESPB und VDIPB. Die Adresse dieses PBs wird vor dem Trap ins Register D1 gebracht (Langwort). Der PB selber enthält 6 (AES) bzw. 5 (VDI) Zeiger auf die benutzten Arrays.

Ich glaube, daß Sie gerade den letzten Punkt am besten in einem Programmlisting nachvollziehen können. Deshalb folgt jetzt der typische Beginn eines GEM-Assembler-Programms, den ich auch in allen weiteren Programmen als Include-Datei benutzen werde (allein schon, um Platz zu sparen). Er enthält die Berechnung des Speicherbedarfs (siehe Kapitel 1.5), die Initialisierung eines ausreichenden Stacks (4 KBytes sollten ei-

gentlich immer reichen) und die Unterprogramme aes und vdi. Diese können im eigentlichen Programm, das übrigens mit dem Label main beginnen muß, aufgerufen werden, nachdem alle nötigen Daten in den Parameter-Arrays verstaut wurden.

```
        ;
        ; Diese Programmzeilen sollten vor jedes
        ; GEM-Programm gehängt werden. (INCLUDE)
        ;
        ; Assembler   MP  09-10-88    GEM_INIT.Q
        ;

        ; erster Schritt: Speicherbedarf des Programms
        ; berechnen und nicht benötigten Speicher zurückgeben
        ; dabei gleich einen 4 KB Stack einrichten

        movea.l   4(sp),a5          ;4(sp) ist Start der Basepage
        move.l    12(a5),d0         ;Länge des Programmcodes
        add.l     20(a5),d0         ;+ Länge des Data-Segments
        add.l     28(a5),d0         ;+ Länge des BSS-Segments +
        addi.l    #$1100,d0         ;Basepage (256 Bytes)+Stack (4KB)

        move.l    d0,d1             ;Länge plus
        add.l     a5,d1             ;Startadresse
        andi.l    #-2,d1            ;(gerundet)
        movea.l   d1,sp             ;ergibt Stackpointer

        move.l    d0,-(sp)          ;Länge des benötigten Speichers
        move.l    a5,-(sp)          ;Startadresse des Bereichs
        clr.w     -(sp)             ;dummy-Byte ohne Bedeutung
        move.w    #$4a,-(sp)        ;Funktionsnummer mshrink
        trap      #1                ;Trap für GEMDOS
        adda.l    #12,sp

        jmp       main              ;Sprung in die Applikation


        ; Unterprogramme aes und vdi

aes:    move.l    #aespb,d1         ;AES-Parameterblock
        move.w    #$c8,d0           ;Magic-Number für AES
        trap      #2                ;GEM-Aufruf
        rts

vdi:    move.l    #vdipb,d1         ;VDI-Parameterblock
        move.w    #$73,d0           ;Code für VDI
        trap      #2                ;GEM-Aufruf
        rts

        ; Es folgen die Parameterblöcke:
```

```
          .DATA

aespb:    .DC.l control
          .DC.l global
          .DC.l int_in              ;Unterschied zwischen AES- uund
          .DC.l int_out             ;VDI-Integer-Arrays: _
          .DC.l addr_in
          .DC.l addr_out

vdipb:    .DC.l contrl
          .DC.l intin
          .DC.l ptsin
          .DC.l intout
          .DC.l ptsout

          ; Jetzt kommen die eigentlichen Arrays:

          .BSS

global:   .DS.w 16  ;AES
control:  .DS.w 10
int_in:   .DS.w 128
int_out:  .DS.w 128
addr_in:  .DS.l 128
addr_out: .DS.l 128

contrl:   .DS.w 12  ;VDI
intin:    .DS.w 128
ptsin:    .DS.w 128
intout:   .DS.w 128
ptsout:   .DS.w 128

          .TEXT

          .END
```

Auf eine häufige Fehlerquelle möchte ich hier schon hinweisen: So schöne Namen wie int_in und intin oder control und contrl verwechselt man sehr schnell. Wie Sie dem Ende des vorigen Listings entnehmen können, habe ich die kritischen (weil leicht zu verwechselnden) Namen der AES-Felder so gewählt, daß sie immer ein Zeichen länger sind als die Felder des VDI. Also: contrl = VDI, control = AES.

Die letzte Anmerkung betrifft die Art und Weise, wie auf die Arrays zugegriffen wird. In den GEM-Büchern oder im INTERN Band 1 steht nämlich immer, daß z.B. ein Integer-Wert in intin[2] übergeben werden soll. Diese C-übliche Schreibweise besagt, daß der Wert im dritten (0, 1, 2...!) Element des Arrays zu stehen hat. Da jedes Element zwei Bytes

einnimmt (bei Integers), entspricht intin[0] in C also intin in Assembler, intin[1] entspricht intin+2 und intin[2] ist intin+4. Sie sehen also, daß der Index (der in den eckigen Klammern steht) mit 2 zu multiplizieren ist und zur Startadresse des jeweiligen Arrays addiert werden muß. Ausnahme: Im addr_in- und addr_out-Array des AES nimmt jedes Element nicht zwei, sondern vier Bytes in Anspruch. Deshalb müssen Sie den Index hier auch nicht mit zwei, sondern mit vier multiplizieren. Was da rauskommt und zur Startadresse addiert wird, nennt man übrigens Offset.

Ein kleines Problem kann auch hier auftauchen: Während in allen mir bekannten Büchern die VDI-Funktionsaufrufe so beschrieben sind, daß für jeden Parameter genau angegeben wird, in welches Array und an welche Stelle (Index) in diesem Array er einzutragen ist, gibt es bei den AES-Funktionen Ausnahmen. Oft wird nur eine Funktionsnummer angegeben. In diesem Falle müssen Sie den C-Aufruf untersuchen und feststellen, welche Parameter Daten ein- bzw. ausgeben, und welche davon Integer-Werte und welche Adressen sind. Dann dürfte klar sein, daß z.B. die Adressen, die von einer Funktion zurückgegeben, also ausgegeben werden, im addr_out-Array stehen, und zwar in der gleichen Reihenfolge, wie sie im C-Aufruf angegeben sind (von links nach rechts gelesen). Wenn Sie also in C die Integer-Werte a und b an die Funktion übergeben müßten, so schreiben Sie in Assembler:

```
move.w a,int_in
move.w b,int_in+2
```

Der erste benutzte Index ist also die immer Null. Doch auch hier gibt es eine Ausnahme: Die Rückgabe-Parameter im int_out-Array beginnen erst bei int_out[1], also int_out+2. Im ersten Element wird nämlich stets das zurückgegeben, was Sie in C als Funktionswert erhalten würden (und oft gar nicht brauchen).Eine besondere Aufgabe hat hier noch das control-Array. Es muß von Ihnen wie folgt gefüllt werden:

control	Funktionsnummer
control+2	Anzahl der Werte im int_in-Array
control+4	Anzahl der Ausgaben im int_out-Array (einschließlich Funktionswert in int_out[0])
control+6	Anzahl der Adressen im addr_in-Array
control+8	Anzahl der Ausgaben im addr_in-Array

Die Funktionsnummer ist in jedem GEM-Buch angegeben, und die Zahl der Ein- und Ausgabe-Parameter müssen Sie halt zählen.

Ich empfehle Ihnen, sich einmal in den folgenden Beispielprogrammen eine AES-Funktion herauszusuchen und das Füllen der Parameter-Arrays mit der entsprechenden C-Funktion zu vergleichen.

5.4 Die erste Applikation

Sie alle kennen Alert-Boxen. Sie erscheinen meist, um irgendwelche Warnungen oder Fehlermeldungen abzugeben, also zum Beispiel: "Wollen Sie dieses Programm wirklich verlassen? Ja .. Nein".

Eine Alertbox hat drei variable, also vom Programmierer zu bestimmende Bestandteile: Ein Bild, den eigentlichen Text selber und den oder die Knöpfe (engl. Buttons), die der Benutzer anklicken kann.

Bei dem Bild darf es sich um ein Stoppschild, ein Fragezeichen und um ein Ausrufezeichen handeln. Es kann auch ganz weggelassen werden. Wann welches Bild gebraucht wird, sollte klar sein: Eine Bemerkung, die der Benutzer lediglich zur Kenntnis nehmen muß, bekommt ein Ausrufezeichen, eine Sicherheitsabfrage (oder allgemein eine Frage) ein Fragezeichen und eine Meldung, die einen schwerwiegenden Fehler anzeigt, der die gewünschte Aktion unterbrochen hat, ein Stopschild.

Der Text darf bis zu 5 Zeilen lang sein; jede Zeile darf 30 Zeichen enthalten. Die einzelnen Zeilen müssen beim Aufruf der Funktion, die eine Alert-Box darstellt, durch den senkrechten Strich (|), die sogenannte Pipe, getrennt sein. Will man diesen Strich selbst ausgeben, so schreibt man ihn doppelt (|| gibt | aus). Wenn Sie eine Leerzeile schreiben lassen wollen, so müssen Sie daher zwischen die beiden, die Zeilen trennenden Pipes ein Leerzeichen schreiben.

Schließlich können Sie ein bis drei Knöpfe definieren. Jeder Knopf darf maximal 20 Zeichen Text beinhalten, allerdings nur eine Zeile. Auch mehrere Knöpfe werden beim Aufruf durch einen senkrechten Strich getrennt. Ein Knopf darf übrigens der sogenannte Default-Button sein. Default heißt etwa voreingestellt, d.h. dieser Knopf gilt als angeklickt, wenn die Return- oder Enter-Taste betätigt wird. Außerdem wird er dicker umrandet als andere, normale Knöpfe.

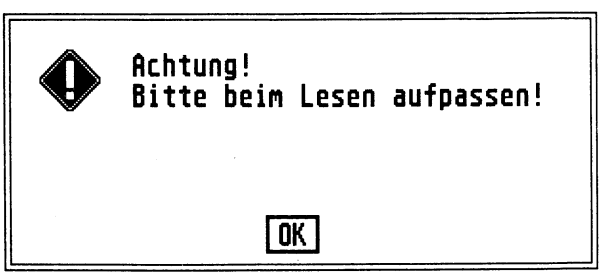
Diese Informationen (über Bild, Text und Knöpfe) müssen vom Programmierer in einem String zusammengefaßt werden, der aus drei Komponenten besteht. Jede Komponente ist in eckige Klammern eingeschlossen.

Die erste Komponente ist nur eine Ziffer zwischen 0 und drei. Es bedeuten: 0=kein Bild, 1=Ausrufezeichen, 2=Fragezeichen und 3=Stopschild. Die zweite Komponente ist der Text, der wie oben beschrieben angegeben werden muß (Zeilen durch | getrennt). Die dritte Komponente bilden die Knöpfe.

Ein Beispiel: Der String

```
[1][Achtung!|Bitte beim Lesen aufpassen!][OK]
```

ergibt folgende Alert-Box:



Was aber, wenn Sie die eckigen Klammern selbst ausgeben wollen? Dazu hat man folgendes vereinbart: Eine doppelte "Klammer zu" (]]) gibt eine einzelne Klammer aus (]), das geht also genauso wie die Ausgabe des senkrechten Strichs, der gewöhnlich die Zeilen trennt. Eine "Klammer auf" ([) kann dagegen ganz normal (also einfach) in den String eingefügt werden, da es nur eine Klammerebene gibt und die Sache somit eindeutig ist.

So, jetzt haben wir den String und brauchen noch die Funktion, an die wir ihn übergeben können und die uns die Alert-Box auf den Bildschirm malt. Sie heißt form_alert. Es ist eine AES-Funktion. Die Vorsilbe form sagt uns, daß sie zur Formularverwaltung gehört - darin ist alles zusammengefaßt, was man für Dialog- und Alert-Boxen benötigt.

Form_alert benötigt aber außer dem String noch einen weiteren Parameter, nämlich die Nummer des Default-Buttons (siehe oben). Dabei gilt: Alle Knöpfe sind von links nach rechts durchnumeriert (linker Knopf: 1); wenn gar kein Knopf Default-Button sein soll, so übergeben Sie eine Null.

Die Funktion ist in dem Moment beendet, in dem der Benutzer einen der Knöpfe angeklickt oder, sofern es einen Default-Button gibt, Return

oder Enter gedrückt hat. Sie gibt natürlich auch einen Wert zurück - wenn der Anwender schon zwischen maximal drei Knöpfen wählen kann, so muß die Applikation nachher auch wissen, für welchen er sich entschieden hat.

Der Aufruf in C sieht dann folgendermaßen aus:

```
angeklickt=form_alert(default,box_string)
```

In GFA-BASIC müssen Sie sich das form_alert großgeschrieben vorstellen, ansonsten ist der Aufruf mit der C-Version identisch. In Omikron-BASIC geht es dagegen so:

```
FORM_ALERT(default,box_string,angeklickt)
```

wobei klar sein sollte, daß angeklickt eine Variable sein muß.

Die Assembler-Programmierer können der GEM-Dokumentation entnehmen, daß der Opcode der Funktion 52 ist, daß je ein Wert in int_in und addr_in übergeben und in int_out zurückgegeben wird, daß in int_in[0] die Nummer des Default-Buttons und in addr_in[0] die Startadresse des Box-Strings übergeben werden müssen und daß nach dem Aufruf in int_out[0] der angeklickte Knopf abgefragt werden kann. Für sie sieht der Aufruf deshalb etwas umfangreicher aus:

```
move.w   #52,control          ;Funktionsnummer
move.w   #1,control+2         ;control[1]: Anzahl der Parameter im
                              ;int_in-Array
move.w   #1,control+4         ;control[2]: dto für int_out
move.w   #1,control+6         ;control[3]: für addr_in
clr.w    control+8            ;control[4]: für addr_out (keine)

move.w   #default,int_in      ;int_in[0]
move.l   #box_string,addr_in  ;addr_in[0]

jsr      aes                  ;AES-Aufruf in der Include-Datei
                              ;GEM_INIT (siehe 5.3.4)

cmpi.w   #...,int_out         ;Verarbeitung des Ergebnisses...
```

Sie sehen: In BASIC und C geht es komfortabler; dafür sind Sie in Assembler schneller (wobei das hier ziemlich egal sein dürfte).

Jetzt könnte man auf die Idee kommen, eine Funktion wie form_alert einfach mal aufzurufen. Sie haben Glück, form_alert ist eine der

Funktionen, die man tatsächlich so aufrufen darf. Dies gilt aber nicht für alle Funktionen. Wenn Sie dann Glück haben gibt's ein paar Bomben; wenn Sie Pech haben, ...

Wie bereits eingangs erwähnt wurde, dürfen in einem GEM-System mehrere Programme gleichzeitig laufen. Der dazu nötige Verwaltungsaufwand erfordert allerdings, daß GEM überhaupt weiß, welche Programme denn gerade laufen. Das erreicht man dadurch, daß jedes GEM-Programm (egal ob Applikation oder Desk-Accessory) sich offiziell an- und auch wieder abmeldet, wenn es beendet ist. Beim Anmelden geschieht auch gleich noch etwas: Jedes Programm erhält eine Applikations-Identifikationsnummer. Weil das ein schrecklich langes Wort ist, sagt man dazu auch kurz ap_id. Diese ap_id wird uns erst später interessieren; merken Sie sich nur, daß man beim Anmelden eine solche Zahl zugeordnet bekommt.

Die Funktionen, die das An- und Abmelden besorgen, heißen appl_init und appl_exit. Wie Sie sich sicher denken können, dient appl_init zum Anmelden (initiieren), während appl_exit zum Abmelden (exit = Ende, Ausgang) benutzt wird (was übrigens nicht Programmende bedeutet). Parameter werden keine Übergeben. Appl_init gibt die ap_id des Programms zurück. Beide Funktionen gehören übrigens wie schon form_alert zum AES.

Unser erstes Programm wird nun aus drei AES-Aufrufen bestehen: Einem appl_init, dem form_alert-Aufruf und schließlich einem appl_exit. Die ap_id brauchen wir in diesem Beispiel genauso wenig wie den Funktionswert von form_alert, also die Information, welcher Knopf angeklickt wurde (wir haben nur einen in dieser Box). In GFA-BASIC werden Sie deshalb vor dem Funktionsnamen ein VOID finden, was bedeutet, daß der Funktionswert, der eigentlich einer Variable zugewiesen oder ausgegeben werden müßte, einfach weggeschmissen wird.

GFA-BASIC

```
'
' Mini-GEM-Applikation zur Demonstration
' GFA-BASIC   MP  09-10-88   DEMOAPP.GFA
'
VOID APPL_INIT()   ! Anmeldung beim AES
'
VOID FORM_ALERT(1,"[1][Hallo!|Das ist eine GEM-Applikation!][Aha]")
'
```

```
VOID APPL_EXIT()   ! und wieder Abmelden
'
END
```

Omikron-BASIC

```
'
' Mini-GEM-Applikation zur Demonstration
' Omikron-BASIC  MP 9-10-88  DEMOAPP.BAS
'
Appl_Init' Anmelden beim AES
'
' Anzeigen der Warnmeldung:
'
FORM_ALERT (1,"[1][Hallo!¦Das ist eine GEM-Applikation.][Aha]",Dummy)
'
Appl_Exit' Abmelden
'
END
```

C

```
/*********************************************/
/*   Mini-GEM-Applikation zur Demonstration   */
/*   Laser C      MP 08-10-88     DEMOAPP.C   */
/*********************************************/

char s[80];           /* Hilfs-String; hier wird der von form_alert */
          /* erwartete String reingeschrieben (muß nicht sein, hier */
                        /* nur, um die Zeilenlänge kurz zu halten */

main()
{
  appl_init();                                  /* Anmelden beim AES */

  strcpy (s, "[1][Hallo!¦Das ist eine GEM-Applikation!][Aha]");

  form_alert (1, s);       /* Anzeigen einer Alertbox und Warten auf */
                                          /* Anklicken des Knopfes */

  appl_exit();                                  /* Abmelden beim AES */
}
```

Assembler

```
          ;
          ; Mini-GEM-Applikation zur Demonstration
          ; Assembler    MP 09-10-88     DEMOAPP.Q
          ;
```

```
gemdos  = 1             .INCLUDE 'GEM_INIT.Q'

        ;appl_init

main:   move.w    #10,control       ;Funktionsnummer: 10
        clr.w     control+2         ;0 Einträge in int_in
        move.w    #1,control+4      ;1 Eintrag in int_out
        clr.w     control+6         ;0 Einträge in addr_in
        clr.w     control+8         ;0 Einträge in addr_out
        jsr       aes               ;Aufruf der Funktion
        ;(der Rückgabewert ap_id interessiert uns nicht)

        ;Darstellen einer Alert-Box mit form_alert

        move.w    #52,control       ;Funktionsnummer
        move.w    #1,control+2      ;...
        move.w    #1,control+4
        move.w    #1,control+6
        clr.w     control+8

        move.w    #1,int_in         ;erster Knopf ist default
        move.l    #al_box,addr_in   ;Inhalt der Box
        jsr       aes

        ;appl_exit

        move.w    #19,control       ;Funktionsnummer
        clr.w     control+2
        move.w    #1,control+4
        clr.w     control+6
        clr.w     control+8
        jsr       aes

        clr.w     -(sp)             ;GEMDOS Funktion Pterm0
        trap      #gemdos           ;beendet das Programm 'richtig'

        .DATA

al_box: .DC.b "[1][Hallo|Das ist eine GEM-Applikation!][Aha]",0
        .END
```

5.5 VDI-Aufrufe

Die Parameter-Arrays für VDI-Funktionen wurden ja schon vorgestellt. Es gibt allerdings auch für das VDI eine Art Initialisierungsfunktion, die ähnlich wie appl_init vor dem ersten Gebrauch einer VDI-Funktion aufgerufen werden muß. Dies geschieht allerdings nicht mehr zu dem

Zweck, der Applikation eine Nummer zur Identifizierung zu geben, sondern ein Ausgabegerät anzumelden. Die Entwickler von GEM waren nun nicht unbedingt Anhänger einer ausgeprägten Bürokratie, denn dieses offizielle An- und Abmelden hat auch seine Vorteile.

Woran denken Sie beim Stichwort Grafik-Ausgabe-Gerät? Doch sicher zunächst an Ihren Monitor, wobei gleichgültig sein soll, ob es ein monochromer oder farbiger ist. Doch es gibt noch andere Geräte: Drucker, Plotter, Diabelichter, Datei... Wenn Sie nun einwenden, daß es es für den ST noch gar keinen Diabelichter gibt, so lassen Sie mich sagen, daß das komplette GEM nicht etwa auf einem Atari ST, sondern auf MS-DOS-Geräten entwickelt wurde. Und für die gibt es solche Geräte durchaus. Der Vollständigkeit halber sei hier auch bemerkt, daß es ein Programm namens GDOS.PRG gibt, das Gerätetreiber für beliebige Geräte nachladen kann, die dann zumindest theoretisch jeder Applikation zur Verfügung stehen (wird auf dem Atari nur von wenigen Programmen ausgenutzt, z.B. GEM-Draw und BeckerPage).

Wie dem auch sei, mit diesem Ballast haben wir uns abzufinden. Bevor der erste Punkt auf dem Bildschirm (belassen wir es einmal bei diesem als Ausgabegerät) erscheinen kann, müssen wir ihn beim VDI anmelden. Wir erhalten, wie schon bei appl_init die Identifikationsnummer, einen Wert zurück, das sogenannte Grafik-Handle, kurz Handle genannt. Sie kennen Handles schon aus dem Kapitel 1 über GEMDOS; dort repräsentierte ein Handle eine Datei; das Handle erhält man, wenn die Datei geöffnet wird, und man benötigt es bei allen folgenden Zugriffen auf die Datei. Im VDI ist es ähnlich, dort steht ein Handle für ein Ausgabegerät. Das Handle erhalten Sie beim Öffnen oder Anmelden des Ausgabegeräts. Und Sie benötigen es ebenfalls bei allen weiteren VDI-Aufrufen, die sich auf dieses Ausgabegerät beziehen. Zwar hat das Handle auch auf dem Atari eine sehr nützliche Funktion, doch die kann ich Ihnen erst am Ende des Kapitels 5.5.2 verraten.

Die Funktion, die ein Ausgabegerät freigibt, das heißt öffnet, heißt v_opnvwk. Das v steht für VDI, und der Rest bedeutet "OPeN Virtual screen WorKstation", also etwa "Öffne virtelle Bildschirm-Arbeitsstation". Da die deutsche Übersetzung schlimmer als die englische Abkürzung klingt, merken Sie sich am besten nur, daß mit dieser Funktion der Bildschirm für Grafik-Ausgaben vorbereitet wird. Diese Funktion gibt das Grafik-Handle zurück.

So wie Sie Dateien irgendwann einmal schließen müssen, muß auch ein Grafik-Ausgabegerät vor dem Verlassen einer Applikation geschlossen

werden. Die entsprechende Funktion heißt v_clsvwk, was nicht schwer zu verstehen ist, wenn man weiß, daß schließen auf Englisch to CLoSe heißt.

Betrachten wir nun noch, wie die beiden Funktionen in verschiedenen Sprachen benutzt werden.

In GFA-BASIC habe ich es leicht - dort gibt es keine VDI-Funktionen. Das heißt: Es gibt sie schon, nur ist ihre Anwendung genauso umständlich und unübersichtlich wie in Assembler, und zweitens ist es vollkommen unnötig, auf VDI-Routinen zurückzugreifen; alles, was das VDI kann, können Sie viel einfacher auch mit richtigen BASIC-Anweisungen machen. Deshalb werden auch in den folgenden Beispielprogrammen keinerlei VDI-Aufrufe in den GFA-BASIC-Programmen zu finden sein. Sie sollten die Lektüre daher im Kapitel 5.6 fortsetzen.

In Omikron-BASIC haben Sie es leicht - die GEM-Bindings enthalten die Befehle V_OPNVWK und V_CLSVWK (beide ohne Parameter). Sie entsprechen in ihrer Funktion den oben genannten Erläuterungen, sind aber einfacher zu handhaben. Statt bei jedem Aufruf einer VDI-Funktion das Grafik-Handle als Parameter zu übergeben, wird es beim V_OPNVWK-Aufruf intern gespeichert und bei Bedarf automatisch in das contrl-Array geschrieben.

In C wird die Sache etwas komplizierter. Auch hier benutzen Sie die beiden Funktionen; allerdings müssen Sie zwei Arrays und eine Variable (in der das Handle zurückgeliefert wird, deshalb schreibt man ein & vor den Variablennamen) übergeben. Die beiden Arrays heißen work_in und work_out. Sie sollten folgendermaßen deklariert sein:

```
int  work_in[11];
int  work_out[57];
```

Welche Bedeutung die Elemente der Arrays im einzelnen haben, können Sie dem ersten Band entnehmen. Work_in wird für Parameter benutzt, die an die Funktion übergeben werden sollen, work_out für Werte, die die Funktion zurückgibt. Work_in muß von Ihnen vor dem Aufruf initialisiert werden. Dazu setzen Sie die Elemente 0 bis 9 auf 1 und das Element 10 auf 2. Das Element 10 ist das sogenannte Koordinatenflag; hier Näheres zu diesen Parametern zu sagen, wäre sinnlos, das können Sie woanders nachlesen - uns kann hier jedenfalls egal sein, was diese Zahlen bedeuten. Gleiches gilt auch für die Ausgaben dieser Funktion, die Sie nach Abschluß der Funktion im Array work_out[] finden werden.

In Assembler sieht die Sache eigentlich genauso aus wie in C, nur daß natürlich keine Arrays übergeben werden, sondern brav ein Wert nach dem anderen direkt in die echten Parameter-Arrays (intin usw.) geschaufelt wird. Wie das genau geht, sehen Sie sich am besten im bald folgenden Programm an.

Für die C-Fans und Assembler-Programmierer habe ich je eine kleine Include-Datei, die die Unterprogramme gem_init und gem_exit enthält. Sie melden das Programm sowohl beim AES als auch beim VDI an bzw. ab. In globalen Variablen (C) beziehungsweise im .BSS-Segment (Assembler) werden folgende Daten festgehalten (Integer bzw. Wortlänge):

ap_id: Identifikationsnummer der Applikation (von appl_ init)

handle: VDI-Grafik-Handle (von v_opnvwk)

x_max: größte x-Koordinate (z.B. 639 bei Mono-Monitor)

y_max: größte y-Koordinate

Die beiden letzten Werte findet man übrigens unter den sehr zahlreichen Ausgaben der v_opnvwk-Funktion.

Hier nun die Listings:

C

```
/*****************************************************************/
/*   Include-Datei für Standard-Anmeldeverfahren (AES und VDI)   */
/*   Megamax Laser C          MP  13-10-88          GEM_INEX.C   */
/*****************************************************************/

int  contrl[12],       /* Diese Felder müssen IMMER in Programmen */
     intin[128],          /* deklariert sein, die VDI-Funktionen */
     ptsin[128],                          /* benutzen wollen */
     intout[128],
     ptsout[128];

int  ap_id,                    /* Werte von allgemeiner Bedeutung */
     handle,                             /* für jede Applikation */
     x_max,
     y_max;


void gem_init()              /* Wird zu Beginn einmal aufgerufen */
{
```

```
int  work_in[11],              /* Diese Arrays werden für v_opnvwk */
     work_out[57],                                    /* benötigt */
     i;               /* Eine Laufvariable brauchen wir auch noch */

  ap_id = appl_init();                        /* Anmeldung beim AES */

  for (i=0; i<10; work_in[i++] = 1);         /* [0-9] auf 1 setzen */
  work_in[10] = 2;         /* Koordinatenflag, sollte immer 2 sein */

  v_opnvwk (work_in, &handle, work_out);     /* Anmeldung brim VDI */

  x_max = work_out[0];                         /* Auflösung merken */
  y_max = work_out[1];
}

void gem_exit()               /* Diese Funktion einmal am Ende der */
{                                          /* Applikation aufrufen */
  v_clsvwk (handle);
  appl_exit();
}
```

Assembler

```
;
; Include-Datei für An-/Abmeldung bei AES und VDI
; Assembler        MP  13-10-88        GEM_INEX.Q
;

;Diese Datei kann in eigenen Applikationen mit Include
;verwendet werden. Die Include-Anweisung sollte der
;erste Befehl im Assembler-Quelltext sein.


;Ganz zu Anfang wird die Datei GEM_INIT geladen,
;die nicht benötigten Speicherplatz freigibt und
;Unterroutinen für AES- und VDI-Aufrufe bereitstellt.
;Außerdem werden hier die GEM-Parameter-Arrays
;angelegt.

.INCLUDE 'GEM_INIT.Q'

;Nach der Initialisierung steht in der Include-Datei ein JMP
;main. Dieses Label muß vom Programmierer am Beginn der
;Applikation gesetzt werden. Als erstes ist dann das Unterpro
;gramm gem_init aufzurufen.


.TEXT
```

```
gem_init: ;vor dem ersten GEM-Call aufrufen

          ;Anmeldung beim AES (appl_init):

          move.w    #10,control      ;appl_init (AES)
          clr.w     control+2
          move.w    #1,control+4
          clr.w     control+6
          clr.w     control+8
          jsr       aes
          move.w    int_out,ap_id    ;Identifikation merken

          ;Bildschirm als Arbeitsstation anmelden (VDI):

          moveq.l   #18,d0           ;intin vorbereiten
          lea.l     intin,a0
gi_lp:    move.w    #1,0(a0,d0.w)    ;Element auf 1 setzen
          subq.w    #2,d0            ;voriges Element
          bpl.s     gi_lp            ;Ende?
          move.w    #2,20(a0)        ;Koordinatenflag (immer 2)

          move.w    #100,contrl      ;v_opnvwk (VDI)
          clr.w     contrl+2
          move.w    #12,contrl+4
          move.w    #11,contrl+6
          move.w    #45,contrl+8
          jsr       vdi
          move.w    contrl+12,handle ;VDI-Grafik-Handle

          move.w    intout,x_max     ;Auflösung speichern
          move.w    intout+2,y_max
          rts


gem_exit: ;vor Verlassen des Programms aufrufen

          move.w    #101,contrl      ;v_clsvwk (VDI)
          clr.w     contrl+2
          clr.w     contrl+4
          clr.w     contrl+6
          clr.w     contrl+8
          move.w    handle,contrl+12
          jsr       vdi

          move.w    #19,control      ;appl_exit (AES)
          clr.w     control+2
          move.w    #1,control+4
          clr.w     control+6
          clr.w     control+8
          jsr       aes
```

```
        rts

        .BSS

ap_id:  .DS.w 1   ;ap_id, wird von appl_init geliefert

handle: .DS.w 1   ;VDI-Grafik-Handle

x_max:  .DS.w 1   ;Bildschirmauflösung, erfährt man
y_max:  .DS.w 1   ;bei v_opnvwk

        .END
```

Diese Include-Dateien nehmen uns Routinearbeit ab und werden deshalb auch in meinen Beispielprogrammen eingesetzt.

5.5.1 Zeichnen mit dem VDI

Jetzt wissen Sie so viel über das VDI, daß einer ersten kleinen Testgrafik nichts mehr im Weg steht. Dazu schlagen Sie bitte in Ihrer GEM-Dokumentation unter VDI-Ausgabe-Funktionen nach. Hier findet der Programmierer all das, was er an grafischen Grundmitteln benötigt, um beliebige Zeichnungen erstellen zu können. Dazu gehören Polyline (v_pline), Polymarker (v_pmarker), Text (v_gtext), Filled Area (v_filarea), Contour Fill (v_contourfill), Filled Rectangle (v_recfl) und die grafischen Grundfunktionen. Letztere enthalten 10 weitere Ausgabe-Funktionen, die durch eine Unterfunktions-Nummer unterschieden werden. Die Entwickler des GEM allein wissen, warum diese Unterteilung vorgenommen wurde, doch davon merken Sie ohnehin nur dann etwas, wenn Sie die Funktionen in Assembler anwenden möchten. Ich schlage vor, daß Sie sich die Beschreibung der oben genannten Funktionen einmal durchlesen, um einen groben Überblick über die Möglichkeiten der Ausgaberoutinen des VDI zu erhalten.

Fertig? Schön! Bevor wir uns an ein Beispiel heranwagen, muß ich allerdings noch auf zwei kleine Dinge hinweisen.

Das erste betrifft die C-Programmierer. Sie werden vielleicht beim Durchblättern der Dokumentation schon bemerkt haben, daß dort ab und zu von einem sogenannten pxyarray die Rede ist. Dieses Array wird immer dann benutzt, wenn an eine Funktion mehr als ein Koordinatenpaar zu übergeben ist. Das hat nicht nur den Vorteil, daß die Aufrufe kurz und übersichtlich bleiben; vielmehr ermöglicht dieses Vorgehen es, bei einem einheitlichen Aufruf die Zahl der zu übergebenden Koordina-

ten variabel zu halten, was zum Beispiel bei der Funktion Polyline (v_pline) nötig ist. Das Array muß von Ihnen deklariert werden. Ob Sie es standardgemäß pxyarray oder aber "koordinaten", "punkte" oder sonst irgendwie nennen, ist dem VDI jedoch gleichgültig.

Der zweite Hinweis betrifft die Text-Ausgabefunktionen unter Assembler, und zwar gleichermaßen die Funktion Text (v_gtext) wie die Funktion Justified Graphics Text (v_justified). Der auszugebende Text wird Zeichen für Zeichen in den jeweils niederwertigen Bytes des intin-Arrays abgelegt, beginnend bei intin[0] (Funktion v_gtext) bzw. bei intin[2] (also intin+4 in Assembler, gilt für v_justified). Der String muß nicht durch ein Nullbyte abgeschlossen werden; statt dessen ist die Länge des intin-Arrays wie üblich im contrl-Array zu übergeben; der Unterschied zu anderen VDI-Funktionen ist hierbei halt nur, daß die Länge des intin-Arrays variabel ist. C- und BASIC-Programmierer übergeben dagegen einen ganz gewöhnlichen String, der in C wie üblich durch ein Nullbyte abgeschlossen wird. Die GEM-Bindings übertragen diese Strings automatisch in das intin-Array, machen also das, was in Assembler manuell geschieht, automatisch.Jetzt können wir endlich zur Sache kommen. Es soll eine Ellipse in einem Rechteck mit abgerundeten Kanten gezeichnet werden. Die Ellipse soll ausgefüllt sein. Ganz unten am Bildschirm soll auch noch ein kurzer Text erscheinen. Das Ganze bitte so, daß es in jeder Auflösungsstufe läuft. Das ist jetzt kein Problem mehr:

Omikron-BASIC

```
'
' VDI-Demonstration für eine kleine Grafik und Text
' Omikron-BASIC      MP  12-11-88      VDI_DEMO.BAS
'
Appl_Init
V_Opnvwk
CLS
'
' Auflösung berechnen (mit XBIOS 4 Getrez)
'
X_Max%=640:Y_Max%=400
XBIOS (Rez%,4)
IF Rez%<2 THEN Y_Max%=200
IF Rez%=0 THEN X_Max%=320
'
V_Rbox(10,10,X_Max%-10,Y_Max%-30)
V_Ellipse(X_Max%/2,Y_Max%/2-10,X_Max%/2-10,Y_Max%/2-20)
V_Gtext(X_Max%/2-96,Y_Max%-10,"VDI - Grafik ist einfach!")
'
' Warten auf Taste:
'
```

```
GEMDOS (,7)'    Crawcin
'
V_Clsvwk
Appl_Exit
END
```

C

```
/*****************************************/
/*  Kleines Grafik-Demonstrationsprogramm */
/*  Laser C     MP 12-11-88    VDI_DEMO.C */
/*****************************************/

#include "gem_inex.c"
#include <osbind.h>

int  pxyarray[4];   /* Wir benötigen 2 Punkte */
                          /* = 2 Koordinaten */
main()
{
  gem_init();

  Cconws ("\33E");    /* Bildschirm löschen mit */
                            /* Escape-Sequenz */
  pxyarray[0] = 11;
  pxyarray[1] = 10;
  pxyarray[2] = x_max-10;
  pxyarray[3] = y_max-30;
  v_rbox (handle, pxyarray);

  v_ellipse (handle, x_max/2, y_max/2-10, x_max/2-10, y_max/2-20);

  v_gtext (handle, x_max/2-96, y_max-10, "VDI - Grafik ist einfach!");

  Crawcin();          /* wartet auf Taste */

  gem_exit();
}
```

Assembler

Wie ich oben schon sagte, müssen Sie als Assembler-Programmierer den String Zeichen für Zeichen in das intin-Array kopieren. Im folgenden Programm besorgt das das Unterprogramm fix_text, dem als Parameter die Adresse des durch ein Nullbyte abgeschlossenen Strings in A0 übergeben wird.

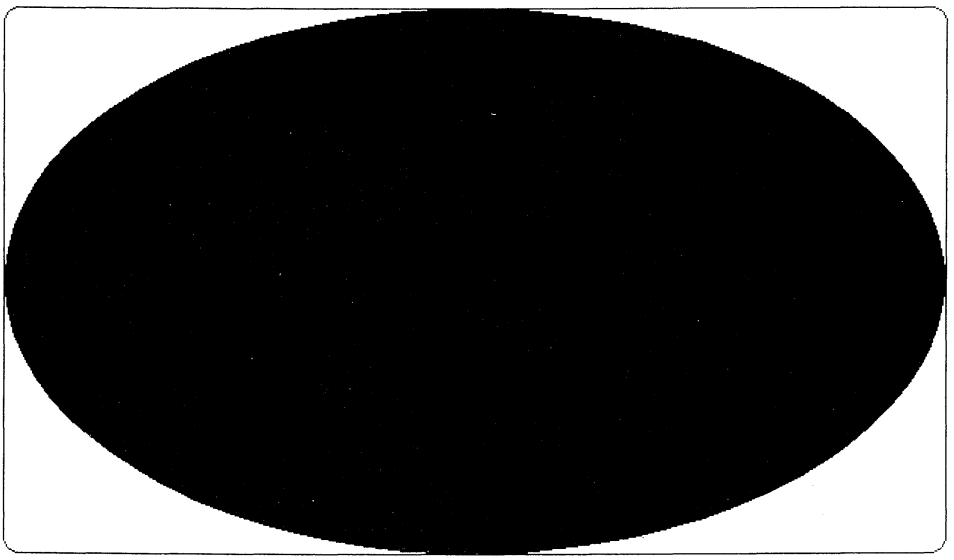
```
          ;
          ; Kleines Grafik-Demonstrationsprogramm

          ; Assembler   MP 16-11-88    VDI_DEMO.Q
          ;

gemdos    = 1
crawcin   = 7
print     = 9

          .INCLUDE 'GEM_INEX.Q'


main:     jsr       gem_init          ;AES und VDI anmelden

          pea       clrscr            ;Bildschirm löschen (VT-52)
          move.w    #print,-(sp)
          trap      #gemdos
          addq.l    #6,sp

          ; Es soll ein Rechteck gezeichnet werden

          move.w    #11,contrl        ;Rounded Rectangle (v_rbox)
          move.w    #2,contrl+2       ;Anzahl der Koordinaten in ptsin
          clr.w     contrl+4          ;Anzahl der Punkte in ptsout
          move.w    #3,contrl+6       ;Länge des intin-Arrays
          clr.w     contrl+8          ;Zahl der Werte in intout
          move.w    #8,contrl+10      ;Funktions-Unter-Nummer
          move.w    handle,contrl+12  ;VDI-Grafik-Handle

          move.w    #10,ptsin         ;Koordinaten für das Rechteck
          move.w    #10,ptsin+2
          move.w    x_max,d0          ;jeweils maximale Koordinate
          subi.w    #10,d0            ;minus 10 (für unteren rechten)
          move.w    d0,ptsin+4        ;Punkt)
          move.w    y_max,d0
          subi.w    #30,d0
          move.w    d0,ptsin+6

          jsr       vdi               ;Rechteck zeichnen

          ; Zeichnen der Ellipse
          move.w    #11,contrl        ;Ellipse (v_ellipse)
          move.w    #2,contrl+2       ;Anzahl der Koordinaten in ptsin
          clr.w     contrl+4          ;Anzahl der Punkte in ptsout
          move.w    #3,contrl+6       ;Länge des intin-Arrays
```

```
clr.w     contrl+8          ;Zahl der Werte in intout
move.w    #5,contrl+10      ;Funktions-Unter-Nummer
move.w    handle,contrl+12  ;VDI-Grafik-Handle

clr.l     d0                ;Koordinaten berechnen
move.w    x_max,d0
divu      #2,d0             ;durch 2 = Bildschirmmitte
move.w    d0,ptsin

subi.w    #10,d0            ;minus 10 ergibt x_max/2-10
move.w    d0,ptsin+4        ;als x-Radius

clr.l     d0
move.w    y_max,d0
divu      #2,d0
subi.w    #10,d0
move.w    d0,ptsin+2

subi.w    #10,d0            ;y_max/2-20 als y-Radius
move.w    d0,ptsin+6

jsr       vdi               ;Ellipse zeichnen

; Text ausgeben

lea.l     ausgabe,a0
bsr       fix_text          ;String ins intin-Array bringen

move.w    #8,contrl         ;Text (v_gtext)
move.w    #1,contrl+2       ;Anzahl der Koordinaten in ptsin
clr.w     contrl+4          ;Anzahl der Punkte in ptsout
clr.w     contrl+8          ;Zahl der Werte in intout
move.w    handle,contrl+12  ;VDI-Grafik-Handle

clr.l     d0                ;Koordinaten berechnen
move.w    x_max,d0
divu      #2,d0
subi.w    #96,d0
move.w    d0,ptsin

move.w    y_max,d0
subi.w    #10,d0
move.w    d0,ptsin+2

jsr       vdi

move.w    #crawcin,-(sp)    ;Warten auf Taste
trap      #gemdos
```

```
        addq.l    #2,sp

        jsr       gem_exit          ;Abmelden bei AES und VDI
        clr.w     -(sp)             ;Programm beenden mit
        trap      #gemdos           ;GEMDOS-Funktion Pterm0


fix_text:
        ; Unterprogramm, das einen String (Startadresse in a0
        ; zu übergeben) in das intin-Array schreibt, die Länge
        ; bestimmt und in contrl[3] ablegt

        clr.w     d0                ;Länge
        clr.w     d1                ;Hilfsregister
        lea.l     intin,a1

fix_loop: move.b  (a0)+,d1          ;ein Byte aus Zielstring holen
        tst.b     d1                ;Stringende?
        beq.s     fix_end
        move.w    d1,(a1)+          ;nein, dann als Wort ins
        addq.w    #1,d0             ;intin-Arrary schreiben
        bra.s     fix_loop

fix_end:  move.w  d0,contrl+6       ;Länge festhalten
        rts


        .DATA

clrscr:   .DC.b 27,'E',0            ;Sequenz für Bildschirm löschen

ausgabe:  .DC.b 'VDI - Grafik ist einfach!',0

        .END
```

Wie Sie gesehen haben, ist dieses kleine Beispielprogramm in Assembler doch nicht ganz so klein. Das liegt zum einen an den Berechnungen der Koordinaten, die sich alle über mehrere Zeilen erstrecken, und natürlich an dem aufwendigeren Aufrufverfahren der Funktionen. Trotz allem ist das lauffähige Programm natürlich wieder das kürzeste und das schnellste.

Das Ergebnis des Programms sollte so aussehen:



VDI - Grafik ist einfach!

5.5.2 Die Attribut-Funktionen

Mit den Attribut-Funktionen können Sie alle Ausgabe-Funktionen, die Sie ja im vorigen Kapitel kennengelernt haben, beeinflussen. Dazu werden vor dem Aufruf der eigentlichen Ausgabe-Funktionen die Attribut-Funktionen bemüht und damit eine Art Voreinstellung für die Ausgabe-Funktionen geschaffen. Die Attribut-Funktionen selbst können also keine Grafik ausgeben.

Auch hier empfehle ich Ihnen, zunächst einmal Ihre GEM-Dokumentation zu durchforsten, damit Sie eine ungefähre Vorstellung von den vielfältigen Möglichkeiten bekommen, die Ihnen diese Funktionen bieten. Anschließend werden wir dann einzelne Funktionen unter die Lupe nehmen.

Zunächst sollte vielleicht etwas zu den scheinbar verwirrenden Funktions-Namen gesagt werden. Bei näherer Betrachtung stellt man nämlich fest, daß die Namen der Attribut-Funktionen nach einem einfachen Schema aufgebaut sind, das sie nicht nur eindeutig von anderen VDI-Routinen unterscheidet, sondern auch eine relativ einfache Entschlüsselung zuläßt. Der erste Buchstabe ist wieder das v für VDI. Der zweite Buchstabe ist ein s für Set ..., was heißt, daß ein Attribut gesetzt werden soll. Daran schließt sich ein Kürzel an, dem Sie entnehmen können,

welches Attribut denn eigentlich geändert werden soll, also z.B. f für Fill (Füllen von Flächen) oder t für Text. Es folgt ein Unterstrich (_) und diesem eine Ergänzung, die das zu ändernde Attribut noch genauer bestimmt. So dient vst_color dazu, die Farbe (Color) für folgende Text-Ausgabe zu bestimmen, während vsf_style ein Füllmuster auswählt. So betrachtet, sind die Funktionsnamen trotz ihrer Kürze sogar sehr aussagekräftig.Beginnen wir mit der Funktion Set Writing Mode (vswr_mode). Von ihr sind alle Ausgabe-Funktionen betroffen (Writing bedeutet hier allgemein Bildschirmausgabe). Für diese Ausgabe soll offensichtlich ein Modus, also eine Betriebsart, bestimmt werden. Davon haben wir, wie uns die GEM-Dokumentation verrät, vier verschiedene zur Auswahl, die von 1 bis 4 durchnumeriert sind:

1. Replace- oder Überschreib-Modus

Dieser Modus ist zu Beginn eines Programms (das heißt: nach der v_opnvwk-Anweisung, also nach dem Öffnen des Bildschirms als Grafik-Ausgabegerät) aktiv. To replace heißt auf Deutsch ersetzen. Auf den Bildschirm übertragen bedeutet das, daß ein alter Bildschirminhalt durch ein neu zu zeichnendes Objekt ersetzt wird.

Dazu ein Beispiel: Der Bildschirm enthält das GEM-typische graue Muster, das Desktop. Jetzt könnten Sie mit der Funktion v_gtext einen Text irgendwo auf diesem Grau ausgeben. Jeder Buchstabe dieses Textes ist dabei für den Rechner als grafisches Objekt nur ein weißes Rechteck mit einigen schwarzen Punkten, aus denen der Mensch als Betrachter dieser Punkte nachher wieder einen Buchstaben macht. Bei der Ausgabe dieses Buchstabes macht der Rechner nun nichts weiter, als das Rechteck in den Bildschirm zu kopieren. Dabei wird jeder alte Bildpunkt des Zielrechtecks auf dem Bildschirm durch den entsprechenden Punkt des vorgegebenen Buchstaben-Rechtecks ersetzt (= replaced), egal, ob der Punkt dieses Buchstaben-Rechtecks schwarz oder weiß war. (Um eventuellen Mißverständnissen vorzubeugen: Ersetzt heißt hier nicht gesetzt, ein weißer Punkt im Buchstaben-Rechteck wird also auch auf dem Bildschirm weiß.) Auf dem Bildschirm erscheint dieser Buchstabe also so, als sei er auf ein weißes Rechteck gezeichnet worden. Das ist etwa so, als wenn Sie aus einer Zeitung ein Wort ausschneiden und es dann auf ein Foto kleben.

2. Transparent-Modus

Greifen wir gleich das Beispiel mit dem Foto, dem Wort und der Zeitung wieder auf: Im Transparent-Modus müßte unsere Zeitung durchscheinend, also transparent sein. Dazu könnten Sie das Wort aus der Zeitung

auf eine Overhead-Folie kopieren und diese Folie auf das Foto legen. Ergebnis: Das weiße Rechteck, das vorher noch scheinbar hinter dem Wort lag, fehlt. Wenn wir diesen Modus wieder im Computer und bei der Zeichen-Ausgabe betrachten, so folgt daraus, daß der Rechner nur noch die schwarzen Punkte des Buchstabens in den Bildschirm kopiert; die Punkte, die im Buchstaben-Rechteck weiß sind, werden im Bildschirm nicht verändert. Wenn wir also auf einen schwarzen Hintergrund einen Text im Transparent-Modus schreiben, so sehen wir gar nichts, denn alle schwarzen Punkte des Textes waren auf dem Bildschirm ja auch vorher schon schwarz. Da die weißen Punkte nicht mitkopiert wurden, hat sich auf dem Bildschirm überhaupt nichts geändert. Wenn dagegen im Transparent-Modus auf eine ganz weiße Fläche geschrieben wird, so hat dies den gleichen Effekt, als hätten Sie es im Replace-Modus gemacht.

3. XOR-Modus

XOR steht für Exklusiv-Oder-Verknüpfung. In unserem Fall bedeutet das, daß der alte Bildpunkt mit dem neu zu setzenden (oder zu löschenden) Punkt exklusiv-oder-verknüpft wird. Also: Sind Hintergrund und neu zu setzender Punkt weiß, so wird auch der neu entstehende Punkt weiß. Das gleiche passiert, wenn alter und zu setzender Punkt schwarz sind. Einen schwarzen Punkt erhalten Sie als Ergebnis nur dann, wenn genau einer der beiden verknüpften Punkte (alter Punkt und zu setzender Punkt) schwarz ist.

Für den XOR-Modus gibt es zwei typische Anwendungen: Zum einen eignet er sich hervorragend zur Beschriftung von Zeichnungen. Stellen wir uns dazu ein schwarzes, ausgefülltes Rechteck auf dem Bildschirm vor. Nun soll ein Text so auf dem Bildschirm erscheinen, daß die eine Hälfte neben dem Rechteck und die andere Hälfte in dem Rechteck (also auf der schwarzen Fläche) liegt. Wenn wir den Text im XOR-Modus schreiben, dann wird die erste Hälfte schwarz geschrieben, während der Teil auf dem schwarzen Hintergrund automatisch weiß erscheint.

Die zweite Verwendungsmöglichkeit ist besonders praktisch. Sie beruht darauf, daß ein zweites Zeichnen desselben Objekts im XOR-Modus die erste Darstellung des Objekts rückgängig macht, wobei der Hintergrund komplett wiederhergestellt wird. Deshalb wird dieser Modus meist in Grafikprogrammen verwendet, z.B. bei der Linienfunktion: Sie klicken den Startpunkt an und ziehen dann die "Gummi"-Linie bis zum gewünschten Endpunkt. Dabei wird eine Zwischenlinie im XOR-Modus gezeichnet. Sobald Sie die Maus ein Stück bewegen, wird die Linie nochmal gezeichnet, sprich: dadurch gelöscht. Dann wird eine neue Linie

vom Startpunkt zu den aktuellen Mauskoordinaten gezeichnet usw. Auf diese Weise muß nie der Hintergrund einer "Gummi"-Linie zwischengespeichert werden.

4. Invers-Transparent-Modus

Dieser Modus ähnelt dem normalen Transparent-Modus. Allerdings wird das zu zeichnende Objekt vor der Bildschirm-Ausgabe invertiert, d.h. aus Schwarz wird Weiß und umgekehrt. In unserem Buchstabenbeispiel ist das zu zeichnende Objekt wieder das Buchstaben-Rechteck, das komplett invertiert wird. Der Buchstabe wird also weiß in einem schwarzen Rechteck erscheinen.

Ein Beispielprogramm möchte ich Ihnen zu diesem Modus nicht geben. Fürs erste werden wir auch keinen besonderen Schreibmodus benötigen. Sie sollten sich jedoch merken, daß es so etwas gibt; bei Bedarf können Sie dann hier noch einmal nachsehen.

Schauen wir uns noch zwei weitere Attribut-Funktionen an: Set Polyline End Styles (vsl_ends) und Set Polyline Line Width (vsl_width). Letztere legt die Breite einer Linie fest, während die erste über das Aussehen von Start- und Endpunkt einer Linie entscheidet, wobei Sie für Start und Ende unterschiedliche Arten wählen dürfen. Zur Wahl stehen kantig, rund und Pfeile. Ob das Ende rund oder kantig ist, können Sie natürlich nur bei etwas dickeren Linien sehen. Die letztgenannte Option bewirkt, daß am Start- bzw. Endpunkt ein Pfeil aufgesetzt wird; aber sehen Sie selbst, was folgendes Beispielprogramm bewirkt:

Omikron-BASIC

```
'
' Linienbreite und Form des Linienendes verändern
' Omikron-BASIC      MP  18-11-88      LINIEN.BAS
'
Appl_Init'     Anmelden beim GEM
V_Opnvwk
'
CLS '           Bildschirm löschen
'
Vsl_Width(33)' Linienbreite: 33 Punkte
'
FOR I=0 TO 2
'
Vsl_Ends(I,I)' Style von 0 bis 2 wechseln
'
Ptsin%(0,0)=20'        Koordinaten müssen bei Omikron
```

```
Ptsin%(1,0)=20+70*I'    in Ptsin%-Array übergeben werden
Ptsin%(0,1)=300
Ptsin%(1,1)=Ptsin%(1,0)
V_Pline(2)'             2 ist die Anzahl der zu verbindenden Punkte
'
NEXT I
'
GEMDOS (,7)'            Warten auf Taste
'
V_Clsvwk'               Abmelden vom GEM
Appl_Exit
END
```

C

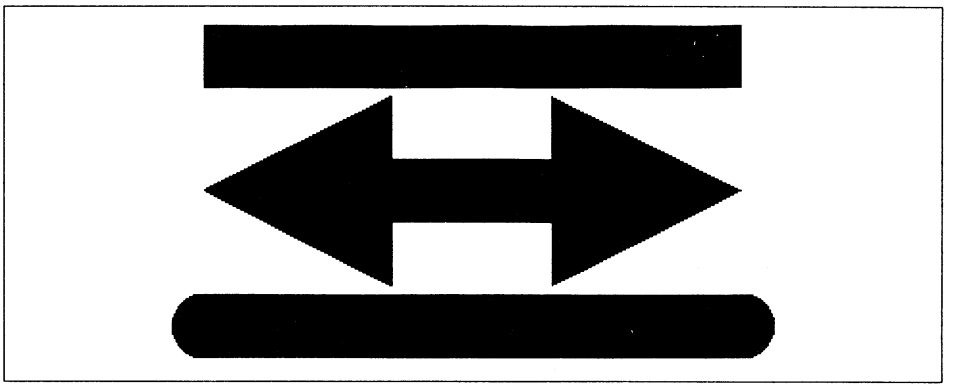
```
/*********************************************************/
/*   Linienbreite und Form des Linienendes verändern   */
/*   Megamax Laser C      MP  18-11-88      LINIEN.C   */
/*********************************************************/

#include <osbind.h>
#include "gem_inex.c"

int  i;
int  pxyarray[4];   /* Dieses Array brauchen wir für Koordinaten */

main()
{
  gem_init();

  Cconws ("\33E");    /* Bildschirm löschen */

  vsl_width (handle, 33);        /* Linienbreite einstellen */
  for (i=0; i<=2; i++)
  {
    vsl_ends (handle, i, i);     /* Linienenden wechseln */

    pxyarray[0] = 20;            /* Koordinaten in Hilfs-Array */
    pxyarray[1] = 20+70*i;
    pxyarray[2] = 300;
    pxyarray[3] = pxyarray[1];

    v_pline (handle, 2, pxyarray);  /* 2 Punkte verbinden */
  }

  Crawcin();     /* Warten auf Taste */

  gem_exit();
}
```

Assembler

```
          ;
          ; Linienbreite und Form des Linienendes verändern
          ; Assembler         MP  18-11-88         LINIEN.Q
          ;

          .INCLUDE 'GEM_INEX.Q'

gemdos    = 1
crawcin   = 7
cconws    = 9

          .TEXT

main:     jsr       gem_init

          pea       clrscr          ;Bildschirm löschen
          move.w    #cconws,-(sp)
          trap      #gemdos
          addq.l    #6,sp

          ; Linienbreite einstellen

          move.w    #16,contrl      ;Funktions-Opcode
          move.w    #1,contrl+2
          move.w    #1,contrl+4
          clr.w     contrl+6
          clr.w     contrl+8
          move.w    handle,contrl+12

          move.w    #33,ptsin       ;Linienbreite
          clr.w     ptsin+2

          jsr       vdi

          clr.w     d3              ;Laufvariable für Schleife

loop:
          ; Linienanfangs- und -endstyle wählen

          move.w    #108,contrl     ;Funktions-Opcode
          clr.w     contrl+2
          clr.w     contrl+4
          move.w    #2,contrl+6
          clr.w     contrl+8
          move.w    handle,contrl+12

          move.w    d3,intin        ;Linienstartpunkt
          move.w    d3,intin+2      ;Linienendpunkt
```

```
        jsr     vdi

        ; Zeichnen der Linie

        move.w  #6,contrl       ;Funktions-Opcode
        move.w  #2,contrl+2     ;2 Punkte verbinden
        clr.w   contrl+4
        clr.w   contrl+6
        clr.w   contrl+8
        move.w  handle,contrl+12

        move.w  #20,ptsin       ;Koordinaten
        move.w  d3,d4
        mulu    #70,d4
        addi.w  #20,d4
        move.w  d4,ptsin+2
        move.w  d4,ptsin+6
        move.w  #300,ptsin+4

        jsr     vdi

        ; Ende der Schleife:

        addq.w  #1,d3
        cmpi.w  #3,d3
        bne     loop

        move.w  #crawcin,-(sp)  ;Warten auf Taste
        trap    #gemdos
        addq.l  #2,sp

        jsr     gem_exit

        clr.w   -(sp)
        trap    #gemdos


        .DATA

clrscr: .DC.b 27,'E',0

        .END
```

So sieht's aus:



An diesen Programmen ist auch die Verwendung der Polyline-Funktion (v_pline) gut zu erkennen: Zunächst werden die Koordinaten aller benötigten Punkte (Polyline kann ja mehr als zwei Punkte verbinden) ins Ptsin-Array geschrieben (Omikron-BASIC bzw. Assembler) oder in einem separaten, selbst zu deklarierenden Array übergeben (C). Die Anzahl der verwendeten Punkte muß explizit angegeben werden, und zwar als Parameter (BASIC und C) oder als Länge des ptsin-Arrays (Assembler).

Ein letzter Hinweis zu den Attribut-Funktionen: Sie dürfen mit gutem Gewissen jede nur denkbare Einstellung der Attribute vornehmen, ohne daß Sie dadurch Accessories beeinflussen könnten, die vielleicht die Standardeinstellung für ihre Ausgaben benötigen. Das liegt daran, daß alle Attribute für jede geöffnete virtuelle Arbeitsstation separat gespeichert werden. Wenn Sie also mit v_opnvwk einen Bildschirm geöffnet haben, so beziehen sich alle folgenden Attribut-Änderungen nur auf den logischen Bildschirm Ihres Programms. Als Gegenleistung müssen Sie halt für alle VDI-Aufrufe ein Handle mitgeben. Jetzt wissen Sie auch, warum.

5.5.3 Die Raster-Funktionen

Die Raster-Funktionen sind sehr komplex und nicht gerade leicht zu verstehen. Deshalb werde ich nur eine einzige Rasterfunktion vorstellen, und auch die wird nicht in allen Einzelheiten erläutert. Ich werde Sie aber so weit führen, daß Sie diese Funktion sinnvoll anwenden können und daß Sie, falls Sie möchten, die anderen Funktionen alleine erforschen können.

Ein Raster ist rechteckiger Grafik-Bereich. Für uns soll zunächst einmal gelten, daß sich dieser Bereich auf dem Bildschirm befindet. Die Rasterfunktion, die ich Ihnen vorstellen möchte, erlaubt es nun, ein solches Raster zu kopieren. Damit sind also Blockoperationen möglich, wie Sie sie sicher von Grafik-Programmen kennen (Block verschieben, Block kopieren). Allerdings ist der Ausdruck kopieren nicht sehr gut gewählt. Eigentlich sollte man sagen: Die Funktion verknüpft ein Quellraster logisch mit einem Zielraster. Es gibt 16 verschiedene Möglichkeiten dieser Verknüpfung, die durch ein Modus-Wort (0-15) unterschieden werden. Die wichtigsten sollen hier kurz vorgestellt werden:

0: Das Zielraster wird (unabhängig vom Quellraster) gelöscht.

3: Das Zielraster wird komplett kopiert. Dieser Modus entspricht dem Replace-Modus (siehe Kapitel 5.5.2, Set Writing Mode).

7: Auch hier wird kopiert, allerdings bleibt der alte Hintergrund des Zielrasters dort, wo das Quellraster weiß ist, erhalten (entspricht dem Transparent-Modus bei Set Writing Mode).

10: In diesem Modus wird lediglich das Zielraster invertiert.

15: Dieser Modus färbt das Zielraster schwarz, ist also das Gegenstück zum Modus 0.

An dieser Liste sehen Sie zwei Dinge: Erstens ist es nicht leicht, die Auswirkungen der verschiedenen Modi in Worten zu beschreiben, und zweitens kann diese Funktion auch bequem dazu benutzt werden, Bildschirmbereiche zu löschen oder zu invertieren.

Betrachten wir jetzt den Aufruf der Funktion: Sie heißt Copy Raster, Opaque (vro_cpyfm). Opaque bedeutet nicht transparent, und wie erwartet gibt es auch eine zweite Rasterfunktion namens Copy Raster, Transparent. Der Unterschied liegt darin, daß letztere Funktion zusätzlich Farbe in ein einfarbiges Quellraster bringen kann. Damit werden wir uns allerdings nicht weiter aufhalten.

Der Funktion werden eine ganze Reihe von Parametern übergeben: Koordinaten für Quell- und Zielraster und das Modus-Wort, das die Art der logischen Verknüpfung bestimmt. Quell- und Zielraster müssen gleich groß sein, d.h. Vergrößerungen oder Verkleinerungen sind mit dieser Funktion nicht möglich. Sollten Sie versehentlich unterschiedliche Größen angeben, so wählt der Rechner die Größe des Zielrasters automatisch

auch als Größe des Quellrasters. Das ist besonders bei den Modi 0, 10 und 15 nützlich (siehe oben), weil dort ja gar kein Quellraster angegeben werden muß.

Zwei weitere Parameter müssen übergeben werden, nämlich Zeiger auf die sogenannten MFDBs, was soviel wie Memory Form Definition Block bedeutet. Das einzig Schöne an diesen MFDBs ist für uns eigentlich die Tatsache, daß wir sie nicht benötigen, solange sich Quell- und Zielraster auf dem Bildschirm befinden. Um dem Rechner das mitzuteilen, müssen wir beim Aufruf statt der beiden Zeiger auf MFDBs zwei Zeiger auf ein Langwort übergeben, das Null ist. Wie das gemacht wird, entnehmen Sie am besten dem folgenden Beispielprogramm, das einen String auf den Bildschirm schreibt und ihn dann kopiert, und zwar einmal im Modus 3 (Replace) und einmal im Modus 7 (Transparent):

Omikron-BASIC

```
'
' Demo-Programm für Copy Raster, Opaque (vro_cpyfm)
' Omikron-BASIC      MP 19-11-88      COPYRAST.BAS
'
Appl_Init'   Die übliche Anmeldung...
V_Opnvwk
'
V_Gtext(20,50,"Hallo, das ist der Text")'  "Original"-Text180 'Null_Long=0'
ein Langwort 0 (4-Byte-Variable)
Nullptr= VARPTR(Null_Long)'          Zeiger auf diese
Null'Vro_Cpyfm(Nullptr,Nullptr,3,20,36,204,52,20,65,204,81)230
' (3 ist Modus-Wort => Replace)
'
Vro_Cpyfm(Nullptr,Nullptr,7,20,36,204,52,20,90,204,106)
' (Modus 7 => Transparent)
'
GEMDOS (,7)'    Warten auf Taste
'
V_Clsvwk'       Ende
Appl_Exit
END
```

C

```
/*************************************************************/
/*   Demo-Programm für Copy Raster, Opaque (vro_cpyfm)   */
/*   Megamax Laser C      MP  19-11-88      COPYRAST.C   */
/*************************************************************/

#include <osbind.h>
```

```
#include "gem_inex.c"

int  pxyarray[8];
long null_long = 0L;

main()
{
  gem_init();

  v_gtext (handle, 20, 50, "Hallo, das ist der Text");

  /* Vor dem Aufruf der Rasterfunktion müssen die Koordinaten */
  /* von Quell- und Zielraster im pxyarray abgelegt werden    */

  pxyarray[0] = 20;    pxyarray[1] = 36;   /* Quellraster */
  pxyarray[2] = 204;   pxyarray[3] = 52;

  pxyarray[4] = 20;    pxyarray[5] = 65;   /* Zielraster */
  pxyarray[6] = 204;   pxyarray[7] = 81;

  vro_cpyfm (handle, 3, pxyarray, &null_long, &null_long);

  /* '&variable' liefert Zeiger auf 'variable' */
  /* Modus 3 ist Replace-Modus */

  pxyarray[5] = 90;   /* Beim zweiten kopieren ändern sich nur */
  pxyarray[7] = 106;  /* die y-Koordinaten des Zielrasters     */

  vro_cpyfm (handle, 7, pxyarray, &null_long, &null_long);

  /* Jetzt wurde Modus 7 benutzt (Transparent) */

  Crawcin();   /* Auf Taste warten */
  gem_exit();
}
```

Assembler

```
          ;
          ; Demo-Programm für Copy Raster, Opaque
          ; Assembler   MP 19-11-88    COPYRAST.Q
          ;

          .INCLUDE 'GEM_INEX.Q'

gemdos    = 1
crawcin   = 7

          .TEXT
```

```
main:   jsr     gem_init

        ; Text ausgeben

        lea.l   ausgabe,a0
        bsr     fix_text        ;String ins intin-Array bringen

        move.w  #8,contrl       ;Text (v_gtext)
        move.w  #1,contrl+2     ;Anzahl der Koordinaten in ptsin
        clr.w   contrl+4        ;Anzahl der Punkte in ptsout
        clr.w   contrl+8        ;Zahl der Werte in intout
        move.w  handle,contrl+12 ;VDI-Grafik-Handle

        move.w  #20,ptsin       ;Koordinaten
        move.w  #50,ptsin+2

        jsr     vdi

        ; Erster Copy Raster-Aufruf (Modus 3 => Replace)

        move.w  #109,contrl     ;Funktions-Opcode
        move.w  #4,contrl+2     ;4 Koordinaten in ptsin
        clr.w   contrl+4
        move.w  #1,contrl+6     ;ein Wert in intin
        clr.w   contrl+8
        move.w  handle,contrl+12

        move.l  #null,contrl+14 ;Zeiger auf Null (MFDB)
        move.l  #null,contrl+18 ;s.o., aber für Ziel

        move.w  #20,ptsin       ;Koordinaten des Quellrasters
        move.w  #36,ptsin+2
        move.w  #204,ptsin+4
        move.w  #52,ptsin+6

        move.w  #20,ptsin+8     ;Koordinaten des Zielrasters
        move.w  #65,ptsin+10
        move.w  #204,ptsin+12
        move.w  #81,ptsin+14

        move.w  #3,intin        ;Modus 3: Replace

        jsr     vdi

        ; Für den zweiten Aufruf müssen nur noch die
        ; Koordinaten und der Modus geändert werden

        move.w  #90,ptsin+10
        move.w  #106,ptsin+14
```

```
        move.w    #7,intin

        jsr       vdi

        move.w    #crawcin,-(sp)   ;Warten auf Taste
        trap      #gemdos
        addq.l    #2,sp

        jsr       gem_exit

        clr.w     -(sp)
        trap      #gemdos


fix_text:
        ; Unterprogramm, daß einen String (Startadresse in a0
        ; zu übergeben) in das intin-Array schreibt, die Länge
        ; bestimmt und in contrl[3] ablegt

        clr.w     d0              ;Länge
        clr.w     d1              ;Hilfsregister
        lea.l     intin,a1

fix_loop: move.b  (a0)+,d1        ;ein Byte aus Zielstring holen
        tst.b     d1              ;Stringende?
        beq.s     fix_end
        move.w    d1,(a1)+        ;nein, dann als Wort ins
        addq.w    #1,d0           ;intin-Arrary schreiben
        bra.s     fix_loop

fix_end:  move.w  d0,contrl+6     ;Länge festhalten
        rts


        .DATA

null:   .DC.l 0   ;Das wird unser MFDB (0 für Bildschirm)

ausgabe: .DC.b 'Hallo, das ist der Text',0

        .END
```

5.6 Ereignisse

Mit der Ereignis-Verwaltung sind wir bei einem ganz wichtigen Thema angelangt. Ein Ereignis ist all das, worauf ein Programm warten kann, also z.B. das Drücken einer Taste, ein Mausklick oder auch das Verstreichen einer bestimmten Zeitspanne.

Wenn eine Applikation auf ein bestimmtes Ereignis warten muß, dann könnte sie in einer Warteschleife verharren, bis dieses Ereignis eingetreten ist. Gibt es verschiedene Ereignisse, die, wie es in GEM-Applikation üblich ist, jederzeit eintreten können (also z.B. Maus und Tastatur), so wird die Sache etwas komplizierter, wir müßten ja in der Warteschleife gleich mehrere Geräte abfragen. Aber es wäre immerhin möglich, dies in einer Schleife zu erledigen.

Auch bei der Entwicklung des GEM hat man sich darüber Gedanken gemacht. Das Ergebnis dieser Gedanken ist eine höchst komfortable Ereignis-Verwaltung, die für den Programmierer sehr einfach zu benutzen ist, wenn er sie erst einmal verstanden hat.

Sie benötigen zunächst einige Grundkenntnisse. Als erstes wäre zu erwähnen, daß GEM zwischen zwei Zuständen eines Programms unterscheidet: bereit zu laufen und nicht bereit zu laufen. Im Englischen wird das kürzer mir ready status und not-ready status bezeichnet. Was bedeutet nun dieser Status, das heißt: wann ist ein Programm bereit zu laufen? Nun, es ist immer dann bereit zu laufen, wenn es nicht auf irgendein Ereignis wartet, wenn das Programm also alle Daten für eine Berechnung oder für eine Ausgabe hat und das tun kann, wofür Computer geschaffen wurden: arbeiten, nicht warten. Damit ist klar, was not-ready Status bedeutet: Ist ein Programm in diesem Status, so wartet es, kann also nicht arbeiten.

Jetzt kommt ein zweites Programm ins Spiel, das heißt: in den gleichen Computer. Nehmen wir an, wir hätten einen Drucker-Spooler als Accessory geladen, der einen Text ausgeben möchte. Gleichzeitig läuft die Textverarbeitung als normale Applikation. Nehmen wir ferner an, daß Sie dem Spooler aufgetragen haben, einen Text zum Drucker zu schicken. Wir haben also ein System mit zwei eigenständigen Programmen, die beide arbeiten wollen. Solange Sie jetzt keine Taste drücken, hat die Textverarbeitung natürlich nichts zu tun, sie hat not-ready Status, wartet also. Der Spooler hingegen wartet nicht (allenfalls auf den Drucker, aber das spielt in diesem Fall keine Rolle). Das erkennt GEM automatisch und läßt den Spooler ungestört arbeiten. Erst, wenn der Benutzer irgendetwas tut, was die Textverarbeitung betrifft, diese also ready Status erhält, gibt es ein kleines Problem, nämlich: Welches der beiden Programme, die ja beide bereit sind zu laufen, darf denn jetzt arbeiten?

Eine mögliche Lösung wäre zum Beispiel, jedem Programm eine bestimmte Zeit zuzuteilen, das heißt der Spooler würde eine Zehntelsekunde arbeiten, dann die Textverarbeitung (genauso lange), dann wieder der Spooler usw. In GEM geht es jedoch anders: Jedesmal, wenn eines

der Programme einen AES-Aufruf (also nicht VDI-Aufrufe) tätigt, schaltet GEM das gerade laufende, also das aufrufende Programm ab und gibt die Kontrolle an das andere. Beim nächsten AES-Aufruf dieses anderen Programms passiert das gleiche: Das jetzt aufrufende, also das zweite Programm wird stillgelegt und das erste wieder aktiviert. Der Teil des GEM, der dieses Umschalten besorgt, heißt übrigens Dispatcher.

Fassen wir zusammen: Ein Programm, das gerne arbeiten möchte, hat ready Status. Gibt es mehrere Programme/Acessories mit ready Status, so schaltet der Dispatcher des GEM bei jedem AES-Aufruf zwischen den Programmen um. Wartet aber ein Programm auf ein Ereignis, und benutzt es dazu die Ereignis-Funktionen des AES, dann bekommt dieses Programm not-ready Status, d.h. es wird solange vom Dispatcher ignoriert (also nicht mehr aktiviert), bis das geforderte Ereignis eingetreten ist.

Stellen Sie bitte sicher, daß Sie bis hierhin alles nachvollziehen konnten. Ich weiß, daß dies ein schwieriges Kapitel ist, aber ich verspreche Ihnen: Es lohnt sich, allein schon deshalb, weil kein GEM-Programm ohne die Ereignis-Verwaltung leben kann.

Eben war von Ereignis-Funktionen die Rede. Ereignis-Funktionen sind Funktionen des AES. Sie werden von einem Programm immer dann aufgerufen, wenn auf ein Ereignis gewartet werden soll. Für unterschiedliche Ereignis-Typen gibt es verschiedene Funktionen. Aber welche Ereignis-Typen gibt es eigentlich? Zählen wir sie einmal der Reihe nach auf:

- Tastendruck (evnt_keybd),
- Mausklick (evnt_button),
- Mausüberwachung. Hiermit kann darauf gewartet werden, daß der Mauszeiger ein von Ihnen vorgegebenes Rechteck betritt oder verläßt (evnt_mouse).
- Nachricht. Mit den Nachrichten (weder heute noch Tagesschau) werden wir uns noch ausführlich beschäftigen. Hier sei nur erwähnt, daß ein Programm Mitteilungen vom GEM oder von anderen GEM-Programmen erhalten kann (evnt_mesag).
- Zeit. Das Programm soll warten, bis eine bestimmte Zeit verstrichen ist (evnt_timer).

In Klammern habe ich Ihnen schon gleich die Namen der AES-Funktionen gegeben, die Sie aufrufen müssen, wenn Sie auf eines dieser Ereignisse warten wollen. Wie üblich bitte ich Sie nun darum, einen kurzen Blick in Ihre GEM-Dokumentation zu werfen, damit Sie wieder einen groben Überblick über die oben genannten Funktionen bekommen. Dabei sollten Sie vielleicht die Funktion evnt_mesag erst einmal auslassen.

Falls Sie den Zustand völliger Verwirrung genießen sollten, dann empfehle ich Ihnen die Lektüre der Parameter, die an die Funktion evnt_multi übergeben werden; es sind mehr als 20! Allerdings ist diese große Zahl recht einfach zu erklären: evnt_multi ist die Ereignisfunktion, die immer dann benutzt wird, wenn das Programm auf verschiedene Ereignisse reagieren muß, und deshalb benötigt diese Funktion natürlich die Parameter aller anderen Ereignisfunktionen. In der Tat ist es so, daß eine GEM-Applikation fast immer mehrere Ereignis-Typen verarbeiten muß; denn ein Programm muß ja gleichermaßen auf Tastendrücke, Mausklicks und -bewegungen und auch, wie wir noch sehen werden, auf Nachrichten reagieren können. Diese Funktion evnt_multi ist daher die in GEM-Applikationen meistbenutzte Ereignis-Funktion.

So, jetzt können wir uns an ein erstes Beispielprogramm machen. Ich habe eine kleine Applikation geschrieben, die auf einen Mausklick wartet und an der Stelle, wo sich der Mauszeiger zur Zeit des Klicks befindet, ein kleines x schreibt. Wenn der Klick allerdings ein Doppelklick war, dann soll statt des kleinen ein großes X ausgegeben werden. Wir benötigen dazu die Funktion evnt_button, die automatisch zwischen Klick und Doppelklick unterscheiden kann. Das Programm soll dann beendet sein, wenn zusätzlich zur Maustaste auch eine Shift-Taste gedrückt wurde.

Betrachten wir zunächst die Parameter, die evnt_button verlangt: Als erstes müssen Sie angeben, ob nur auf normale Klicks (1), Doppelklicks (2) oder auch n-fach-Klicks (n) gewartet werden soll, wobei Werte über 2 den Benutzer wohl überfordern. Für den alltäglichen Gebrauch kommen also nur die Werte 1 und 2 in Frage, wobei Sie 2 nur dann angeben sollten, wenn Sie auch wirklich auf Doppelklicks reagieren möchten.

Ferner benötigt die Funktion zwei Werte, mit denen Sie angeben, worauf ganz genau gewartet werden soll. In der GEM-Dokumentation heißen dieser Werte ev_bmask und ev_bstate. In beiden sind nur die Bits 0 und 1 relevant; Bit 0 bezieht sich auf die linke Taste, Bit 1 auf die rechte. Nun, mit diesen Bits können Sie in ev_bmask bestimmen, welche Maustaste für das Ereignis überprüft werden soll. Ein gesetztes Bit heißt, daß die entsprechende Taste zu überprüfen ist. In ev_bstate müssen Sie hingegen für jede abzufragende Taste angeben, ob das Ereignis dann

eintreten soll, wenn die Taste gedrückt ist (Bit gesetzt) oder wenn sie nicht gedrückt ist (Bit gelöscht). Sie sehen, die Funktion kann also auch darauf warten, daß ein Mausknopf wieder losgelassen wird.

Eines kann die Funktion aber nicht: darauf warten, daß entweder die linke oder die rechte Maustaste betätigt wird. Wenn Sie sowohl ev_bmask als auch ev_bstate auf drei setzen, also alle relevanten Bits setzen, dann gilt das Ereignis erst dann als erfüllt, wenn beide Tasten gleichzeitig gedrückt sind.

Die Funktion gibt auch Werte zurück (in GFA-BASIC optional). Dazu gehören die x- und y-Koordinaten des Mauszeigers zu der Zeit, als das Ereignis ausgelöst wurde, ferner der Zustand beider Maustasten (Bit 0 und 1 wie gehabt; Bit gesetzt = Maustaste gedrückt) nach (!) Ablauf der Funktion (damit kann man z.B. feststellen, ob es ein kurzer Klick war oder ein langer, z.B. um Objekte zu bewegen) und ein Wert, der den Zustand der Sondertasten (Shift, Control und Alternate) darstellt. Die genaue Zuordnung der Bits zu diesen Tasten entnehmen Sie bitte Ihrer GEM-Dokumentation.

Für eine so simple Angelegenheit wie die Abfrage eines kleinen Mausknopfes ist also eine ganze Menge an Verwaltungsaufwand erforderlich. Schauen Sie sich das Beispielprogramm an, und sehen Sie auch in Ihren anderen Büchern noch einmal unter evnt_button nach.

GFA-BASIC

```
'
' evnt_button - Demoprogramm (Ereignis-Verwaltung)
' GFA-BASIC        MP  22-11-88        BUTTON.GFA
'
VOID APPL_INIT()   ! Anmelden bei AES
'
PRINT "Ende mit Shift und Klick..."
'
GRAPHMODE 2      ! Transparent-Modus
'
DO
  klicks%=EVNT_BUTTON(2,1,1,x%,y%,k%,switch%)
  '
  ' Parameter: 2 -> maximal Doppelklick (kein 'Dreifachklick')
  '            1 -> linke Maustaste soll abgefragt werden
  '            1 -> Ereignis: die Taste soll gedrückt sein
  '            x%, y%  -> Mauskoordinaten nach dem Klick
  '            k%      -> Maustastenstatus beim Klick
  '            switch% -> Sondertastenstatus während des Klicks  '
```

```
  EXIT IF switch%>0   ! Programm beenden, wenn Shift o.Ä. gedrückt ist
  '
  IF klicks%=1      ! einfacher Klick
    TEXT x%,y%,"x"
  ELSE
    TEXT x%,y%,"X"
  ENDIF
LOOP
'
VOID APPL_EXIT()
'
END
```

Omikron-BASIC

```
'
' evnt_button - Demoprogramm (Ereignis-Verwaltung)
' Omikron-BASIC       MP  23-11-88     BUTTON.BAS
'
Appl_Init
V_Opnvwk
'
CLS
Vswr_Mode(2)'    Transparent Write-Mode
PRINT "Ende mit Shift und Klick..."
'
WHILE 1'   Omikron hat keine DO ... LOOP - Schleife
  Evnt_Button(2,1,1,Klicks,X,Y,K,Switch)
  ' Die Parameter wurden im GFA-Listing erläutert. Unterschied:
  ' Klicks wird als VAR-Parameter übergeben (GFA: Funktionswert)
  '
  IF Switch>0 THEN EXIT '  Ende, wenn zusätzlich zum Klick eine
  '                        der Sondertasten gedrückt wurde
  IF Klicks=1'  einfacher oder doppelter Klick?
    THEN V_Gtext(X,Y,"x")
    ELSE V_Gtext(X,Y,"X")
  ENDIF
WEND
'
V_Clsvwk
Appl_Exit
END
```

C

```
/*****************************************************/
/*   evnt_button - Demoprogramm (Ereignis-Verwaltung)  */
/*   Megamax Laser C      MP  23-11-88      BUTTON.C   */
/*****************************************************/
```

```
#include <osbind.h>
#include "gem_inex.c"

int  x, y, k,
     klicks,
     swtch;  /* abweichender Name, weil 'switch' reserviert ist */main()
{
  gem_init();
  graf_mouse (0, 0L);   /* Pfeil statt Biene als Mauszeiger */
/* (Biene ist ja nach dem Laden noch da) */
                        /* 0L ist zunächst ohne Bedeutung für Sie */

  Cconws ("\33E");  /* Bildschirm löschen */
  vswr_mode (handle,2);    /* Transparent-Modus */

  do
  {
    klicks = evnt_button (2, 1, 1, &x, &y, &k, &swtch);

    if (klicks == 1)   /* wieviele Klicks? */
      v_gtext (handle, x, y, "x");
    else
      v_gtext (handle, x, y, "X");
  }
  while (swtch == 0);

  gem_exit();
}
```

Assembler

```
          ;
          ; evnt_button - Demoprogramm (Ereignis-Verwaltung)
          ; Assembler         MP  23-11-88         BUTTON.Q
          ;

          .INCLUDE 'GEM_INEX.Q'

gemdos    = 1cconws    = 9          .TEXT

main:     jsr       gem_init

          pea       clrscr           ;Bildschirm löschen
          move.w    #cconws,-(sp)
          trap      #gemdos
          addq.l    #6,sp

          ; Transparent-Modus einschalten

          move.w    #32,contrl       ;Opcode
```

```
        clr.w     contrl+2
        clr.w     contrl+4
        move.w    #1,contrl+6
        move.w    #1,contrl+8
        move.w    handle,contrl+12

        move.w    #2,intin          ;2 = Transparent

        jsr       vdi

        ; Nach dem Laden ist die Biene als Mauszeiger aktiv.
        ; Das behindert unser Programm jedoch. Deshalb
        ; schalten wir statt dessen auf den normalen Pfeil um.
        ; Das macht die AES-Funktion graf_mouse.

        move.w    #78,control
        move.w    #1,control+2
        move.w    #1,control+4
        move.w    #1,control+6
        clr.w     control+8

        clr.w     int_in            ;0 für Pfeil
        clr.l     addr_in           ;für uns: Dummy-Null

        jsr       aes

loop:
        ; evnt_button - Aufruf

        move.w    #21,control       ;AES, daher contro(!)l
        move.w    #3,control+2
        move.w    #5,control+4
        clr.w     control+6
        clr.w     control+8

        move.w    #2,int_in         ;maximal Doppelklick registrieren
        move.w    #1,int_in+2       ;linken Knopf abfragen
        move.w    #1,int_in+4       ;Ereignis: Mausknopf gedrückt

        jsr       aes

        tst.w     int_out+8         ;Status der Sondertasten
        bne       ende              ;Ende  falls irgendwas gedrückt

        cmpi.w    #1,int_out        ;wie viele Klicks?
        bne.s     l
        move.b    #'x',d0           ;einer, dann kleines x vormerken
        bra.s     ll
l:      move.b    #'X',d0           ;sonst Doppelklick und großes X
```

```
ll:

        ; v_gtext aufrufen

        move.w    #8,contrl
        move.w    #1,contrl+2
        clr.w     contrl+4
        clr.w     contrl+8
        move.w    handle,contrl+12

        move.w    int_out+2,ptsin  ;Koordinaten von evnt_button
        move.w    int_out+4,ptsin+2

        move.b    d0,ausgabe       ;Zeichen in String umwandeln
        lea.l     ausgabe,a0
        bsr       fix_text         ;String in intin-Array schreiben

        jsr       vdi

        bra       loop

ende:   jsr       gem_exit

        clr.w     -(sp)
        trap      #gemdos


fix_text:
        ; Unterprogramm, das einen String (Startadresse in a0
        ; zu übergeben) in das intin-Array schreibt, die Länge
        ; bestimmt und in contrl[3] ablegt

        clr.w     d0               ;Länge
        clr.w     d1               ;Hilfsregister
        lea.l     intin,a1

fix_loop: move.b  (a0)+,d1         ;ein Byte aus Zielstring holen
        tst.b     d1               ;Stringende?
        beq.s     fix_end
        move.w    d1,(a1)+         ;nein, dann als Wort ins
        addq.w    #1,d0            ;intin-Arrary schreiben
        bra.s     fix_loop

fix_end: move.w   d0,contrl+6      ;Länge festhalten
        rts

        .DATA
```

```
ausgabe:  .DC.w 0

clrscr:   .DC.b 27,'E',0

          .END
```

Ein anderer Ereignistyp wurde oben schon einmal erwähnt: eine Nachricht. Nachricht heißt auf Englisch message, und deshalb können Sie mit der AES-Funktion evnt_mesag auf eine solche Nachricht warten. Nun interessiert Sie natürlich, was das für Nachrichten sind, auf die ein Programm warten kann. Und hier stehe ich vor einem großen Problem: Die meisten der möglichen Nachrichten gehören zu Dingen, mit denen wir uns erst später beschäftigen werden. Andererseits kann ich die Besprechung dieser Dinge auch nicht vorziehen, weil dazu Kenntnisse über das Nachrichten-System des GEM nötig sind. Deshalb erzähle ich Ihnen erstmal das nötigste; Beispiele bekommen Sie dann später bei den gewissen anderen Dingen.

Nachrichten kann ein GEM-Programm von zwei verschiedenen Quellen erhalten: von anderen GEM-Programmen und vom GEM selber. Eine Nachricht ist dabei ganz konkret ein Speicherbereich von 8 Worten, also 16 Bytes. Wenn ein Programm auf ein Ereignis wartet, so muß es immer eine Adresse angeben, ab der diese Nachricht abgelegt werden kann. Wenn das Ereignis eingetreten ist, dann befindet sich im ersten Wort (Wort 0) immer eine Zahl, die die Art der Nachricht beschreibt. Auf die möglichen Werte und ihre Bedeutungen gehe ich aber noch nicht ein.

Betrachten wir noch einmal die beiden verschiedenen Quellen, von denen eine Nachricht ausgehen kann. Die erste Möglichkeit, eine Nachricht von einem anderen GEM-Programm, ist sehr selten; denn die Programme, die Nachrichten untereinander austauschen (etwa zwei Accessories untereinander, aber auch ein Accessory und eine Applikation könnten das tun), müssen aufeinander abgestimmt sein. Im Klartext heißt das, daß Sie das Empfänger-Programm mit einer Reihe von definierten Kommandos ausstatten müssen, die es verstehen und verarbeiten kann. Das Kommando und eventuelle zusätzliche Werte werden dann in den sogenannten Ereignis-Puffer (das sind die oben erwähnten 8 Worte) des Ziel-Programms geschrieben; zu diesem Zweck gibt es eine spezielle AES-Funktion. Es gibt für dieses Vorgehen durchaus sinnvolle Anwendungsmöglichkeiten, doch ist ein derartiger Informationsaustausch im Atari nur gelegentlich zu finden.

Viel wichtiger ist für uns die andere Nachrichtenquelle: GEM selber. Der Teil, der uns die Nachrichten schickt, heißt Screen-Manager. Wie der

Name vermuten läßt, hat er etwas mit dem Bildschirm zu tun. Und tatsächlich kontrolliert dieser Screen-Manager all das, was der Benutzer mit der Maus auf dem Bildschirm eines normalen GEM-Programms machen kann. Und hier sind wir bei einem ganz wichtigen Punkt angelangt, denn wir müssen unterscheiden, wann ein Mausklick als reiner Klick über die Funktion evnt_button, die wir ja schon kennen, gemeldet wird, und wann als Nachricht über die Funktion evnt_mesag. Der Unterschied mag Ihnen im Moment bedeutungslos erscheinen, doch Sie werden mir recht geben, daß ein Klick auf einem Menüpunkt eines Pull-Down-Menüs etwas anderes ist als ein Klick auf irgendeine Stelle im Arbeitsbereich des Fensters. Ja, es gibt sogar Extremfälle: Denken Sie wieder einmal an eine Textverarbeitung. Sie klicken nun einen Menüpunkt ganz unten in einem ellenlangen Pull-Down-Menü an. Würde die Textverarbeitung nur die Bestätigung erhalten, daß da an einer bestimmten Koordinate ein Klick war, dann könnte sie nicht wissen, ob sich an dieser Koordinate zur Zeit des Klicks ein Pull-Down-Menü befunden hat; schließlich könnte es ja auch sein, daß der Anwender nur den Cursor an eine bestimmte Stelle bewegen wollte und dazu an der gleichen Koordinate auf einen Buchstaben geklickt hat.

Damit soll folgendes verdeutlicht werden: Alle Mausaktivitäten des Benutzers werden in zwei Gruppen eingeteilt: Alles, was innerhalb (!) von Fenstern (also nicht am Fensterrahmen) oder auf dem Desktop (also nicht in der Menüleiste) angeklickt wird, verursacht eine Benachrichtigung des Programms über die Funktion evnt_button (natürlich muß das Programm die Funktion dazu vorher aufgerufen haben). Alles andere, das heißt Klicks im Fensterrahmen (Fenster vergrößern, scrollen, schließen) und Klicks auf Menüpunkte von Pull-Down-Menüs sind Nachrichten, die nur über die Funktion evnt_mesag empfangen werden können. Am Rande sei bemerkt, daß natürlich beide Arten von Klicks auch mit evnt_multi abgefangen werden können.

So, wenn Sie Lust haben, dann schauen Sie sich doch einfach mal die möglichen Ereignisse in einem Ihrer anderen Bücher an. Ich werde diese erst nach und nach vorstellen, wenn wir sie gerade brauchen. Bitte haben Sie Verständnis dafür, daß ich dieses Thema nur häppchenweise vorstelle, aber alle Informationen auf einen Schlag zu bringen, wäre nicht sinnvoll.

Jetzt aber noch ein kleines Beispiel für die Verwendung von evnt_multi, damit Sie auch diese Funktion in Zukunft benutzen können. Es handelt sich um ein Programm, das Sie mit einem Tastendruck beenden können. Tun Sie das nicht, dann wird es nach drei Sekunden automatisch beendet.

Wir benötigen also die Kombination der Funktionen evnt_keybd und evnt_timer. Um evnt_multi das mitzuteilen, wird dieser Funktion ein Wert übergeben, in dem es für jedes mögliche Ereignis ein Bit gibt. Ein gesetztes Bit sagt, daß das entsprechende Ereignis eines von denen ist, auf die wir warten wollen. Beachten Sie übrigens, daß die ursprüngliche Funktion evnt_mouse (sie überprüft das Eintreten oder Verlassen des Mauszeigers in einen bzw. aus einem bestimmten Bildschirmbereich) doppelt vorhanden ist; Sie können also gleichzeitig zwei Bildschirmbereiche aktiv haben. Aber das nur am Rande.

In unserem Fall sind die Bits 0 (für evnt_keybd) und 5 (für evnt_timer) zu setzen, was einen dezimalen Wert von 33 ergibt. Die Funktion wird unter anderem einen Wert zurückgeben, der uns anzeigt, welches der möglichen Ereignisse denn nun wirklich eingetreten ist und den Abbruch der Funktion (und damit die Wiederaufnahme des Programms) bewirkt hat. Die Bits haben die gleiche Bedeutung wie oben. Die restlichen Parameter sind mit denen der einzelnen Ereignisfunktionen identisch, was nicht weiter verwunderlich ist, da ja die gleichen Funktionen übernommen werden. Informieren Sie sich also dort über die Bedeutung der Parameter. Übrigens: Auch, wenn Sie nicht alle Parameter benötigen, so müssen Sie dennoch beim Aufruf alle angeben. Allerdings ist es üblich, für nicht benutzte Rückgabevariablen den Namen dummy (etwa: bedeutungsloser Ersatz) zu wählen.

Das Programm sieht folgendermaßen aus:

GFA-BASIC

```
'
' Meherere Ereignisse gleichzeitig abfragen
' GFA-BASIC       MP 24-11-88       MULTI.GFA
'
VOID APPL_INIT()
'
PRINT "Drücken Sie eine Taste. Wenn Sie nicht innerhalb von"
PRINT "3 Sekunden eine Taste gedrückt haben, ist das Programm"PRINT "beendet."
'
which%=EVNT_MULTI(33,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3000,dummy%,dummy%,...
     ...dummy%,dummy%,taste%,dummy%)
'
PRINT
IF which%=1  ! which% ist 1 (Bit 0) oder 32 (Bit 5)
  PRINT "Abbruch mit Taste"
ELSE
  PRINT "Abbruch nach 3 Sekunden"
ENDIF
```

```
'
VOID APPL_EXIT()
END
```

Omikron-BASIC

```
'
' Mehrere Ereignisse gleichzeitig abfragen
' Omikron-BASIC   MP  25-11-88   MULTI.BAS
'
Appl_Init
V_Opnvwk
'
CLS
PRINT "Drücken Sie eine Taste. Wenn Sie nicht innerhalb von"
PRINT "3 Sekunden eine Taste gedrückt haben, ist das Programm"PRINT "beendet."
'
Evnt_Multi(33,0,0,0,0,0,0,0,0,0,0,0,0,0,3000,A$,Which,Dummy,...
      ...Dummy,Dummy,Dummy,Taste,Dummy)
'
PRINT
IF Which=1
  THEN PRINT "Abbruch mit Taste"
  ELSE PRINT "Abbruch nach 3 Sekunden"
ENDIF
'
GEMDOS (,7)' Warten auf Taste...
V_Clsvwk
Appl_Exit
END
```

Die Evnt-Multi-Zeile in BASIC (GFA und Omikron) mußte aus drucktechnischen Gründen in zwei Zeilen abgedruckt werden. Lassen Sie bitte die Punkte weg und geben Sie alles in einer Zeile ein.

C

```
/************************************************/
/*   Mehrere Ereignisse gleichzeitig abfragen   */
/*   Laser C        MP  25-11-88      MULTI.C   */
/************************************************/

#include <osbind.h>
#include "gem_inex.c"

int  which,
     taste, dummy;

main()
```

```
{
  gem_init();

  v_gtext (handle, 0, 14,
     "Drücken Sie eine Taste. Wenn Sie nicht innerhalb von");
  v_gtext (handle, 0, 30,
     "3 Sekunden eine Taste gedrückt haben, ist das Programm");
  v_gtext (handle, 0, 46, "beendet.");

  which = evnt_multi (33, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                      0L, 3000, 0, &dummy, &dummy, &dummy,
                      &dummy, &taste, &dummy);

  if (which == 1)
    v_gtext (handle, 0, 80, "Abbruch mit Taste");
  else
    v_gtext (handle, 0, 80, "Abbruch nach 3 Sekunden");

  Crawcin();   /* Warten auf Taste */
  gem_exit();
}
```

Assembler

```
          ;
          ; Mehrere Ereignisse gleichzeitig abfragen
          ; Assembler      MP  25-11-88      MULTI.Q
          ;

gemdos    = 1
crawcin   = 7
cconws    = 9

          .INCLUDE 'GEM_INEX.Q'

          .TEXT

main:     jsr       gem_init

          pea       meldung          ;Hinweis ausgeben
          move.w    #cconws,-(sp)
          trap      #gemdos
          addq.l    #6,sp

          ; evnt_multi aufrufen

          move.w    #25,control      ;Funktions-Opcode
          move.w    #16,control+2
          move.w    #7,control+4
```

```
          clr.w     control+8

          move.w    #33,int_in       ;Ereigniskombination (Bit 0 und 5)
          move.w    #3000,int_in+28  ;low-Wert für Timer
          clr.w     int_in+30        ;high-Wert (in Millisekunden)

          jsr       aes

          cmpi.w    #1,int_out       ;Welches Ereignis ist denn nun
          bne.s     time             ;wirklich eingetreten?
          pea       a_taste
          bra.s     l
time:     pea       a_time

l:        move.w    #cconws,-(sp)
          trap      #gemdos
          addq.l    #6,sp

          move.w    #crawcin,-(sp)   ;Warten auf Taste...
          trap      #gemdos
          addq.l    #2,sp

          jsr       gem_exit

          clr.w     -(sp)
          trap      #gemdos

          .DATA

meldung:  .DC.b 27,'EDrücken Sie irgendeine Taste. Wenn Sie nicht '
          .DC.b 'innerhalb von',13,10,'3 Sekunden eine Taste gedrückt '
          .DC.b 'haben, ist das Programm',13,10,'beendet.',13,10,10,0

a_taste:  .DC.b 'Abbruch mit Taste',13,10,0

a_time:   .DC.b 'Abbruch nach 3 Sekunden',13,10,0

          .END
```

5.7 Die File-Selector-Box

Nun kommen wir endlich zu einer AES-Funktion, die nicht nur in fast jedes GEM-Programm gehört, sondern die auch nicht mehr ausschließlich zu Demonstrationszwecken gezeigt wird. Diese Funktion können Sie in jedem Ihrer Programme verwenden und dabei einiges an Arbeit sparen.Die

Ich rede von dem Datei-Auswahl-Fenster, vielleicht besser bekannt als File-Selector-Box. Das ist das Fenster, das immer dann erscheint, wenn ein Programm Dateien laden oder sichern muß und dazu einen Dateinamen von Ihnen erfragt. Sie können sich vorstellen, daß dieses ganze Theater mit Laufwerken, Ordnern und Dateinamen recht umständlich abzuwickeln ist, und gerade deshalb stellt die File-Selector-Funktion des GEM eine so große Erleichterung dar: Alles, was Sie als kompliziertes, standardisiertes Fenster auf dem Bildschirm sehen, kostet den Programmierer unter GEM nicht mehr als einen einzigen Funktionsaufruf.

Nun, so ganz richtig ist das nicht, denn mit dem Funktionsaufruf alleine ist es noch nicht getan. Bevor wir sie aufrufen können, müssen wir einige Parameter zusammenbasteln. Die Funktion (sie heißt übrigens fsel_input) möchte nämlich vom Programmierer wissen, auf welchem Pfad (das heißt: in welchem Ordner auf welchem Laufwerk) sich die anzugebende Datei voraussichtlich befindet. Wenn Sie sich eine File-Selector-Box einmal ansehen, so werden Sie diesen Pfadnamen dort finden. Sie können ihn auch verändern, was besonders dann sinnvoll ist, wenn Sie eine Datei auf einem anderen Laufwerk als dem voreingestellten angeben möchten (zumindest bis TOS 1.2, in der Version 1.4 gibt es ja dafür endlich spezielle Knöpfe). Kurz und gut: Diesen voreingestellten Pfadnamen muß das GEM-Programm selbst erstellen. Der zweite Parameter, den die Funktion benötigt, ist dann nur noch ein String, der optional einen voreingestellten Dateinamen enthält; dieser erscheint dann automatisch unter Auswahl neben dem Fenster mit den anderen Dateinamen. Das ist immer dann sehr praktisch, wenn das Programm eine Datei geladen und bearbeitet hat und sie nun wieder abspeichern möchte. Dann kann der gleiche Dateiname, der auch beim Laden angegeben wurde, als Standard-Name (man nennt das auch Default-Name) angegeben werden. Der Benutzer muß so nur noch Return drücken oder auf OK klicken.

Nach Abschluß der Funktion gibt es noch etwas zu tun: Sie bekommen vom AES nur den eventuell geänderten Pfadnamen und den ausgewählten Dateinamen zurück, und zwar in zwei verschiedenen Strings. Sie müssen diese zu einem einzigen verschmelzen, so daß Sie ihn als Dateinamen benutzen können, der den Pfadnamen und den eigentlichen Dateinamen enthält. Ein Beispiel: Aus

```
Pfad:       A:\ORDNER\*.PRG   und
Dateiname:  BTEXT.PRG
```

muß das GEM-Programm folgendes machen:

```
Ergebnis:    A:\ORDNER\BTEXT.PRG
```

Dieser Name kann nun einer GEMDOS-Funktion wie Fopen zur Bearbeitung übergeben werden. Das Verfahren, dieses Ergebnis zu bilden, ist nicht schwer: Wir suchen im Pfad von hinten nach dem ersten (also nach dem letzten, weil wir uns vom Ende zum Anfang durchsuchen) Backslash (\). In unserem Fall wäre das das Zeichen direkt hinter dem Namen ORDNER. Alles, was nach diesem Backslash noch kommt, wird gelöscht. Es bleibt also A:\ORDNER\ übrig. Daran hängen wir den Dateinamen an - fertig!

In GFA-BASIC ist das nicht nötig, es gibt dort nämlich eine etwas komfortablere Version von fsel_input. Der Befehl heißt Fileselect. Hier entfällt die Bildung eines Gesamtnamens aus Pfad- und Dateiname; das macht GFA-BASIC automatisch. Auch in Omrikon-BASIC gibt es einen gleichlautenden Befehl, dessen Syntax und Funktion jedoch mit fsel_input (also der weniger komfortablen Version) identisch ist. Einziger Unterschied ist, daß Sie für Fileselect die GEM-Library (GEMLIB.BAS) nicht laden müssen.

Da diese Funktion sehr häufig benutzt wird, möchte ich Ihnen ein kleines Unterprogramm geben, das folgende Schritte selbständig durchführt:

- das Basteln eines Pfadnamens, wobei vom GEMDOS Daten über das aktuelle Laufwerk und den aktuellen Ordner besorgt werden,
- das Aufrufen von fsel_input bzw. Fileselect (in BASIC) und
- das Zurückführen von Pfad- und Dateinamen auf einen einzigen kombinierten Pfad- und Dateinamen.

Das GEM-Programm muß dann nur noch das Unterprogramm aufrufen und erhält als Ergebnis einen kompletten Dateinamen samt Pfad, mit dem es sofort weiterarbeiten kann.

Wir müssen drei Parameter übergeben, und zwar drei Strings (bzw. in C und Assembler deren Startadressen; siehe Beispielaufruf im Hauptprogramm). Der erste gibt die Default-Maske an, also z.B. *.* oder LISTE?.TXT. Der zweite String ist ein Default-Dateiname; das kann auch ein Leerstring sein. Die Assembler-Programmierer müssen beachten, daß dieser String jedoch mindestens 13 Bytes lang werden kann (8 Bytes Name, 1 Byte Punkt, 3 Bytes Extension und ein abschließendes Null-

byte). Der dritte String dient nur der Ausgabe: Hier wird der kombinierte Pfad- und Dateiname der Datei hinterlegt, die der Benutzer angeklickt hat.

Das Unterprogramm ist übrigens eine Funktion. Der Funktionswert ist 0, wenn der Benutzer den Knopf Abbruch angeklickt hat oder aber sein OK gegeben hat, obwohl er gar keine Datei angeklickt hat. Wenn aber ein Dateiname ordnungsgemäß ausgewählt wurde, dann ist der Funktionswert 1. Nur in Omrikon-BASIC mußte ich davon etwas abweichen; hier wird der Funktionswert als Rückgabeparameter in einer Prozedur geliefert (das war nötig, weil Omrikon-BASIC keine Funktionsaufrufe mit Rückgabeparametern erlaubt).

Die Unterprogramme sind natürlich auch eigenständig in Ihren eigenen GEM-Programmen lauffähig, wenn Sie das wünschen. Betrachten Sie einfach die Beispiel-Aufrufe der folgenden Programme; Sie werden staunen, wie einfach sich der Aufruf dieser Funktion nunmehr gestaltet!

GFA-BASIC

```
'
' Dateinamen erfragen mit der File-Selector-Box
' GFA-BASIC        MP  27-11-88        FSEL.GFA
'
IF FN filename("*.*","",dateiname$)
  PRINT dateiname$
ELSE
  PRINT "ungültige Auswahl!"
ENDIF
END
'
'
FUNCTION filename(maske$,default$,VAR ausgabe$)
  LOCAL akt_drive%,path$
  akt_drive%=GEMDOS(&H19) ! GEMDOS-Funktion bestimmt aktuelles Laufwerk
  path$=CHR$(ASC("A")+akt_drive%)+":"+DIR$(0)+"\"+maske$
  '
  FILESELECT path$,default$,ausgabe$
  '
  IF ausgabe$="" OR RIGHT$(ausgabe$,1)="\"
    RETURN 0   ! ungültige Auswahl oder Abbruch angeklickt
  ELSE
    RETURN 1   ! OK, Dateiname wurde angeklickt
  ENDIF
ENDFUNC
'
```

Omikron-BASIC

```
'
' Dateinamen erfragen mit der File-Selector-Box
' Omikron-BASIC       MP 27-11-88      FSEL.BAS
'
' Das Programm benötigt nicht die Libraray GEMLIB.BAS.
' Wir müssen allerdings selber den Textcursor ausschalten
' und den Mauszeiger sichtbar machen.
'
PRINT CHR$(27);"f";'  Cursor aus
MOUSEON
'
Filename(Ret,"*.*","",Dateiname$)
IF Ret
  THEN PRINT Dateiname$
  ELSE PRINT "ungültige Auswahl!"
ENDIF
'
END
'
'
DEF PROC Dgetpath(R Path$)'  ermittelt aktuelles Directory
  Path$= STRING$(64,0)'  Platz machen
  Adr= LPEEK( VARPTR(Path$))+ LPEEK( SEGPTR +28)
  GEMDOS (,$47, HIGH(Adr), LOW(Adr),0)' 0=aktuelles Laufwerk
  '
  ' alle Null-Bytes aus dem String abschneiden:
  '
  Path$= LEFT$(Path$, INSTR(Path$, CHR$(0))-1)
RETURN
'
DEF PROC Filename(R Back,Maske$,Default$,R Ausgabe$)
  LOCAL Akt_Drive,Path$
  GEMDOS (Akt_Drive,$19)' aktuelles Laufwerk ermitteln
  Dgetpath(Path$)' Pfad des aktuellen Laufwerks bestimmen
  Path$= CHR$( ASC("A")+Akt_Drive)+":"+Path$+"\"+Maske$
  '
  FILESELECT (Path$,Default$,Flag)
  '
  IF Flag=0 OR Default$="" THEN Back=0: RETURN
  '
  Back=1
  Ausgabe$= LEFT$(Path$, LEN(Path$)- INSTR( MIRROR$(Path$)+"\","\")) ...
       ... +"\"+Default$
RETURN
```

C

```
/*****************************************************/
/*   Dateinamen erfragen mit der File-Selector-Box   */
/*   Megamax Laser C      MP  27-11-88      FSEL.C   */
/*****************************************************/

#include <osbind.h>
#include "gem_inex.c"

char fname[80];

int filename(maske, def, ausgabe)    /* def statt default */
    char maske[], def[], ausgabe[];  /* (in C reserviert) */
{
  char path[64],
       fnam[13];
  int  back,
       button,
       i;

  strcpy (fnam, def);

  path[0] = 'A' + Dgetdrv();  /* aktuelle Laufwerksbezeichnung */
  path[1] = ':';
  Dgetpath (&path[2], 0);     /* Pfad von aktuellem Laufwerk */  strcat (path,
"\\");        /* \\ entspricht \ */
  strcat (path, maske);

  back = fsel_input (path, fnam, &button);

  if (button == 0 || fnam[0] == '\0')
    return (0);  /* Funktion ist 0 wenn Abbruch oder falsche Auswahl */

  for (i = strlen(path) - 1;   path[i] != '\\';   path[i--] = '\0');
  strcat (path, fnam);
  strcpy (ausgabe, path);

  return (1);
}


main()
{
  gem_init();
  graf_mouse (0, 0L);     /* normaler Mauszeiger (Pfeil) */

  if (filename ("*.*", "", fname))
    Cconws (fname);
  else
```

```
    Cconws ("ungültige Auswahl!");

  Crawcin();

  gem_exit();
}
```

Assembler

```
          ;
          ; Dateinamen erfragen mit der File-Selector-Box
          ; Assembler        MP  28-11-88          FSEL.Q
          ;

          .INCLUDE 'GEM_INEX.Q'

gemdos    = 1
crawcin   = 7
cconws    = 9
dgetdrv   = $19
dgetpath  = $47


          .TEXT

main:     jsr       gem_init

          pea       ausgabe          ;Funktion aufrufen
          pea       default          ;(mit Parametern wie in C, nur)
          pea       maske            ;umgekehrte Reihenfolge)
          jsr       dateiname
          adda.l    #12,sp           ;auch hier: Stackkorrektur
          tst.w     d0               ;Fehler?
          beq.s     error

          pea       ausgabe
          bra.s     go_on
```

```
          clr.w     -(sp)
          trap      #gemdos


dateiname:

          ; Dieses Unterprogramm ruft die File-Selector-Box auf.
          ; Folgende Parameter sind auf dem Stack zu übergeben:
          ;  -long:  Zeiger auf String, in den der komplette Dateiname
          ;          geschrieben wird
          ;  -long:  Zeiger auf Default-Dateiname (min. 13 Bytes)
          ;  -long:  Zeiger auf Default-Dateimaske (z.B. *.*)
          ; Alle Strings werden mit Nullbyte abgeschlossen.
          ; Funktionswert in d0: 0 -> Abbruch angeklickt oder Fehler
          ;                      1 -> Alles klar

          link      a6,#0           ;a6 als Basis für Parameter
          movem.l   d1-d7/a0-a5,-(sp)
          movea.l   16(a6),a2       ;Ziel für Ausgabe (1. Parameter)

          ; Aktuelles Lauswerk erfragen

          move.w    #dgetdrv,-(sp)
          trap      #gemdos
          addq.l    #2,sp
          addi.b    #'A',d0         ;Funktionsergebnis -> Buchstabe
          move.b    d0,(a2)
          move.b    #':',1(a2)      ;Doppelpunkt hinter Laufwerk

          ; Pfad des aktuellen Laufwerks erfragen

          clr.w     -(sp)           ;aktuelles Laufwerk
          pea       2(a2)
          move.w    #dgetpath,-(sp)
          trap      #gemdos
          addq.l    #8,sp

          ; \ und Maske anhängen

          clr.w     d1              ;Stringlänge ermitteln
fsel_lp:  tst.b     0(a2,d1.w)      ;Nullbyte?
          beq.s     fsel_le
          addq.w    #1,d1
          bra.s     fsel_lp

fsel_le:  move.b    #'\',0(a2,d1.w)

          clr.w     d2
          movea.l   8(a6),a3        ;Zeiger auf Maske
```

```
fsel_l2:  move.b    0(a3,d2.w),1(a2,d1.w)
          beq.s     fsel_e2
          addq.w    #1,d1
          addq.w    #1,d2
          bra.s     fsel_l2

fsel_e2:
          ; Aufruf der GEM-File-Selector-Box

          move.w    #90,control       ;Funktions-Opcode
          clr.w     control+2
          move.w    #2,control+4
          move.w    #2,control+6
          clr.w     control+8

          move.l    a2,addr_in        ;vorbereiteter Pfadname
          move.l    12(a6),addr_in+4  ;und Default-Dateiname

          jsr       aes

          tst.w     int_out+2         ;Abbruch statt OK angeklickt?
          bne.s     fsel_na
          clr.w     d0                ;dann mit Fehler abbrechen
          bra       fsel_q
fsel_na:  movea.l   12(a6),a3
          tst.b     (a3)              ;Dateiname überhaupt ausgewählt?
          bne.s     fsel_l3
          clr.w     d0                ;nein, dann siehe oben
          bra       fsel_q

          ; Jetzt müssen wir nur noch aus dem Pfad- und dem Dateinamen
          ; einen kombinierten Pfad- und Dateinamen machen:

fsel_l3:  tst.b     (a2)+             ;Ende des Pfadnamens suchen
          bne.s     fsel_l3
fsel_l4:  cmpi.b    #'\',-(a2)        ;Backslash suchen
          bne.s     fsel_l4

          addq.l    #1,a2             ;Backslash stehen lassen

fsel_l5:  move.b    (a3)+,(a2)+       ;Dateiname kopieren
          bne.s     fsel_l5

          move.w    #1,d0             ;Funktionswert: alles OK

fsel_q:   movem.l   (sp)+,d1-d7/a0-a5
          unlk      a6
          rts
```

```
          ; .DATA- und .BSS-Segment werden nur vom Hauptprogramm
          ; und nicht vom Unterprogramm 'dateiname' benötigt.

          .DATA

maske:    .DC.b '*.*',0

default:  .DC.l 0,0,0,0

err_text: .DC.b 'Ungültige Auswahl!',0

          .BSS

ausgabe:  .DS.b 64

          .END
```

5.8 Fenster

Bisher habe ich es mir in meinen Beispielprogrammen recht leicht gemacht: Eventuelle Bildschirm-Ausgaben wurden entweder mit der VDI-Funktion v_gtext oder aber mit der GEMDOS-Funktion Cconws irgendwo auf den Bildschirm geschrieben. Das sah nicht besonders schön aus, da das Desktop-Grau durch ein weißes Rechteck zerstört wurde und in diesem der Text stand. Außerdem fällt es so schwer zu kennzeichnen, welche Ausgaben auf dem Bildschirm zusammengehören. Von der Möglichkeit, daß mehrere Programme gleichzeitig Bildschirm-Ausgaben machen möchten (z.B. GEM-Applikation und Uhr-Accessory), will ich gar nicht erst reden.

Deshalb gibt es Fenster, oder auf Englisch: Windows. Fenster sind zunächst einmal nur Rechtecke auf dem Bildschirm, die einen Arbeitsbereich umschließen. Wie dieser Arbeitsbereich aussieht, ist allein Sache des Programms, dem das Fenster zugeteilt ist. In GEM ist das jedoch nicht alles: Der Rand des Fensters, der Fensterrahmen, kann Bedienungselemente enthalten, die es ermöglichen, daß der Benutzer Einfluß auf das Fenster hat: Er kann zum Beispiel Position und Größe mit Hilfe der Maus frei bestimmen. Aber das wissen Sie ja bestimmt.

Sie sollten jetzt schon einmal einen Blick in Ihre Dokumentation der Window-Library werfen. Selbst wenn Sie nicht alles verstehen, so schadet Ihnen ein Überblick für die folgenden Erläuterungen sicher nicht.

Betrachten wir zuerst einmal die Schritte, die nötig sind, um ein Fenster auf dem Bildschirm darzustellen. Man nennt das auch das Öffnen eines Fensters. Beim Öffnen erhalten Sie übrigens eine Zahl, das sogenannte Window-Handle, zurück, das es ermöglicht, mehrere Fenster voneinander zu unterscheiden.

Nun, das Öffnen eines Windows geschieht in zwei Schritten: Zuerst müssen Sie eine Art Antrag stellen, das Fenster öffnen zu dürfen. Dabei müssen Sie auch schon angeben, welche Bedienungselemente im Fensterrahmen erscheinen sollen, wenn es im zweiten Schritt auf dem Bildschirm erscheint, und wie groß das Fenster maximal werden darf. Das geschieht mit der AES-Funktion wind_create. Funktionswert ist entweder eine Fehlernummer oder das Window-Handle. Eine Fehlermeldung (Funktionswert kleiner als Null) bekommen Sie übrigens dann, wenn Sie versuchen, das achte Fenster zu öffnen - mehr als sieben geht nicht. Entscheidend dafür ist nicht, wie viele Fenster schon sichtbar sind, sondern wie viele mit wind_create angemeldet wurden.

Nach dem wind_create-Aufruf müssen Sie eine zweite Funktion bemühen, die das Fenster auf den Bildschirm bringt: wind_open. Koordinaten sind das Window-Handle (das haben Sie von wind_create) und die Koordinaten, an denen das Fenster erscheinen soll (x, y, Breite, Höhe). Diese Koordinaten beziehen sich übrigens nicht auf den Arbeitsbereich (also nicht auf den Teil innerhalb des Rahmens), sondern auf das gesamte Fenster einschließlich des Rahmens.

Nach diesem Aufruf steht der Rahmen auf dem Bildschirm. Der Arbeitsbereich wird jedoch nicht automatisch gelöscht, das heißt nach dem Start des Programms bleibt er grau. Um diesen zu löschen, können wir jedoch mit der Hintergrundfarbe (Index 0) ein gefülltes Rechteck über den Arbeitsbereich malen, so daß der Inhalt des Fensters in strahlendem Weiß erscheint. Das ist bequem mit ein paar VDI-Routinen zu erledigen.

Dabei ergibt sich ein neues Problem: Wo ist überhaupt der Arbeitsbereich des Fensters? Beim Öffnen haben wir ja nur die Werte für das gesamte Fenster angeben müssen. Je nachdem welche Bedienungselemente wir ausgewählt haben, fällt unser Arbeitsbereich verschieden aus, da ja mehr oder weniger Platz für den Rand beschlagnahmt wird. Da kommt uns die Funktion wind_calc gerade recht: Dieser können Sie entweder die Koordinaten des gesamten Fensters oder die des Arbeitsbereiches übergeben, und Sie erhalten Lage und Ausmaße des jeweils anderen Bereiches zurück.Dabei ist jedoch eine wichtige Kleinigkeit zu beachten: GEM benutzt zwei unterschiedliche Verfahren, um Eckpunkte eines Rechtecks in Koordinaten auszudrücken. Wenn Sie also mit wind_calc die Größe des

Arbeitsbereiches ermitteln (eigentlich auch nur ein Rechteck), dann erhalten Sie x- und y-Koordinaten für die linke obere Ecke sowie die Breite und Höhe des Rechtecks in Bildpunkten (Pixel). Wenn Sie jedoch das gleiche Rechteck mit VDI-Funktionen löschen wollen, so müssen Sie zusätzlich zu den x- und y-Werten für die linke obere Ecke nicht Breite und Höhe, sondern die x- und y-Koordinate des unteren rechten Punktes angeben. Wir unterscheiden also zwischen AES-Koordinaten (x/y/w/h; w und h bedeuten width=Breite und height=Höhe) und VDI-Koordinaten (x1/y1/x2/y2). Die Umrechnung erfolgt ganz einfach: x2=x+w-1 und y2=y+h-1. Daß wir die Breite zu x und die Höhe zu y addieren, dürfte einleuchten; die -1 kommt daher, daß ja selbst eine horizontale oder vertikale Linie (ein sehr dünnes Rechteck) noch eine Breite von 1 hat (jedenfalls auf dem Bildschirm unseres Rechners).

Jetzt haben wir also ein Fenster auf dem Bildschirm und könnten es benutzen. Doch bevor ich Ihnen sage, was es dabei alles zu beachten gibt, zeige ich Ihnen, wie das Fenster wieder vom Bildschirm verschwindet. Dazu gibts dann auch ein Beispielprogramm, und Sie haben Gelegenheit, alles bis dahin Gelernte anzuwenden.

So, wie ein Fenster in zwei Schritten geöffnet werden muß (create und open), so gibt es auch zwei Funktionen, die es schließen können. Die erste heißt wind_close, ist das Gegenstück zu wind_open und entfernt das Fenster vom Bildschirm. Das Window-Handle bleibt erhalten; Sie könnten das Fenster also mit wind_open erneut erscheinen lassen, ohne daß Sie vorher wind_create bemühen müssen. Um das Handle endgültig freizugeben, was Sie übrigens auch tun müssen, wenn Sie das Fenster nicht mehr benötigen (z.B. am Programmende), müssen Sie nur die Funktion wind_delete aufrufen. Der einzige Parameter ist das Window-Handle. Damit gehört dieses Fenster der Vergangenheit an.

Bevor wir ein Fenster zur Demonstration öffnen, will ich Ihnen noch erklären, welche Bedienungselemente es gibt und wie Sie dem Rechner klarmachen, welche dieser Elemente Sie in einem bestimmten Fenster haben möchten. Letzteres ist einfach: beim wind_create-Aufruf müssen Sie einen Wert angeben, der die Ausstattung des anzumeldenden Fensters festlegt. Dabei haben die Bits 0 bis 11 bestimmte Bedeutungen; ein gesetztes Bit sagt, daß die durch dieses Bit repräsentierte Eigenschaft des Fensters vorhanden ist.

Die Bits selber entnehmen Sie bitte der untenstehenden Tabelle. Hier werden erst einmal die Möglichkeiten der einzelnen Elemente kurz an-

gesprochen. Ich verwende dabei den offiziellen Namen eines jeden Elements, den Sie in jeder guten Beschreibung der Routine wiederfinden werden.

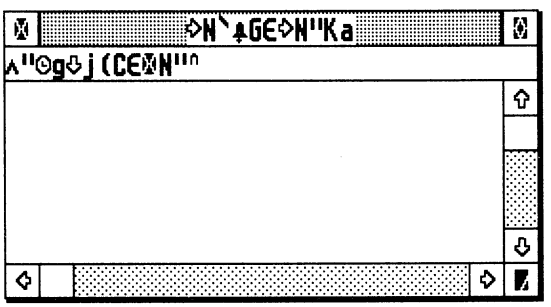
NAME gibt an, ob das Fenster eine Titelzeile besitzt. Die Titelzeile ist der oberste Teil eines Fensters, in dem z.B. im Desktop das Laufwerk und der aktuelle Pfad stehen. CLOSER legt fest, ob das Fenster links oben ein Schließfeld haben soll. FULLER bestimmt die rechte obere Fensterecke; dort kann das Feld erscheinen, dessen Anklicken gewöhnlich eine Vergrößerung des Fensters auf die maximal mögliche Größe bewirkt. MOVER entscheidet darüber, ob der Anwender mit der Titelzeile das Fenster verschieben darf oder nicht. INFO ermöglicht die Einrichtung einer speziellen Textzeile, die nicht zum eigentlichen Arbeitsbereich gehört. Sie liegt direkt unter der Titelzeile. Im Desktop steht hier die Anzahl der Objekte und deren Platzbedarf auf dem Massenspeicher. Mit SIZER können Sie bestimmen, ob der Benutzer die Größe des Fensters ändern darf; dazu wird dann in der rechten unteren Ecke ein entsprechendes Feld angelegt. UPARROW, DNARROW und VSLIDE geben an, ob Sie Pfeile nach oben (UP) und unten (DN) und einen vertikalen Schieber zwischen den Pfeilen wünschen. Alle drei befinden sich stets am rechten Fensterrand. LFARROW, RTARROW und HSLIDE beziehen sich auf die entsprechenden Elemente am unteren Fensterrand und bedeuten demnach Links- (LF) und Rechtspfeil (RT) sowie horizonaler Schieber.

Die folgende Tabelle faßt dies noch einmal kurz zusammen. Dort finden Sie auch die Bitnummer und den dieser Nummer entsprechenden dezimalen Wert. Wenn Sie also Titel und Infozeile wünschen, so müssen Sie die Eins und die 16 addieren bzw. die Bits 0 und 4 setzen, was eigentlich das gleiche ist. Tun Sie das, was Sie für einfacher halten.

Offizieller Name	Bit-Nummer	Wert	Bedeutung
NAME	0	1	Titelzeile
CLOSER	1	2	Schließfeld
FULLER	2	4	Volle-Größe-Feld
MOVER	3	8	Fenster kann bewegt werden
INFO	4	16	Info-Zeile
SIZER	5	32	Größe kann verändert werden
UPARROW	6	64	Pfeil nach oben
DNARROW	7	128	Pfeil nach unten
VSLIDE	8	256	vertikaler Schieber
LFARROW	9	512	Pfeil nach links
RTARROW	10	1024	Pfeil nach rechts
HSLIDE	11	2048	horizontaler Schieber

Die Frage: "Woher weiß denn mein Programm überhaupt, ob der Benutzer einen dieser Knöpfe angeklickt hat?" will ich noch zurückstellen. Soviel sei verraten: Es ist nicht schwer, und wir haben dieses Thema bereits angesprochen. Doch für's erste soll es uns genügen, überhaupt ein Fenster auf den Bildschirm zu zaubern.

Das erledigt folgendes Beispielprogramm, das nach den bisherigen Erläuterungen nicht schwer zu verstehen sein dürfte. Es öffnet ein Fenster, löscht den Arbeitsbereich, nachdem es dessen Größe berechnet hat, schreibt etwas in das Fenster hinein und wartet auf einen Tastendruck. Dann schließt es das Fenster wieder.



GFA-BASIC

```
'
' Fenster öffnen und wieder schließen
' GFA-BASIC   MP 30-11-88   WIND1.GFA
'
VOID APPL_INIT()
'
handle%=WIND_CREATE(4095,20,20,280,150)
' (4095 bedeutet: Alle Bedienelemente sind aktiv)
'
IF handle%<0   ! sollte eigentlich nicht passieren
  form_alert(1,"[3][Sorry!|Kein Window-Handle mehr frei!][OK]")
ELSE
  VOID WIND_OPEN(handle%,20,20,280,150)
  '
  ' Arbeitsbereich ausrechnen:
  '
  VOID WIND_CALC(1,4095,20,20,280,150,x%,y%,w%,h%)
  '
  ' Arbeitsbereich löschen
  '
  DEFFILL 0        ! Füllen mit Hintergrundfarbe
```

```
  PBOX x%,y%,x%+w%-1,y%+h%-1
  '
  VOID GEMDOS(7)   ! Warten auf Taste
  '
  VOID WIND_CLOSE(handle%)
  VOID WIND_DELETE(handle%)
  '
ENDIF
'
VOID APPL_EXIT()
END
'
```

Omikron-BASIC

```
'
' Fenster öffnen und wieder schließen
' Omikron    MP  30-11-88   WIND1.BAS
'
Appl_Init
V_Opnvwk
'
Wind_Create(4095,20,20,280,150,Handle)
IF Handle<0
  THEN FORM_ALERT (1,"[3][Sorry!|Kein Window-Handle mehr frei!][OK]",Ret)
  ELSE Wind_Open(Handle,20,20,280,150)
       '
       ' Arbeitsbereich ermitteln lassen:
       '
       Wind_Calc(1,4095,20,20,280,150,X,Y,W,H)
       '
       Vsf_Interior(0)'    Füllen mit Hintergrundfarbe
       Vsf_Perimeter(0)'   ohne Umrandung
       '
       V_Bar(X,Y,X+W-1,Y+H-1)
       '
       Vsf_Perimeter(1)'   Umrandung wieder einstellen
       '
       GEMDOS (,7)'   Warten auf Taste
       '
       Wind_Close(Handle)
       Wind_Delete(Handle)
  ENDIF
'
V_Clsvwk
Appl_Exit
END
'
```

C

```
/*****************************************/
/*   Fenster öffnen und wieder schließen   */
/*   Laser C     MP  30-11-88    WIND1.C   */
/*****************************************/

#include <osbind.h>
#include "gem_inex.c"

int  whandle,
     pxyarray[4],
     x,y,w,h;

main()
{
  gem_init();

  whandle = wind_create (4095, 20, 20, 280, 150);
            /* 4095 = alle möglichen Bedienelemente */

  if (whandle < 0)
    form_alert (1, "[3][Sorry!¦Kein Window-Handle mehr frei!][OK]");
  else
  {
    wind_open (whandle, 20, 20, 280, 150);

    /* Arbeitsbereich berechnen lassen: */

    wind_calc (1, 4095, 20, 20, 280, 150, &x, &y, &w, &h);

    /* Umrechnen von Höhe/Breite in zweiten Eckpunkt: (x2/y2) */
    pxyarray[0] = x;          pxyarray[1] = y;
    pxyarray[2] = x+w-1;      pxyarray[3] = y+h-1;

    /* Arbeitsbereich löschen: */

    vsf_interior (handle, 0);   /* Füllen mit Hintergrundfarbe */
    vsf_perimeter (handle, 0);  /* keine Umrandung */

    v_bar (handle, pxyarray);

    vsf_perimeter (handle, 1);  /* Umrandung wieder einschalten */

    Crawcin();   /* Warten auf Taste */
        wind_close (whandle);
    wind_delete (whandle);
  }
    gem_exit();
}
```

Assembler

```
          ;
          ; Fenster öffnen und wieder schließen
          ; Assembler   MP  30-11-88    WIND1.Q
          ;

          INCLUDE 'GEM_INEX.Q'

gemdos    = 1
crawcin   = 7

          TEXT

main:     jsr       gem_init

          ; wind_create

          move.w    #100,control
          move.w    #5,control+2
          move.w    #1,control+4
          clr.w     control+6
          clr.w     control+8

          move.w    #4095,int_in      ;alle möglichen Bedienelemente
          move.w    #20,int_in+2      ;maximale Größe des Fensters (x)
          move.w    #20,int_in+4      ;(y)
          move.w    #280,int_in+6     ;(Breite)
          move.w    #150,int_in+8     ;(Höhe)

          jsr       aes

          tst.w     int_out           ;negativ? Dann Fehler!
          bmi       error
          move.w    int_out,whandle   ;sonst als Window-Handle merken

          ; wind_open


          move.w    #101,control      ;Funktions-Nummer
          move.w    whandle,int_in    ;Handle von wind_create

          ; Alle übrigen Werte sind mit denen von wind_create identisch,
          ; also noch in den Parameter-Arrays!

          jsr       aes
```

```
; Arbeitsbereich berechnen (wind_calc):

move.w    #108,control
move.w    #6,control+2
move.w    #5,control+4
clr.w     control+6
clr.w     control+8

move.w    #1,int_in         ;1 -> Arbeitsbereich ausrechnen
move.w    #4095,int_in+2    ;Eigenschaften des Fensters
move.w    #20,int_in+4      ;äußere Ausmaße
move.w    #20,int_in+6
move.w    #280,int_in+8
move.w    #150,int_in+10

jsr       aes

; Die Ergebnis-Koordinaten können für den nächsten
; Schritt (löschen des Arbeitsbereiches) in den
; AES-Arrays gelassen werden.

; vsf_interior (Füllmuster):

move.w    #23,contrl
clr.w     contrl+2
clr.w     contrl+4
move.w    #1,contrl+6
move.w    #1,contrl+8
move.w    handle,contrl+12
clr.w     intin             ;Füllen mit Hintergrundfarbe
jsr       vdi

; vsf_perimeter (Umrandung ausschalten):

move.w    #104,contrl
clr.w     contrl+2
clr.w     contrl+4
move.w    #1,contrl+6
move.w    #1,contrl+8
move.w    handle,contrl+12
clr.w     intin             ;Flag: Umrahmung ausschalten
jsr       vdi

; v_bar (löscht den Arbeitsbereich):
; (Koordinaten von wind_calc, müssen aber von x/y/breit/hoch
; in x1/y1/x2/y2 umgerechnet werden.

move.w    #11,contrl        ;Opcode für graphische
move.w    #2,contrl+2       ;Grundfunktionen
```

```
        clr.w     contrl+6
        clr.w     contrl+8
        move.w    #1,contrl+10     ;Funktionsnummer für v_bar
        move.w    handle,contrl+12

break:

        move.w    int_out+2,d0     ;x1 = x aus wind_calc
        move.w    d0,ptsin
        move.w    int_out+4,d1     ;y1 = y aus wind_calc
        move.w    d1,ptsin+2
        add.w     int_out+6,d0     ;x + Breite...
        subq.w    #1,d0            ;... - 1...
        move.w    d0,ptsin+4       ;... = x2
        add.w     int_out+8,d1     ;y + Höhe...
        subq.w    #1,d1            ;... - 1...
        move.w    d1,ptsin+6       ;... = y2

        jsr       vdi

        ; vsf_perimeter (Umrandung wieder einschalten):

        move.w    #104,contrl
        clr.w     contrl+2
        clr.w     contrl+4
        move.w    #1,contrl+6
        move.w    #1,contrl+8
        move.w    handle,contrl+12
        move.w    #1,intin         ;Flag: Umrahmung einschalten
        jsr       vdi

        ; Warten auf Taste:

        move.w    #crawcin,-(sp)
        trap      #gemdos
        addq.l    #2,sp

        ; wind_close:

        move.w    #102,control
        move.w    #1,control+2
        move.w    #1,control+4
        clr.w     control+6
        clr.w     control+8

        move.w    whandle,int_in

        jsr       aes
```

```
        ; wind_delete:

        move.w    #103,control      ;nur die Funktionsnummer muß
        jsr       aes               ;geändert werden

quit:   jsr       gem_exit

        clr.w     -(sp)
        trap      #gemdos


error:  move.w    #52,control       ;Form-Alert
        move.w    #1,control+2
        move.w    #1,control+4
        move.w    #1,control+6
        clr.w     control+8
        move.l    #err_txt,addr_in ;String, der die Box beschreibt
        move.w    #1,int_in         ;erster Knop ist Default-Button
        jsr       aes

        bra.s     quit


        DATA

err_txt: DC.b '[3][Sorry!¦Kein Window-Handle mehr frei!][OK]',0          BSS

whandle: DS.w 1

        END
```

5.8.1 Fenster in Aktion

Ein Fenster auf den Bildschirm zu bringen und wieder zu löschen, mag ja ganz nett sein, doch Sinn und Zweck ist es wohl, Informationen in diesem Fenster darzustellen. Daneben gibt es noch ein paar andere Dinge, die wir an unserem Fenster aus dem letzten Beispielprogramm verbessern müßten. So war ja bisher, wie Sie vielleicht beim Ablauf dieses Programms bemerkt haben, der Inhalt der Titel- und der Infozeile zufällig, und auch die vorhandenen Bedienungselemente zeigten, wenn man Sie anklickte, keine Reaktion. Das wollen wir nun ändern.

Für das erste Problem, also die Titel- und Infozeile, gibt es eine relativ einfache Lösung: Die AES-Funktion wind_set. Das ist eine sehr vielseitige Funktion, die jede nur denkbare Veränderung eines Fensters durchführen kann. Dabei ist es egal, ob sich das Fenster bereits auf dem

Bildschirm befindet oder nicht, das heißt Sie können wind_set vor oder nach dem wind_open-Aufruf anwenden. Das Fenster muß lediglich mit wind_create schon angemeldet sein.

In der Standardversion erhält die Funktion sechs Parameter. Der erste ist das Window-Handle; die Funktion erkennt hieran, auf welches Fenster sich die geplante Änderung bezieht. Das zweite ist ein Wert, der die Art der Änderung beschreibt - es gibt nicht weniger als zehn Möglichkeiten, diese Funktion einzusetzen. Daran schließen sich vier Integer-Parameter an, deren Bedeutung von der gewählten Unterfunktion abhängt. Wir geben Ihnen hier erstmal die vielsagenden Namen 1 bis 4 (in der Reihenfolge, wie sie im Aufruf erscheinen).

Für uns sind die Unterfunktionen 2 und 3 interessant. Mit Nummer 2 kann die Titelzeile eines Fensters eingestellt werden, während Funktion 3 für die Info-Zeile zuständig ist. Dazu benötigt die Funktion aber auch noch einen String, der die gewünschte Zeile enthält. Dieser soll laut offizieller Dokumentation mit zwei Nullbytes abgeschlossen sein; die Erfahrung hat jedoch gezeigt, daß ein Nullbyte auch reicht. Die Adresse dieses Strings ist jedenfalls an wind_set zu übergeben, wobei der Parameter 1 das höherwertige Wort enthält und der Parameter 2 das niederwertige (die Startadresse belegt ja vier Bytes und kann in einer einzigen Integer-Variablen nicht untergebracht werden). Die Parameter 3 und 4 haben keine Bedeutung.

Nun gibt es aber von Programmiersprache zu Programmiersprache ein paar Kleinigkeiten, die nicht ganz unwichtig sind. Gehen wir in der gewohnten Reihenfolge vor:

GFA-BASIC

Sie könnten sich natürlich mit VARPTR die Startadresse eines Strings besorgen, und auch das abschließende Nullbyte läßt sich durch A$=A$+CHR$(0) leicht beschaffen. Allerdings garantiert BASIC nicht, daß eine Stringvariable im Laufe des Programms ihren Platz beibehält (Stichwort: Garbage Collection). Deshalb sollten Sie einen kleinen Speicherbereich mit Malloc() reservieren, in den der String in einer Schleife hineinkopiert wird (mit Poke-Befehlen). Dann übergeben Sie einfach die Startadresse als High- und Low-Word an die Funktion. Die Beispielprogramme dürften das deutlich machen.

Omikron-BASIC

Es gilt sinngemäß das gleiche wie unter GFA-BASIC. Allerdings können Sie hier wirklich einen String übergeben (statt der Parameter 1 bis 4). Zusätzlich müssen Sie einen Speicherbereich angeben (mit MEMORY-Funktion besorgen), der ausreicht, um diesen String samt Nullbyte aufzunehmen. Sie ersparen sich also das Einpoken von Hand.

C

Hier liegt ein String, wenn Sie ihn nicht gerade als lokale Variable einer Funktion deklariert haben, immer an der gleichen Startadresse. Die Startadresse wird bekanntlich durch den Namen des Strings (ohne []) angegeben oder durch den gesamten String in Anführungszeichen (bei Stringkonstanten, wie im folgenden Beispiel). Der Trick in C ist aber, daß Sie die Parameter 1 und 2 durch ein einzelnes Langwort ersetzen können. Der Compiler macht dann (ohne es zu wissen) alles richtig. Betrachten Sie einfach das Beispielprogramm, dann wissen Sie, was ich meine.

Assembler

Hier geht es ähnlich zu wie in C: Sie können die Befehle

```
move.w #high,int_in+4
move.w #low,int_in+6
```

durch

```
move.l #adresse,int_in+4
```

ersetzen. Das Ergebnis ist das gleiche.

So, damit wäre der erste Punkt geklärt. Und der größte Brocken bleibt: Die Abfrage der Benutzeraktivitäten. Und das ist (wie so vieles) im Prinzip kein Problem. Man hat nämlich all das, was ein Anwender mit einem Mauszeiger an einem Fenster anstellen kann, als Ereignis mit in die Ereignis-Verwaltung des AES gepackt. Es gibt zwar keine Funktion evnt_close, die darauf wartet, daß die Close-Box eines Fensters angeklickt wird; statt dessen hat man alle möglichen Aktivitäten zusammengefaßt und macht daraus eine Nachricht. Damit sind wir nicht nur bei einem der wichtigsten Ereignisse überhaupt angelangt; Sie sehen nun auch, wie einzelne Teile des AES zusammenarbeiten und für sich alleine

überhaupt keinen Sinn ergeben. Ohne Fenster keine Nachrichten, und ohne Nachrichten (und ohne die Ereignisfunktionen) keine Möglichkeit, vernünftig mit Fenstern zu hantieren!

Die Nachrichten wurden schon kurz im Kapitel 5.6 angesprochen, ohne daß sie für uns einen Sinn gehabt hätten. Eine Nachricht besteht aus 8 Worten, die von 0 bis 7 durchnumeriert sind. Das Wort 0 gibt an, um welche Nachricht es sich handelt. Wenn sich diese Nachricht auf ein Fenster bezieht, so finden Sie im Wort 3 außerdem das Handle des Fensters, auf das sich die Nachricht bezieht. Die Worte 4 bis 7 haben je nach Nachricht unterschiedliche Bedeutungen; meist werden sie dazu benutzt, um Koordinaten zu übermitteln. Übrigens: Die Nachricht selbst bekommen Sie mit der AES-Funktion evnt_mesag, der als Parameter die Startadresse eines Bereichs zu übergeben ist, der die 8 Worte aufnehmen kann. Mit evnt_multi ist die Abfrage selbstverständlich auch möglich.

Ein Beispiel: Das Ereignis mit der Kennzahl 28 (der offizielle Name heißt WM_MOVED) sagt einem Programm, daß der Benutzer ein Fenster an eine andere Stelle bewegen möchte. Bevor das AES die Nachricht losschickt, hat der Benutzer bereits über das Ihnen sicherlich bekannte Gummi-Rechteck die neue Position vorgegeben. Die Aufgabe des Programms ist es nun, den Worten 4, 5, 6 und 7 die neuen Koordinaten (x, y, Breite und Höhe; letztere haben sich beim Verschieben natürlich nicht verändert) zu entnehmen. Dann muß das Programm manchmal prüfen, ob die gewünschte Aktion zulässig ist, denn es könnte ja durchaus vernünftige Gründe dafür geben, die Bewegung eines Fensters auf einen bestimmten Bildschirmausschnitt zu beschränken (oder überhaupt im Bildschirm zu halten). Die Koordinaten müßten also eventuell geändert werden. Dann erst gibt das Programm den Befehl, die neue Position des Fensters einzustellen. Dazu dient wieder die Funktion wind_set, diesmal mit einer anderen Unterfunktions-Nummer. Im nächsten Programm können Sie diese Schritte gut nachvollziehen.

Ein ganz wesentlicher Punkt bei der Fensterprogrammierung ist auch das sogenannte Redraw-Handling. Redraw heißt auf Deutsch neuzeichnen oder nochmal zeichnen und bezieht sich in unserem Fall auf den Fensterinhalt. Ich möchte hier nur einen kleinen Einblick in dieses Verfahren geben; die wirklichen Möglichkeiten (man sollte eher von Pflichten sprechen) lernen Sie dann im nächsten Kapitel kennen.

Denken Sie sich ein Fenster mit irgendeinem Inhalt. Das Fenster wird nun so weit nach rechts unten verschoben, daß ein Teil des Inhalts vom Bildschirm verschwunden ist. Anschließend machen Sie das wieder rückgängig, das heißt der alte Inhalt soll wieder auf dem Bildschirm erschei-

nen. Ein zweiter Fall: Ein Fenster ist so klein, daß sein Inhalt nur teilweise abgebildet werden kann. Nun wird es vergrößert, um alle Informationen auf einen Blick haben zu können.

Beiden Situationen ist eines gemeinsam: Nach dem Verschieben bzw. Vergrößern des Fensters müssen Teile des Fensters neu aufgebaut werden. Beim Verschiebe-Beispiel wird das wohl besonders deutlich: Wenn ich das Fenster nach unten schiebe, so kann der verbleibende Teil mit den VDI-Raster-Funktionen auf die neue Position kopiert werden, weil der komplette neue sichtbare Teil ja auch vorher schon auf dem Bildschirm vorhanden war. Andersherum geht das jedoch nicht, weil der neue Fensterinhalt nicht komplett auf dem Bildschirm steht, also auch nicht einfach an eine andere Stelle kopiert werden kann.

Kurz: Es gibt Situationen, in denen das Neuzeichnen eines Fensters oder eines Teils eines Fensters erforderlich ist. Solche Situationen werden vom AES automatisch erkannt und der Applikation in Form einer Nachricht mitgeteilt. Das Neuzeichnen selbst ist Sache der Applikation. GEM sorgt nur dafür, daß in einem solchen Fall die Fensterränder, die Info-Zeile und solche Dinge wieder erscheinen.

In unserem nächsten Programm ist das Neuzeichnen recht einfach zu bewerkstelligen. Wenn das Fenster neu gezeichnet werden muß, dann erhalten wir eine Nachricht. Die Kennzahl der Nachricht ist 20. Sobald wir diese Nachricht erhalten, wird das Unterprogramm zur Ausgabe des Fensterinhalts aufgerufen. Dieses Unterprogramm löscht auch gleichzeitig den Arbeitsbereich.

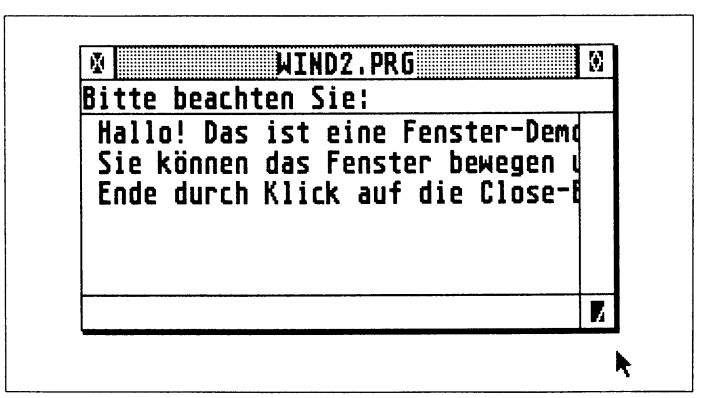
Das sind die wesentlichen Veränderungen des nächsten Programms gegenüber dem letzten Beispiel. Außerdem gibt es ein paar kleine Erweiterungen:

- Der Mauszeiger wird vor einer Grafik-Ausgabe abgeschaltet. Das ist nötig, weil sonst bei der nächsten Bewegung des Mauszeigers die neue Zeichnung der unmittelbaren Umgebung des Zeigers wieder gelöscht würde. Das können Sie im vorigen Fenster-Programm beobachten: Starten Sie das Programm und bewegen Sie den Mauszeiger in eine Position innerhalb des Fensters, bevor es sichtbar wird (also noch während des Ladens). Wenn das Fenster dann gelöscht, der Arbeitsbereich also weiß wird, ist der Mauszeiger verschwunden. Wenn Sie die Maus nun bewegen, wird zwar der Zeiger wieder sichtbar; es bleibt aber ein häßlicher grauer Fleck zurück.

Deshalb müssen Sie vor einer Ausgabe mit VDI-Funktionen den Mauszeiger mit v_hide_c (VDI-Funktion) ausschalten und anschließend mit v_show_c (ebenfalls VDI) wieder einschalten. Das gilt jedoch nicht für AES-Funktionen, die ja auch ab und zu Bildschirm-Ausgaben machen müssen - die schalten den Mauszeiger selbst ab, so daß Sie sich darum nicht mehr kümmern müssen.

- Wenn das Fenster kleiner ist als der Inhalt, der im Arbeitsbereich dargestellt werden soll, dann darf natürlich nur ein Ausschnitt der geplanten Ausgabe erscheinen; die Ausgabe muß auf einen bestimmten Bereich, nämlich den Arbeitsbereich, begrenzt werden. Das erledigt bequem die VDI-Funktion vs_clip, die alle nachfolgenden Ausgaben auf ein anzugebendes Rechteck beschränkt. Clip bedeutet denn im Englischen auch etwa abschneiden.

Beide Tricks können Sie im folgenden Listing gut wiedererkennen: Das Programm öffnet ein Fenster, so wie es im letzten Programm auch schon geschah. Doch werden die Koordinaten diesmal in Variablen gehalten, so daß sie im Verlauf des Programms geändert werden können. Titel- und Infozeile werden mit vernünftigen Inhalten gefüllt, im Fenster selbst steht ein dreizeiliger Text (Stichwort: clipping, s.o.) und die Randelemente (Size,Box, Move-Bar, Full-Box und Close-Box erhalten die üblichen Funktionen. Mit der Close-Box können Sie das Programm übrigens beenden. Auch das Redraw-Verfahren ist in einer sehr einfachen Form schon eingebaut:



GFA-BASIC

```
'
' Fenster mit Funktionen (vergrößern, ...)
' GFA-BASIC     MP 13-12-88      WIND2.GFA
'
DEFINT "a-z"    ! alle Variablen sind Integer -> erspart uns viele %s
'
DIM puffer&(7)  ! Platz für 8 Worte einer Nachricht (evnt_mesag)
'
VOID APPL_INIT()
'
' Bildschirmauflösung feststellen:
'
rez=XBIOS(4)     ! Getrez liefert 0=low, 1=mid, 2=high
x_max=640+320*(rez=0)
y_max=200-200*(rez=2)
'
' Anfangs-Fenstergröße vorgeben:
'
wx=20    ! x-Koordinate
wy=20    ! y
ww=280   ! w = width  (Breite)
wh=150   ! h = height (Höhe)
'
' Fenster anmelden für maximale Größe und alle Elemente (63)
' außer Pfeile und Schieber
'
whandle=WIND_CREATE(63,0,0,x_max,y_max)
'
IF whandle<0     ! Dann wird wohl ein Fehler aufgetreten sein
  VOID FORM_ALERT(1,"[3][Kein Handle mehr verfügbar!][OK]")
ELSE
  '
  ' Titel- und Infozeile einstellen:
  ' Die Strings werden in einen sicheren Speicherbereich
  ' kopiert, da sich ihre Startadresse nicht ändern darf
  '
  title$="WIND2.PRG"+CHR$(0)
  info$="Bitte beachten Sie:"+CHR$(0)
  '
  s=MALLOC(40)
  '
  FOR i=1 TO LEN(title$)              ! Titelzeile
    POKE s-1+i,ASC(MID$(title$,i,1))
  NEXT i
  '
  FOR i=1 TO LEN(info$)               ! Infozeile
    POKE s+19+i,ASC(MID$(info$,i,1))
  NEXT i
  '
```

```
  ' Wir brauchen Low- und High-Words (durch DIV 2^16und MOD 2^16)
  ' Titelzeile: Unterfunktion 2  /  Infozeile: Unterfunktion 3
  '  VOID WIND_SET(whandle,2,s DIV 65536,s MOD 65536,0,0)
  VOID WIND_SET(whandle,3,(s+20) DIV 65536,(s+20) MOD 65536,0,0)
  '
  VOID WIND_OPEN(whandle,wx,wy,ww,wh)
  '
  GOSUB output_text   ! Füllt das Fenster mit Inhalt
  '
  REPEAT
    '
    ' Auf ein Ereignis warten:
    '
    VOID EVNT_MESAG(LPEEK(*puffer&())+4) ! Adresse des ersten Elements
    '
    IF puffer&(0)=20     ! Redraw-Meldung
      GOSUB output_text
    ENDIF
    '
    IF puffer&(0)=23     ! Full-Box wurde angeklickt
      wx=2               ! Fenster auf maximal mögliche
      wy=20              ! Größe bringen (nur Menüzeile und
      ww=x_max-5         ! 2 Punkte an den Rändern werden
      wh=y_max-25        ! gespart)
      VOID WIND_SET(whandle,5,wx,wy,ww,wh)  ! Größe einstellen    ENDIF
    '
    IF puffer&(0)=27     ! Size-Box
      ww=puffer&(6)      ! Größe aus der Nachricht entnehmen
      wh=puffer&(7)
      VOID WIND_SET(whandle,5,wx,wy,ww,wh)
    ENDIF
    '
    IF puffer&(0)=28     ! Move-Box
      wx=puffer&(4)      ! x und y sind verändert
      wy=puffer&(5)
      VOID WIND_SET(whandle,5,wx,wy,ww,wh)
    ENDIF
    '
  UNTIL puffer&(0)=22   !  ... bis Close-Box gedrückt wurde
  '
  VOID WIND_CLOSE(whandle)
  VOID WIND_DELETE(whandle)
  '
  VOID MFREE(s)
ENDIF
'
VOID APPL_EXIT()
'
END
'
```

```
'
PROCEDURE output_text
  HIDEM    ! Maus abschalten, die stört nur
  '
  ' Größe des Arbeitsbereichs ermitteln:
  '
  VOID WIND_CALC(1,63,wx,wy,ww,wh,x,y,w,h)
  '
  ' Umrechnen in x1/y1/x2/y2:
  '
  x1=x
  y1=y
  x2=x+w-1
  y2=y+h-1
  '
  ' Löschen des Arbeitsbereichs:
  '
  BOUNDARY 0
  DEFFILL 0
  PBOX x1,y1,x2,y2
  BOUNDARY 1
  '
  ' Clipping einschalten (für Text-Ausgabe):
  '
  CLIP x1,y1 TO x2,y2
  '
  ' Text ausgeben:
  '
  TEXT x1+8,y1+14,"Hallo! Das ist eine Fenster-Demonstration!"  TEXT
x1+8,y1+30,"Sie können das Fenster bewegen und seine Größe...
... verändern."
  TEXT x1+8,y1+46,"Ende durch Klick auf die Close-Box!"
  '
  ' Clipping wieder ausschalten:
  '
  CLIP OFF
  '
  SHOWM
RETURN
'
```

Auch hier wurde wieder eine Zeile getrennt (erkennbar an den drei Punkten), die Sie bitte wieder in einer Zeile eingeben.

Omikron-BASIC

```
'
' Fenster mit Funktionen (vergrößern, ...)
' Omikron-BASIC   MP  02-12-88   WIND2.BAS
'
Appl_Init:V_Opnvwk
```

```
'
' Bildschirmauflösung feststellen:
'
XBIOS (Rez,4)'    0=low, 1=mid, 2=high
X_Max=640+320*(Rez=0)
Y_Max=200-200*(Rez=2)
'
' Anfangs-Fenstergröße vorgeben:
'
Wx=20:Wy=20:Ww=280:Wh=150
'
Wind_Create(63,0,0,X_Max,Y_Max,Whandle)
IF Whandle<0 THEN
  FORM_ALERT (1,"[3][Kein Handle verfügbar!][OK]",Dummy)195 ELSE
  '
  ' Titel- und Infozeile einstellen:
  '
  Adr= MEMORY(40)'   wird für wind_set benötigt
  Wind_Set(Whandle,2,"WIND2.PRG",Adr)
  Wind_Set(Whandle,3,"Bitte beachten Sie:",Adr+15)
  '
  Wind_Open(Whandle,Wx,Wy,Ww,Wh)
  '
  ' Jetzt rufen wir ein Unterprogramm auf, das den Arbeits-
  ' bereich des Fensters löscht und etwas hineinschreibt:
  '
  Output_Text
  '
  ' *** Warten auf Benutzeraktivitäten ***
  '
  REPEAT
    '
    Evnt_Mesag(Puffer$)'   je 2 Bytes bilden ein Wort
    '
    IF FN P(0)=20 THEN Output_Text'   Redraw-Aufforderung
    '
    IF FN P(0)=23 THEN '   Full-Box wurde angeklickt
      Wx=2:Wy=20:Ww=X_Max-5:Wh=Y_Max-25
      Wind_Set(Whandle,Wx,Wy,Ww,Wh)
    ENDIF
    '
    IF FN P(0)=27 THEN '   Size-Box
      Ww=FN P(6)'    Größe aus dem Nachrichtenpuffer übernehmen
      Wh=FN P(7)
      Wind_Set(Whandle,Wx,Wy,Ww,Wh)
    ENDIF
    '
    IF FN P(0)=28 THEN '   Move-Box
      Wx=FN P(4)
      Wy=FN P(5)
```

```
      Wind_Set(Whandle,Wx,Wy,Ww,Wh)
    ENDIF
    '
  UNTIL FN P(0)=22'   ...bis Close-Box geklickt wurde
  '
  ' Fenster schließen und freigeben:
  '
  Wind_Close(Whandle)
  Wind_Delete(Whandle)
ENDIF
'
V_Clsvwk:Appl_Exit
END
'
'
' Die folgende Funktion wird zur Verarbeitung des Ergebnis-Strings
' von evnt_mesag verwendet. Sie übergeben einfach die Nummer des
' Wortes der Nachricht, das Sie lesen möchten.
'
DEF FN P(N)= CVI( MID$(Puffer$,N*2+1,2))
'
'
DEF PROC Output_Text'   Gibt Inhalt des Fensters aus
  V_Hide_C'   Der Mauszeiger stört hier nur...
  '
  ' Wir müssen die Größe des Arbeitsbereichs errechnen lassen:  '
Wind_Calc(1,63,Wx,Wy,Ww,Wh,X,Y,W,H)
  '
  X2=X+W-1:Y2=Y+H-1'   Umrechnen von x/y/w/h in x1/y1/x2/y2
  '
  ' Löschen des Arbeitsbereichs:
  '
  Vsf_Interior(0)'   Füllmuster: weiße Fläche
  Vsf_Perimeter(0)'  keine Umrandung
  V_Bar(X,Y,X2,Y2)
  Vsf_Perimeter(1)'  Umrandung wieder einschalten
  '
  ' Clipping einschalten:
  '
  Vs_Clip(X,Y,X2,Y2)
  '
  V_Gtext(X+8,Y+14,"Hallo! Das ist eine Fenster-Demonstration!")
  V_Gtext(X+8,Y+30,"Sie können das Fenster bewegen und seine Größe verändern.")
  V_Gtext(X+8,Y+46,"Ende durch Klick auf die Close-Box!")
  '
  Vs_Clip'   ohne Parameter = Clipping wird ausgeschaltet
  '
  V_Show_C(1)'   Mauszeiger wieder anschalten
RETURN
'
```

C

```
/*************************************************/
/*   Fenster mit Funktionen (vergrößern, ...)    */
/*   Laser C       MP  30-11-88       WIND2.C    */
/*************************************************/

#include <osbind.h>
#include "gem_inex.c"

int  whandle,              /* Das hatten wir auch schon in WIND1.C */
     pxyarray[4],
     x,y,w,h;

int  puffer[8];             /* Hierhin kommen die AES-Nachrichten */

int  wx = 20,              /* äußere Fenstergröße mit Startwerten */
     wy = 20,
     ww = 280,
     wh = 150;


void clip_window (pxyarray)    /* Begrenzt alle nachfolgenden VDI- */
     int pxyarray[];                /* Ausgaben auf einen Bereich */
{
  vs_clip (handle, 1, pxyarray);        /* 1: Clipping einschalten */
}


void switch_clipping_off()         /* macht clip_window rückgängig */
{
int  pxyarray[4];

  vs_clip (handle, 0, pxyarray);        /* 0: Clipping ausschalten */
}


void output_text()
{
  /* Mauszeiger ausschalten: */

  v_hide_c (handle);

  /* Arbeitsbereich errechnen: */

  wind_calc (1, 63, wx, wy, ww, wh, &x, &y, &w, &h);

  /* Umrechnen von Höhe/Breite in zweiten Eckpunkt: (x2/y2) */
  pxyarray[0] = x;          pxyarray[1] = y;
  pxyarray[2] = x+w-1;      pxyarray[3] = y+h-1;
```

```
  /* Arbeitsbereich löschen: */

  vsf_interior (handle, 0);   /* Füllen mit Hintergrundfarbe */
  vsf_perimeter (handle, 0);  /* keine Umrandung */
  v_bar (handle, pxyarray);   /* weißes gefülltes Rechteck */
  vsf_perimeter (handle, 1);  /* Umrandung wieder einschalten */

  /* Jetzt schreiben wir irgendeinen Text in das Fenster. */

  clip_window (pxyarray);

  v_gtext (handle, x+8, y+14,
        "Hallo! Das ist eine Fenster-Demonstration!");
  v_gtext (handle, x+8, y+30,
        "Sie können das Fenster bewegen und seine Größe verändern.");
  v_gtext (handle, x+8, y+46,
        "Ende durch Klick auf die Close-Box!");

  switch_clipping_off();

  /* Mauszeiger wieder zeigen: */

  v_show_c (handle, 1);
}


main()
{
  gem_init();

  graf_mouse (0, 0L);   /* Pfeil als Mauszeiger */

  whandle = wind_create (63, 0, 0, x_max, y_max);
            /* 63 = alles, aber keine Pfeile und Schieber */

  if (whandle < 0)
    form_alert (1, "[3][Sorry!|Kein Window-Handle mehr frei!][OK]");
  else
  {
    /* Titel- und Infozeile einstellen */

    wind_set (whandle, 2, "WIND2.PRG", 0, 0);       /* 2 -> Titel und */
    wind_set (whandle, 3, "Bitte beachten Sie:", 0, 0);  /* 3 -> Info */

    wind_open (whandle, wx, wy, ww, wh);

    /* Jetzt soll der Arbeitsbereich des Fensters gelöscht und */
    /* das Fenster mit einem sinnvollen Inhalt gefüllt werden. */
    output_text();   /* Das machen wir in einem Unterprogramm. */
```

```
    /*************************************************************/
    /*                Jetzt kommt das Wichtigste:                */
    /*   Wir warten auf Benutzeraktivitäten und werten sie aus   */
    /*************************************************************/

    do
    {
      evnt_mesag (puffer);      /* Warten auf eine Nachricht */
      switch (puffer[0])
      {
        case 20:                /* Redraw */
          output_text();
          break;

        case 23:                /* Full-Box */
          wx = 2;
          wy = 20;
          ww = x_max - 5;
          wh = y_max - 25;
          wind_set (whandle, 5, wx, wy, ww, wh); /* neue Größe */          break;

        case 27:                /* Size-Box */
          ww = puffer[6];
          wh = puffer[7];
          wind_set (whandle, 5, wx, wy, ww, wh); /* neue Größe */          break;

        case 28:                /* Move-Bar */
          wx = puffer[4];
          wy = puffer[5];
          wind_set (whandle, 5, wx, wy, ww, wh);
          break;
      }
    }   while (puffer[0] != 22);  /* Bis Close-Box betätigt */

    wind_close (whandle);
    wind_delete (whandle);
  }
  gem_exit();
}
Assembler:
          ;
          ; Fenster mit Funktionen (vergrößern, ...)
          ; Assembler     MP  30-11-88       WIND2.Q
          ;

          INCLUDE 'GEM_INEX.Q'

gemdos    = 1
```

```
        TEXT

main:   jsr     gem_init

        ; 'Anständiger' Mauszeiger:

        move.w  #78,control     ;graf_mouse
        move.w  #1,control+2
        move.w  #1,control+4
        move.w  #1,control+6
        clr.w   control+8
        clr.w   int_in          ;0 für Pfeil
        jsr     aes

        ; wind_create

        move.w  #100,control
        move.w  #5,control+2
        move.w  #1,control+4
        clr.w   control+6
        clr.w   control+8

        move.w  #63,int_in      ;alles außer Pfeilen und Schiebern
        move.w  #0,int_in+2     ;maximale Größe des Fensters (x)
        move.w  #0,int_in+4     ;(y)
        move.w  x_max,int_in+6  ;(Breite)  die Werte stammen aus
        move.w  y_max,int_in+8  ;(Höhe)    dem Include-File

        jsr     aes

        tst.w   int_out         ;negativ? Dann Fehler!
        bmi     error
        move.w  int_out,whandle ;sonst als Window-Handle merken

        ; Titel- und Infozeile festlegen:

        move.w  #105,control    ;wind_set
        move.w  #6,control+2
        move.w  #1,control+4
        clr.w   control+6
        clr.w   control+8

        move.w  whandle,int_in
        move.w  #2,int_in+2     ;2: Titelzeile
        move.l  #title,int_in+4

        jsr     aes

        move.w  #3,int_in+2     ;nochmal, aber mit 3 für Infozeile
```

```
        move.l    #info,int_in+4

        jsr       aes

        ; wind_open

        move.w    #101,control      ;Funktions-Nummer

        move.w    whandle,int_in    ;Handle von wind_create
        move.w    wx,int_in+2
        move.w    wy,int_in+4
        move.w    ww,int_in+6
        move.w    wh,int_in+8

        jsr       aes

        ; Löschen des Arbeitsbereiches und Ausgabe eines Textes:

        jsr       text_out

        ;
        ;  Hier beginnt die Hauptschleife:
        ; *** Wir warten auf Ereignisse ***
        ;

        ; evnt_mesag:

loop:   move.w    #23,control
        clr.w     control+2
        move.w    #1,control+4
        move.w    #1,control+6
        clr.w     control+8
        move.l    #puffer,addr_in
        jsr       aes

        ; Die folgenden Zeilen sehen nicht sehr schön aus.
        ; Sie sind ein Ersatz für switch/case aus C

        cmpi.w    #20,puffer        ;20: Redraw-Message
        bne.s     lab1
        jsr       text_out          ;Fenster neuzeichnen
        bra       loop

lab1:   cmpi.w    #23,puffer        ;23: Full-Box
        bne.s     lab2
        move.w    #2,wx             ;Maximale Fenstergröße einstellen
        move.w    #20,wy
        move.w    x_max,d0          ;Breite und Höhe sind abhängig
        subq.w    #5,d0             ;von der Auflösung
        move.w    d0,ww
```

```
          move.w    y_max,d0
          subi.w    #25,d0
          move.w    d0,wh
          bra       wind_set

lab2:     cmpi.w    #27,puffer      ;27: Size-Box
          bne.s     lab3
          move.w    puffer+12,ww    ;Die neue Größe steht im Puffer
          move.w    puffer+14,wh
          bra       wind_set

lab3:     cmpi.w    #28,puffer      ;28: Move-Bar
          bne.s     lab4
          move.w    puffer+8,wx
          move.w    puffer+10,wy

wind_set: move.w    #105,control    ;wind_set zum Einstellen von
          move.w    #6,control+2    ;neuer Größe und Position
          move.w    #1,control+4
          clr.w     control+6
          clr.w     control+8

          move.w    whandle,int_in  ;Fenster, das einzustellen ist
          move.w    #5,int_in+2     ;5: x/y/w/h einstellen
          move.w    wx,int_in+4     ;neue Werte
          move.w    wy,int_in+6
          move.w    ww,int_in+8
          move.w    wh,int_in+10

          jsr       aes

lab4:     cmpi.w    #22,puffer      ;Close-Box betätigt?
          bne       loop            ;nein, dann noch ein Ereignis

          ; Jetzt können wir das Fenster schließen:

          ; wind_close:

          move.w    #102,control
          move.w    #1,control+2
          move.w    #1,control+4
          clr.w     control+6
          clr.w     control+8

          move.w    whandle,int_in

          jsr       aes

          ; wind_delete:
```

```
          move.w    #103,control      ;nur die Funktionsnummer muß
          jsr       aes               ;geändert werden

quit:     jsr       gem_exit

          clr.w     -(sp)
          trap      #gemdos


error:    move.w    #52,control       ;Form-Alert
          move.w    #1,control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          move.l    #err_txt,addr_in ;String, der die Box beschreibt
          move.w    #1,int_in         ;erster Knopf ist Default-Button
          jsr       aes

          bra.s     quit

          ;
          ; Ende des Hauptprogramms, ab jetzt folgen Unterroutinen
          ;

text_out:           ; Füllt das Fenster mit seinem Inhalt

          ; Mauszeiger abschalten

          move.w    #123,contrl       ;v_hide_c
          clr.w     contrl+2
          clr.w     contrl+4
          clr.w     contrl+6
          clr.w     contrl+8
          move.w    handle,contrl+12
          jsr       vdi

          ; Arbeitsbereich berechnen (wind_calc):

          move.w    #108,control
          move.w    #6,control+2
          move.w    #5,control+4
          clr.w     control+6
          clr.w     control+8

          move.w    #1,int_in         ;1 -> Arbeitsbereich ausrechnen
          move.w    #63,int_in+2      ;Eigenschaften des Fensters
          move.w    wx,int_in+4       ;äußere Ausmaße
          move.w    wy,int_in+6
          move.w    ww,int_in+8
```

```
move.w    wh,int_in+10

jsr       aes

move.w    int_out+2,d0      ;x1 = x aus wind_calc
move.w    d0,x1
move.w    int_out+4,d1      ;y1 = y aus wind_calc
move.w    d1,y1
add.w     int_out+6,d0      ;x + Breite...
subq.w    #1,d0             ;... - 1...
move.w    d0,x2             ;... = x2
add.w     int_out+8,d1      ;y + Höhe...
subq.w    #1,d1             ;... - 1...
move.w    d1,y2             ;... = y2

; Clipping für diesen Arbeitsbereich einschalten

move.w    #129,contrl       ;Funktions-Opcode
move.w    #2,contrl+2
clr.w     contrl+4
move.w    #1,contrl+6
clr.w     contrl+8
move.w    handle,contrl+12

move.w    x1,ptsin          ;Koordinaten des Arbeitsbereichs
move.w    y1,ptsin+2
move.w    x2,ptsin+4
move.w    y2,ptsin+6
move.w    #1,intin          ;1: Einschalten

jsr       vdi

; vsf_interior (Füllmuster):

move.w    #23,contrl
clr.w     contrl+2
clr.w     contrl+4
move.w    #1,contrl+6
move.w    #1,contrl+8
move.w    handle,contrl+12
clr.w     intin             ;Füllen mit Hintergrundfarbe
jsr       vdi

; vsf_perimeter (Umrandung ausschalten):

move.w    #104,contrl
clr.w     contrl+2
clr.w     contrl+4
move.w    #1,contrl+6
move.w    #1,contrl+8
```

```
move.w    handle,contrl+12
clr.w     intin           ;Flag: Umrahmung ausschalten
jsr       vdi

; v_bar (löscht den Arbeitsbereich):
; (Koordinaten von wind_calc, müssen aber von x/y/breit/hoch
; in x1/y1/x2/y2 umgerechnet werden.

move.w    #11,contrl      ;Opcode für graphische
move.w    #2,contrl+2     ;Grundfunktionen
clr.w     contrl+4
clr.w     contrl+6
clr.w     contrl+8
move.w    #1,contrl+10    ;Funktionsnummer für v_bar
move.w    handle,contrl+12

move.w    x1,ptsin
move.w    y1,ptsin+2
move.w    x2,ptsin+4
move.w    y2,ptsin+6

jsr       vdi

; vsf_perimeter (Umrandung wieder einschalten):

move.w    #104,contrl
clr.w     contrl+2
clr.w     contrl+4
move.w    #1,contrl+6
move.w    #1,contrl+8
move.w    handle,contrl+12
move.w    #1,intin        ;Flag: Umrahmung einschalten
jsr       vdi

; Ausgabe der Texte mit v_gtext

move.w    #8,contrl
move.w    #1,contrl+2
clr.w     contrl+4        ;contrl+6 wird je nach Textlänge
clr.w     contrl+8        ;gesetzt
move.w    handle,contrl+12

move.w    x1,ptsin        ;x-Koordinate
addi.w    #8,ptsin

move.w    y1,ptsin+2      ;y-Koordinate
addi.w    #14,ptsin+2

lea       zeile1,a0
jsr       fix_text        ;Text in intin-Array schreiben
```

```
        jsr     vdi

        addi.w  #16,ptsin+2     ;nächste Zeile
        lea     zeile2,a0
        jsr     fix_text
        jsr     vdi

        addi.w  #16,ptsin+2
        lea     zeile3,a0
        jsr     fix_text
        jsr     vdi

        ; Clipping ausschalten

        move.w  #129,contrl     ;Funktions-Opcode
        move.w  #2,contrl+2
        clr.w   contrl+4
        move.w  #1,contrl+6
        clr.w   contrl+8
        move.w  handle,contrl+12
        clr.w   intin           ;0: Clipping aus
        jsr     vdi

        ; Mauszeiger wieder zeigen

        move.w  #122,contrl     ;v_show_c
        clr.w   contrl+2
        clr.w   contrl+4
        move.w  #1,contrl+6
        clr.w   contrl+8
        move.w  handle,contrl+12
        move.w  #1,intin
        jsr     vdi

        rts


fix_text:
        ; Unterprogramm, das einen String (Startadresse in a0
        ; zu übergeben) in das intin-Array schreibt, die Länge
        ; bestimmt und in contrl[3] ablegt

        clr.w   d0              ;Länge
        clr.w   d1              ;Hilfsregister
        lea     intin,a1

fix_loop: move.b  (a0)+,d1      ;ein Byte aus Zielstring holen
        tst.b   d1              ;Stringende?
        beq.s   fix_end
        move.w  d1,(a1)+        ;nein, dann als Wort ins
```

```
          addq.w    #1,d0           ;intin-Arrary schreiben
          bra.s     fix_loop

fix_end:  move.w    d0,contrl+6     ;Länge festhalten
          rts


          DATA

err_txt:  DC.b '[3][Sorry!¦Kein Window-Handle mehr frei!][OK]',0

wx:       DC.w 20  ;Anfangsgröße des Fensters
wy:       DC.w 20
ww:       DC.w 280
wh:       DC.w 150

title:    DC.b 'WIND2.PRG',0
info:     DC.b 'Bitte beachten Sie:',0

zeile1:   DC.b 'Hallo! Das ist eine Fenster-Demonstration!',0
zeile2:   DC.b 'Sie können das Fenster bewegen und seine '
          DC.b 'Größe verändern.',0
zeile3:   DC.b 'Ende durch Klick auf die Close-Box!',0

          BSS

whandle:  DS.w 1   ;Window-Handle

x1:       DS.w 1   ;Koordinaten des Arbeitsbereichs
y1:       DS.w 1
x2:       DS.w 1   ;Eckpunkt !!! Nicht Breite/Höhe!
y2:       DS.w 1

puffer:   DS.w 8   ;Hier werden die AES-Nachrichten abgelegt

          END
```

5.8.2 Mehrere Fenster

Wer ein Fenster auf den Bildschirm bringt, der kann auch mit zwei Fenstern hantieren - könnten Sie jetzt denken. Leider ergibt sich dabei ein Problem, das mit dem Redraw-Verfahren zusammenhängt. Im vorigen Programm wurde dieses Verfahren ja schon einmal angesprochen. Sie haben gesehen, daß in bestimmten Situationen der Fensterinhalt neu gezeichnet werden muß. Bis jetzt haben wir dazu ganz einfach die Routine aufgerufen, die den Arbeitsbereich löscht und den Text ins Fenster schreibt.

Wenn wir das gleiche jetzt mit zwei oder mehr Fenstern auf dem Bildschirm wieder machen würden, dann hätten wir damit zwar die Möglichkeit geschaffen, ein Fenster zu vergrößern (der neue, jetzt sichtbare Teil muß ja neu gezeichnet werden); die Kombination von mehreren Fenstern schafft aber ganz neue Situationen, um die wir uns bis jetzt nicht kümmern mußten. Damit meine ich folgendes: Wenn wir eine Redraw-Meldung im Nachrichtenpuffer erhalten, dann ist zunächst einmal unklar, ob ein ganzes Fenster neu gezeichnet werden soll (was wir bisher getan haben) oder nur ein Teil eines Fensters.

Dazu ein Beispiel: Sie haben zwei Fenster auf dem Bildschirm, wobei das eine (das im Vordergrund, also das aktive Fenster) die rechte Hälfte des anderen verdeckt. Nun schieben Sie dieses (das aktive) Fenster nach rechts, aber nur soweit, daß immer noch ein Teil des Hintergrund-Fensters verdeckt bleibt. Dann wird ein Stück des zweiten Fensters sichtbar, das vorher nicht zu sehen war. Dieses Stück muß das Programm neu zeichnen, aber... - Können Sie sich vorstellen, was passiert, wenn wir einfach das ganze Fenster neu ausgeben? Genau! Das Vordergrund-Fenster wird teilweise überschrieben; denn die VDI-Routinen, mit denen wir die Ausgabe ja vornehmen, kümmern sich nicht darum, ob sie im vorgeschriebenen Fenster bleiben oder nicht.

Wir müssen also irgendwie herausfinden, welche Teile eines neu zu zeichnenden Rechtecks überhaupt sichtbar sind. Sie können sich vorstellen, daß das bei mehr als zwei Fenstern recht schwierig zu programmieren ist. Es ist sogar unmöglich, sobald ein Accessory wie das Kontrollfeld sein "Privatfenster" geöffnet hat; denn die Applikation kann dessen Koordinaten nicht erfragen. (Abgesehen davon erfährt sie auch nichts vom Öffnen des Accessory-Fensters.) Statt dessen bedienen wir Programmierer uns der Rechteckliste.

Diese Liste kann von einem GEM-Programm für jedes seiner Fenster angefordert werden. Die Liste besteht dann aus einer möglichst kleinen Zahl von Rechtecken (bzw. deren Koordinaten), die zusammen den Bereich des Fensters abdecken, der im Moment sichtbar ist, d.h. nicht durch andere Fenster verdeckt wird. Die Zahl der Rechtecke variiert natürlich: Wenn ein großes Fenster ein kleines ganz verdeckt, dann gibt es für das kleine Fenster gar kein sichtbares Rechteck. Wenn aber das kleine Rechteck in den Vordergrund gebracht wird, dann liegt die Zahl der sichtbaren Rechtecke des großen Fensters bei vier (eines oben, eins unten, eins rechts und eins links neben dem kleinen Fenster, wenn das kleine Fenster in der Mitte des großen liegt)!

Nun, nehmen wir an, wir hätten diese Rechteckliste. Was ist dann bei einer Redraw-Meldung zu tun?

Im Nachrichtenpuffer finden wir zunächst einmal das Window-Handle des Fensters, das neu gezeichnet werden muß; falls mehrere Fenster auf Vordermann gebracht werden müssen, erhalten wir auch mehrere Redraw-Nachrichten. Dieses Handle steht im Wort 3. Außerdem liegen in den Worten 4 bis 7 die Koordinaten des Bereichs vor, in dem die Störung aufgetreten ist. Das ist also der Bereich, der maximal neu gezeichnet werden muß. Auch hierzu ein Beispiel: Fenster 2 (klein) liegt über Fenster 1 (groß). Wenn Fenster 2 geschlossen wird, dann ist Fenster 1 neu zu zeichnen, jedoch nur der Teil, der vorher von Fenster zwei verdeckt wurde (der gestörte Bereich halt).

Bevor wir diesen Bereich jedoch neu zeichnen, müssen wir zunächst die Rechteckliste des nunmehr vollständig sichtbaren und neu zu zeichnenden Fensters anfordern (es könnte ja sein, daß zwischen Fenster 1 und Fenster 2 noch ein drittes Fenster liegt). Nehmen wir an, wir erhalten zwei sichtbare Rechtecke. Dann müssen wir sehen, ob sich diese beiden Rechtecke mit dem neu zu zeichnenden Bereich (Koordinaten aus dem Nachrichten-Puffer) überlappen. Wir müssen quasi ein Schnittrechteck zwischen dem gestörten Bereich und jedem Rechteck aus der Rechteckliste bilden. Jedes Ergebnisrechteck (falls es das gibt) ist dann ein wirklich neu zu zeichnende Bereich, den wir mit VDI-Funktionen bearbeiten können (am besten: Clipping und dann zeichnen).

Soweit zur Theorie. Die wichtigsten Fragen, die Sie jetzt stellen sollten, sind: Wie komme ich an die Rechteckliste, und wie bilde ich ein Schnittrechteck? Nun, die Rechteckliste wird vom AES automatisch geführt und bei jeder Bewegung von Fenstern auf den neuesten Stand gebracht. Bevor Sie die Liste anfordern können, muß sie freigegeben werden. Das geschieht mit der AES-Funktion wind_update. Sie erhält einen Parameter, der ein Wert zwischen 0 und 3 sein darf. Für uns sind die Werte 0 und 1 interessant: Mit einer Eins teilen wir dem AES mit, daß wir im folgenden einen Fensterbereich neu aufbauen wollen und dazu die Rechteckliste benötigen werden. Nebeneffekt: GEM läßt keine Benutzeraktivitäten mit der Maus mehr zu. Ihr Programm kann also in Ruhe ein Fenster aufbauen, ohne daß der Anwender Größe oder Position dieses oder eines anderen Fensters ändern könnte. Auch die Drop-Down-Menüs (mehr später!) sind gesperrt. Erst wenn wir fertig sind und der Benutzer wieder handeln darf, heben wir die Sperre durch einen wind_update-Aufruf mit dem Parameter Null wieder auf.

Die Rechteckliste selbst kann dann mit der Funktion wind_get angefordert werden. Sie ist das Gegenstück zu wind_set und erlaubt es, die verschiedensten Daten über ein Fenster, dessen Handle zu übergeben ist, in Erfahrung zu bringen. Zu diesen verschiedensten Daten gehört nun auch die Rechteckliste. Bei jedem Aufruf von wind_get wird je ein sichtbares Rechteck zurückgegeben. Wichtig ist auch, daß es ein Erstes Rechteck und dann nur noch das jeweils Nächste Rechteck gibt. Dafür stehen zwei verschiedene Unterfunktions-Nummern von wind_get zur Verfügung: 11 für das erste und 12 für das nächste Rechteck. Gewöhnlich wird also in einer Schleife solange nach Rechtecken gefragt, bis das AES das Ende der Liste meldet. Die Liste gilt dann als beendet, wenn Breite und Höhe des zuletzt zurückgegebenen Rechtecks Null sind. Um dies im Programm abzufragen, genügt es natürlich, eine der beiden Größen zu überprüfen. (Sie erinnern sich: Das kleinste Rechteck, das es auf dem Computer geben kann, hat die Breite 1 und die Höhe 1; besser bekannt als Punkt.)

So bleibt die Frage offen, wie wir das Schnittrechteck aus einem Rechteck der Liste und derm zerstörten Bereich errechnen. Dafür gibt es keine GEM-Funktion; das müssen wir also von Hand machen. Doch keine Sorge, so schwer ist es nicht: Wenn die Koordinaten der Rechtecke im VDI-Format (x1/y1/x2/y2) vorliegen, dann ist x1 des Schnittrechtecks die größere der beiden x1-Koordinaten der beiden sich schneidenden Bereiche; x2 des Ergebnisrechtecks ist dagegen der kleinere der beiden x2-Werte der Quellbereiche. Wenn am Ende noch x2 größer als x1 ist, dann überschneiden sich die Rechtecke, sonst nicht. Für die y-Werte gilt Entsprechendes. Im Programm sieht das meist etwas anders aus, weil in der Regel Koordinaten im AES-Format vorliegen (x/y/w/h).

In meinem Beispielprogramm verwende ich eine Mischung: Die Quellrechtecke werden im AES-Format angegeben, und das Unterprogramm gibt VDI-Koordinaten zurück. Das hat sich als besonders praktisch erwiesen, weil das Schnittrechteck meist mit VDI-Routinen weiterbehandelt wird. Übrigens: Ein solches Unterprogramm bekommt üblicherweise den Namen intersect. Manche C-Compiler haben eine Intersect-Funktion in ihrer Standard-Bibliothek, doch werden dort spezielle Datenstrukturen verwendet, die ich hier nicht behandeln möchte, weil sie sich nicht auf die anderen Sprachen übertragen lassen. Auch in C programmieren wir die Funktion also selbst. In GFA-BASIC ist die Funktion dagegeben brauchbar implementiert, so daß wir sie auch benutzen werden.

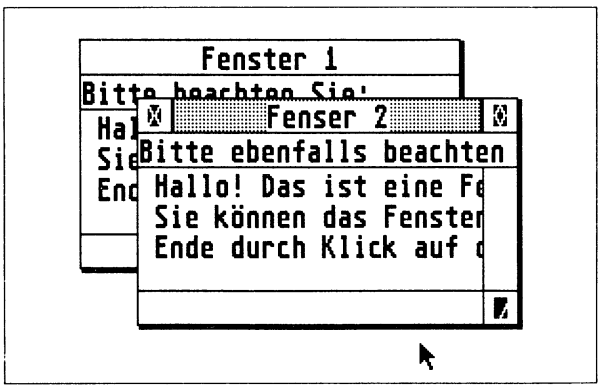
Einige weitere Deteils sollten Sie ab sofort beachten: Selbstverständlich muß ein Programm jetzt mit Hilfe der Window-Handles sehr genau zwischen den verschiedenen Fenstern unterscheiden. Dabei ist auch zu be-

achten, daß auch ein Accessory, das ein Fenster geöffnet hat, Nachrichten empfangen muß. Diese Nachrichten erhält aber möglicherweise nicht nur das Accessory; unser eigenes Programm könnte sie "mithören". Jedes GEM-Programm sollte also als erstes prüfen, ob die Nachricht überhaupt für eines der eigenen Fenster bestimmt ist. In diesem Zusammenhang sollte auch erwähnt werden, daß Sie zwar Daten (also Handles, Koordinaten usw.) mehrerer Fenster in Arrays unterbringen können (so habe ich es auch gemacht), doch müssen Sie dann höllisch aufpassen, daß Sie nicht Window-Handles und Indizes verwechseln (eine wunderbare Fehler- und damit Bombenquelle)!

Und noch etwas ist wichtig: Sobald es mehrere Fenster gibt, kann der Benutzer durch Anklicken eines Fensters im Hintergrund dieses Fenster in den Vordergrund bringen, also als aktives Fenster deklarieren. Das ist so nicht ganz richtig; denn Sie erhalten im Nachrichtenpuffer lediglich die Mitteilung, daß der Benutzer ein Fenster aktivieren möchte (Ereignis-Nr. 21, WM_TOPPED). Das Fenster müssen Sie (bzw. Ihr Programm) dann schon selbst nach vorne bringen. Das geht mit der wind_set-Funktion, Unterfunktion Nr. 10. Eins noch am Rande: Wenn sowohl Accessories als auch die Applikation Fenster geöffnet haben, dann werden Tastendrücke nur dem Programm gemeldet, dessen Fenster oben liegt, also aktiv ist. Das ist ganz praktisch, weil dann die Ziffern, die Sie für das Taschenrechner-Accessory eintippen, nicht gleich mit in die Textverarbeitung übernommen werden.

Die letzte Neuerung betrifft die maximalen Koordinaten eines Fensters. Bisher haben wir die recht umständlich über die Bildschirmauflösung hergeleitet. Es geht aber sauberer: Mit der Funktion wind_get können wir nicht nur Daten (z.B. Größe) unserer eigenen Fenster, sondern auch die Werte des Desktops erfragen. Desktop ist in diesem Fall der gesamte Bildschirm bis auf die Menüzeile. Das Desktop-Fenster hat ein festes Window-Handle, nämlich 0. Die Koordinaten des Arbeitsbereichs erfragen wir dann mit der Unterfunktion Nr. 4 von wind_get.

So, jetzt haben Sie alle Kenntnisse, um das folgende Programm zu verstehen. Lassen Sie es erst einmal laufen, und probieren Sie alles aus, bevor Sie sich das Listing vornehmen. Sie werden feststellen, daß die "Prögrämmchen" mittlerweile etwas umfangreicher ausfallen, gerade in Assembler. Das folgende Programm ist aber eine Ausnahme; die nächsten Programme werden wieder kürzer!



GFA-BASIC

```
'
' Zwei Fenster gleichzeitig mit Funktionen
' GFA-BASIC      MP 13-12-88       WIND3.GFA
'
DEFINT "a-z"    ! alle Variablen sind Integer -> erspart uns viele %s
'
DIM puffer&(7)  ! Platz für 8 Worte einer Nachricht (evnt_mesag)
DIM whandle(1),wx(1),wy(1),ww(1),wh(1),opened(1)
'
' Initialisierung der Arrays:
'
wx(0)=20    ! Koordinaten Fenster 0
wy(0)=20
ww(0)=200
wh(0)=120
'
wx(1)=50    ! Koordinaten Fenster 1
wy(1)=50
ww(1)=200
wh(1)=120
'
VOID APPL_INIT()
'
' Größe des Desktop (Abeitsbereich) feststellen:
'
VOID WIND_GET(0,4,x_desk,y_desk,w_desk,h_desk)
'
' Fenster anmelden für maximale Größe und allen Elementen (63)' außer Pfeile und
Schieber
'
IF FN create_windows
  VOID FORM_ALERT(1,"[3][Kein Handle mehr verfügbar!][OK]")
```

```
ELSE
  '
  ' Titel- und Infozeile einstellen:
  ' Die Strings werden in einen sicheren Speicherbereich
  ' kopiert, da sich ihre Startadresse nicht ändern darf
  '
  t1$="Fenster 1"+CHR$(0)
  i1$="Bitte beachten Sie:"+CHR$(0)
  t2$="Fenser 2"+CHR$(0)
  i2$="Bitte ebenfalls beachten:"+CHR$(0)
  '
  s=MALLOC(100)
  '
  FOR i=1 TO LEN(t1$)                ! Titelzeile 1
    POKE s-1+i,ASC(MID$(t1$,i,1))
  NEXT i
  '
  FOR i=1 TO LEN(i1$)                ! Infozeile 1
    POKE s+24+i,ASC(MID$(i1$,i,1))
  NEXT i
  '
  FOR i=1 TO LEN(t2$)                ! Titelzeile 2
    POKE s+49+i,ASC(MID$(t2$,i,1))
  NEXT i
  '
  FOR i=1 TO LEN(i2$)                ! Infozeile 2
    POKE s+74+i,ASC(MID$(i2$,i,1))
  NEXT i
  '
  ' Wir brauchen Low- und High-Words (durch DIV 2^16und MOD 2^16)
  ' Titelzeile: Unterfunktion 2  /  Infozeile: Unterfunktion 3  '
  VOID WIND_SET(whandle(0),2,s DIV 65536,s MOD 65536,0,0)
  VOID WIND_SET(whandle(0),3,(s+25) DIV 65536,(s+25) MOD 65536,0,0)
  VOID WIND_SET(whandle(1),2,(s+50) DIV 65536,(s+50) MOD 65536,0,0)
  VOID WIND_SET(whandle(1),3,(s+75) DIV 65536,(s+75) MOD 65536,0,0)
  '
  ' Fenster öffnen:
  '
  FOR i=0 TO 1
    VOID WIND_OPEN(whandle(i),wx(i),wy(i),ww(i),wh(i))
    opened(i)=1
    output_text(i)
  NEXT i
  '
  REPEAT
    '
    ' Auf ein Ereignis warten:
    '
    VOID EVNT_MESAG(LPEEK(*puffer&())+4) ! Adresse des ersten Elements
    '
```

```
    ' Index des Fensters ermitteln, von dem die Nachricht stammt:
    ' (Window-Handle steht in puffer&(3))
    '
    w=0
    WHILE NOT (puffer&(3)=whandle(w) OR w=2)
      INC w
    WEND
    '
    IF window<2           ! War das überhaupt unser Fenster (1 oder 2)?
      IF puffer&(0)=20    ! Redraw-Meldung
        GOSUB redraw(w)
      ENDIF
      '
      IF puffer&(0)=21        ! Window-Topped
        VOID WIND_SET(whandle(w),10,0,0,0,0)
      ENDIF
      '
      IF puffer&(0)=23        ! Full-Box wurde angeklickt
        wx(w)=x_desk+2        ! Fenster auf maximal mögliche
        wy(w)=y_desk+2        ! Größe bringen
        ww(w)=w_desk-6
        wh(w)=h_desk-6
        VOID WIND_SET(whandle(w),5,wx(w),wy(w),ww(w),wh(w))
      ENDIF
      '
      IF puffer&(0)=27        ! Size-Box
        ww(w)=puffer&(6)      ! Größe aus der Nachricht entnehmen
        wh(w)=puffer&(7)
        VOID WIND_SET(whandle(w),5,wx(w),wy(w),ww(w),wh(w))
      ENDIF
      '
      IF puffer&(0)=28        ! Move-Box
        wx(w)=puffer&(4)      ! x und y sind verändert
        wy(w)=puffer&(5)
        VOID WIND_SET(whandle(w),5,wx(w),wy(w),ww(w),wh(w))
      ENDIF
      '
      IF puffer&(0)=22        ! Close-Box
        VOID WIND_CLOSE(whandle(w))
        VOID WIND_DELETE(whandle(w))
        opened(w)=0
      ENDIF
      '
    ENDIF
  UNTIL NOT ((opened(0)=1) OR (opened(1)=1)) ! solange Fenster offen...
  '
  VOID MFREE(s)
ENDIF
'
VOID APPL_EXIT()
```

```
'
END
'
'
PROCEDURE output_text(w)    !    Fensterinhalt am Programmstart
  '                                                ausgeben
  ' Größe des Arbeitsbereichs ermitteln:
  '
  VOID WIND_CALC(1,63,wx(w),wy(w),ww(w),wh(w),x,y,w,h)
  x2=x+w-1
  y2=y+h-1
  CLIP x,y TO x2,y2
  draw_text(x,y,x2,y2,x,y)
  CLIP OFF
RETURN
'
'
PROCEDURE draw_text(x1,y1,x2,y2,x,y)
  '
  ' Parameter:
  '   x1,y1,x2,y1 ist das Rechteck, das neu gezeichnet, also
  '   gelöscht werden soll. x,y ist die linke obere Ecke des
  '   Textes. x,y sind nicht unbedingt gleich x1,y1 !!!
  '
  HIDEM    ! Maus abschalten, die stört nur
  '
  ' Löschen des Arbeitsbereichs:
  '
  BOUNDARY 0
  DEFFILL 0
  PBOX x1,y1,x2,y2
  BOUNDARY 1
  '
  ' Text ausgeben:
  '
  TEXT x+8,y+14,"Hallo! Das ist eine Fenster-Demonstration!"
  TEXT x+8,y+30,"Sie können das Fenster bewegen und seine Größe...
... verändern."
  TEXT x+8,y+46,"Ende durch Klick auf die Close-Box!"
  '
  SHOWM
RETURN
'
'
FUNCTION create_windows
  FOR i=0 TO 1
    whandle(i)=WIND_CREATE(63,x_desk,y_desk,w_desk,h_desk)
  NEXT i
  RETURN (whandle(0)<0) AND (whandle(1)<0)
ENDFUNC
```

```
'
'
PROCEDURE redraw(w)
  '
  ' Nochmal den Arbeitsbereich berechnen:
  ' (brauchen wir für linke obere Ecke des Textes)
  '
  VOID WIND_CALC(1,63,wx(w),wy(w),ww(w),wh(w),ax,ay,aw,ah)
  '
  VOID WIND_UPDATE(1)   ! Alle Mausfunktionen sperren,
  '             damit wir ungestört sind / Rechteckliste freigeben  '
  ' Verarbeitung der Rechteckliste:
  '
  VOID WIND_GET(whandle(w),11,rx,ry,rw,rh)  ! 11->erstes Rechteck
  '
  WHILE rw>0    ! solange, wie das Rechteck noch Breite besitzt
    IF RC_INTERSECT(puffer&(4),puffer&(5),puffer&(6),puffer&(7),...
                            ...rx,ry,rw,rh)
      rx2=rx+rw-1
      ry2=ry+rh-1
      CLIP rx,ry TO rx2,ry2
      GOSUB draw_text(rx,ry,rx2,ry2,ax,ay)
    ENDIF
    '
    VOID WIND_GET(whandle(w),12,rx,ry,rw,rh) ! 12 -> nächstes Rechteck
  WEND
  '
  CLIP OFF
  VOID WIND_UPDATE(0)   ! Maus wieder anwerfen
RETURN
'
```

Omikron-BASIC

```
'
' Zwei Fenster gleichzeitig mit Funktionen
' Omikron-BASIC   MP 04-12-88    WIND3.BAS
'
DIM Whandle(1),Wx(1),Wy(1),Ww(1),Wh(1),Opened(1)
Appl_Init:V_Opnvwk
'
' Größe des Desktop-Windows (Nummer 0) erfragen:
'
Wind_Get(0,4,Xdesk,Ydesk,Wdesk,Hdesk)
'
' Fenster anmelden:
IF FN Create_Windows THEN
  FORM_ALERT (1,"[3][Sorry!¦Kein Window-Handle mehr frei!][OK]",Dummy)
ELSE
  '
```

```
' Standardgröße der Fenster in Array eintragen:
'
Wx(0)=20:Wy(0)=20:Ww(0)=200:Wh(0)=120
Wx(1)=50:Wy(1)=50:Ww(1)=200:Wh(1)=120
'
' Titel- und Infozeile einstellen:
'
Adr= MEMORY(100)' Speicherplatz für wind_set
'
Wind_Set(Whandle(0),2,"Fenster 1",Adr)
Wind_Set(Whandle(0),3,"Bitte beachten Sie:",Adr+25)
Wind_Set(Whandle(1),2,"Fenster 2",Adr+50)
Wind_Set(Whandle(1),3,"Bitte ebenfalls beachten:",Adr+75)
'
' Fenster öffnen und etwas reinschreiben:
'
FOR I=0 TO 1
  Wind_Open(Whandle(I),Wx(I),Wy(I),Ww(I),Wh(I))
  Opened(I)=1
  Output_Text(I)
NEXT I
'
' *** Warteschleife ***
'
REPEAT
  Evnt_Mesag(Puffer$)
  '
  ' Feststellen, von welchem Fenster die Nachricht kam (0 oder 1)
  ' Handle steht in Puffer[3]
  '
  Window=0
  WHILE NOT (FN P(3)=Whandle(Window) OR Window=2)
    Window=Window+1
  WEND
  '
  IF Window<2 THEN '    nur eigene Fenster (0 und 1) beachten
    IF FN P(0)=20 THEN Redraw(Window)
    '
    IF FN P(0)=21 THEN '    Window Topped
      Wind_Set(Whandle(Window))' Fenster nach oben bringen
    ENDIF
    '
    IF FN P(0)=23 THEN '    Full-Box
      Wx(Window)=Xdesk+2
      Wy(Window)=Ydesk+2
      Ww(Window)=Wdesk-6
      Wh(Window)=Hdesk-6
      Wind_Set(Whandle(Window),Wx(Window),Wy(Window),...
                      ...Ww(Window),Wh(Window))
    ENDIF
```

```
      '
      IF FN P(0)=27 THEN '    Size-Box
        Ww(Window)=FN P(6)
        Wh(Window)=FN P(7)
        Wind_Set(Whandle(Window),Wx(Window),Wy(Window),...
                        ...Ww(Window),Wh(Window))
      ENDIF
      '
      IF FN P(0)=28 THEN '    Move-Bar
        Wx(Window)=FN P(4)
        Wy(Window)=FN P(5)
        Wind_Set(Whandle(Window),Wx(Window),Wy(Window),...
                        ...Ww(Window),Wh(Window))
      ENDIF
      '
      IF FN P(0)=22 THEN '    Close-Box
        Wind_Close(Whandle(Window))
        Wind_Delete(Whandle(Window))
        Opened(Window)=0'    als geschlossen markieren
      ENDIF
    '
    ENDIF
    '
  ' Weiter, solange noch Fenster offen sind...
  UNTIL NOT ((Opened(0)=1) OR (Opened(1)=1))
  '
ENDIF
'
V_Clsvwk:Appl_Exit
END
'
'
DEF FN P(N)= CVI( MID$(Puffer$,N*2+1,2))
'
'
DEF FN Create_Windows
  FOR I=0 TO 1
    Wind_Create(63,Xdesk,Ydesk,Wdesk,Hdesk,Whandle(I))  NEXT I
  '
RETURN (Whandle(0)<0) OR (Whandle(1)<0)
'
'
DEF PROC Intersect(X1,Y1,W1,H1,X2,Y2,W2,H2,R X,...
                            ...R Y,R Xx,R Yy,R Ret)
  ' Berechnet die Schnittfläche zweier Rechtecke
  W= MIN(X2+W2,X1+W1)
  H= MIN(Y2+H2,Y1+H1)
  X= MAX(X2,X1)
  Y= MAX(Y2,Y1)
  Xx=W-1:Yy=H-1
```

```
  '
  Ret=(W>X) AND (H>Y)' Prüft, ob Schnittfläche vorhanden ist
RETURN
'
'
DEF PROC Draw_Text(X1,Y1,X2,Y2,X,Y)
  ' Bedeutung der Parameter:
  ' X1, Y2, X2 und Y2 bilden das Rechteck, das zu löschen ist.
  ' x und y geben an, wo die linke obere Ecke des Textes sein soll.
  ' Das muß nicht innerhalb des Rechtecks sein, wenn z.B. bei
  ' Redraw nur die untere Hälfte des Fensters zu löschen ist.
  '
  V_Hide_C
  '
  Vsf_Interior(0)'   Rechteck löschen
  Vsf_Perimeter(0)
  V_Bar(X1,Y1,X2,Y2)
  Vsf_Perimeter(1)
  '
  V_Gtext(X+8,Y+14,"Hallo! Das ist eine Fenster-Demonstration!")
  V_Gtext(X+8,Y+30,"Sie können das Fenster bewegen und seine...
                          ... Größe verändern.")
  V_Gtext(X+8,Y+46,"Ende durch Klick auf die Close-Box!")
  '
  V_Show_C(1)
RETURN
'
'
DEF PROC Output_Text(Window)
  Wind_Calc(1,63,Wx(Window),Wy(Window),Ww(Window),...
                  ...Wh(Window),X,Y,W,H)
  X2=X+W-1:Y2=Y+H-1
  Vs_Clip(X,Y,X2,Y2)
  Draw_Text(X,Y,X2,Y2,X,Y)
  Vs_Clip
RETURN
'
'
DEF PROC Redraw(Window)
  '
  ' Wir brauchen nochmal den Arbeitsbereich:
  '
  Wind_Calc(1,63,Wx(Window),Wy(Window),Ww(Window),...
                   ...Wh(Window),Ax,Ay,Aw,Ah)
  '
  Wind_Update(1)'  Mausfunktionen sperren
  '
  ' Verarbeitung der Rechteckliste:
  '
  Wind_Get(Whandle(Window),11,Rx,Ry,Rw,Rh)
```

```
  '
  WHILE Rw<>0'   Aufhören wenn Breite Null
    Intersect(FN P(4),FN P(5),FN P(6),FN P(7),Rx,Ry,...
                 ...Rw,Rh,X,Y,Xx,Yy,Ret)
    IF Ret THEN '   Falls Überschneidung
      Vs_Clip(X,Y,Xx,Yy)
      Draw_Text(X,Y,Xx,Yy,Ax,Ay)
    ENDIF
    '
    Wind_Get(Whandle(Window),12,Rx,Ry,Rw,Rh)' nächst. Recht.
  WEND
  '
  Vs_Clip'   Clipping ausschalten
  Wind_Update(0)
RETURN
'
```

Denken Sie wieder daran, durch Punkte geteilte Zeilen in einer Zeile einzugeben!

C

```
/************************************************/
/*   Zwei Fenster gleichzeitig mit Funktionen   */
/*   Laser C        MP  30-11-8       WIND3.C   */
/************************************************/

#include <osbind.h>
#include "gem_inex.c"

int  whandle[2],                          /* Alles für zwei Fenster */
     opened[2] = {1, 1};                 /* Flag ob Fenster geöffnet */

int  puffer[8];                /* Hierhin kommen die AES-Nachrichten */

int  wx[2] = {20, 50},       /* äußere Fenstergrößen mit Startwerten */
     wy[2] = {20, 50},
     ww[2] = {200, 200},
     wh[2] = {120, 120};

int  xdesk, ydesk, wdesk, hdesk;              /* Größe des Desktop */

int  i, window;                                  /* Hilfsvariablen */


void clip_window (pxyarray)     /* Begrenzt alle nachfolgenden VDI- */
     int pxyarray[];                  /* Ausgaben auf einen Bereich */
{
```

```
  vs_clip (handle, 1, pxyarray);        /* 1: Clipping einschalten */
}


void switch_clipping_off()          /* macht clip_window rückgängig */
{
int  pxyarray[4];

  vs_clip (handle, 0, pxyarray);        /* 0: Clipping ausschalten */
}


void output_text(window)
     int window;
{
int  pxyarray[4],
     x,y,w,h;

  /* Arbeitsbereich errechnen: */

  wind_calc (1, 63, wx[window], wy[window], ww[window],
             wh[window], &x, &y, &w, &h);

  /* Umrechnen von Höhe/Breite in zweiten Eckpunkt: (x2/y2) */
   pxyarray[0] = x;
   pxyarray[1] = y;
   pxyarray[2] = x+w-1;
   pxyarray[3] = y+h-1;

  clip_window (pxyarray);
    draw_text (pxyarray, x, y);
    switch_clipping_off();
}

draw_text (pxyarray, x, y)
int pxyarray[], x, y;
{
  /* Mauszeiger ausschalten: */

  v_hide_c (handle);

  /* Arbeitsbereich löschen: */

  vsf_interior (handle, 0);   /* Füllen mit Hintergrundfarbe */
  vsf_perimeter (handle, 0);  /* keine Umrandung */
  v_bar (handle, pxyarray);   /* weißes gefülltes Rechteck */
  vsf_perimeter (handle, 1);  /* Umrandung wieder einschalten */
```

```
  /* Jetzt schreiben wir irgendeinen Text in das Fenster. */

  v_gtext (handle, x + 8, y + 14,
        "Hallo! Das ist eine Fenster-Demonstration!");
  v_gtext (handle, x + 8, y + 30,
        "Sie können das Fenster bewegen und seine Größe verändern.");
  v_gtext (handle, x + 8, y + 46,
        "Ende durch Klick auf die Close-Box!");

  /* Mauszeiger wieder zeigen: */

  v_show_c (handle, 1);
}


int create_windows()
{
  /* Desktop-Größe ermitteln mit wind_get Funktion Nr. 4 */
  /* Desktop hat festes Window-Handle 0 */

  wind_get (0, 4, &xdesk, &ydesk, &wdesk, &hdesk);

  for (i = 0;  i <= 1;  i++)
    whandle[i] = wind_create (63, xdesk, ydesk, wdesk, hdesk);
  return (whandle[0] < 0 || whandle[1] < 0);
}


int intersect (x1, y1, w1, h1, x2, y2, w2, h2, pxyarray)
    int x1, y1, w1, h1, x2, y2, w2, h2, pxyarray[];
{
int x, y, w, h;

  w = (x2+w2 < x1+w1) ? x2+w2 : x1+w1;
  h = (y2+h2 < y1+h1) ? y2+h2 : y1+h1;
  x = (x2 > x1) ? x2 : x1;
  y = (y2 > y1) ? y2 : y1;

  pxyarray[0] = x;
  pxyarray[1] = y;
  pxyarray[2] = w - 1;
  pxyarray[3] = h - 1;

  return (w > x && h > y);
}


void redraw (window)
     int window;
{
```

```
int  x, y, w, h,        /* Neu zu zeichnender Bereich */
     rx, ry, rw, rh,    /* Variablen für die Rechteckliste */
     ax, ay, aw, ah,    /* Arbeitsbereich */
     pxyarray[4];

  wind_update (1);   /* Maus desaktivieren, Rechteckliste freigeben */

  x = puffer[4];     /* Koordinaten des neu zu zeichnenden Bereichs */
  y = puffer[5];     /* (gehört zur Nachricht) */
  w = puffer[6];
  h = puffer[7];

  /* Größe Arbeitsbereich ermitteln */

  wind_calc (1, 63, wx[window], wy[window], ww[window],
       wh[window], &ax, &ay, &aw, &ah);

  /* Erstes Rechteck aus der Rechteckliste anfordern: (11) */

  wind_get (whandle[window], 11, &rx, &ry, &rw, &rh);

  while (rw != 0)    /* Solange Breite > 0... */
  {
    if (intersect (x, y, w, h, rx, ry, rw, rh, pxyarray))
    {
      clip_window (pxyarray);
      draw_text (pxyarray, ax, ay);
      /* pxyarray gibt das zu zeichnende Rechteck an, während */
      /* x und y die gedachte obere linke Ecke angeben */
      /* (muß nicht im Rechteck liegen */
    }
  wind_get (whandle[window], 12, &rx, &ry, &rw, &rh);  /* nächstes R. */
  }

  switch_clipping_off();
  wind_update (0);   /* Wir sind fertig. */
}


main()
{
  gem_init();

  graf_mouse (0, 0L);   /* Pfeil als Mauszeiger */

  if (create_windows())  /* Funktion liefert 1 bei Fehler */
    form_alert (1, "[3][Sorry!|Kein Window-Handle mehr frei!][OK]");
  else
  {
```

```
/* Titel- und Infozeile einstellen */

wind_set (whandle[0], 2, "Fenster 1", 0, 0);
wind_set (whandle[1], 2, "Fenster 2", 0, 0);

wind_set (whandle[0], 3, "Bitte beachten Sie:", 0, 0);
wind_set (whandle[1], 3, "Bitte ebenfalls beachten:", 0, 0);

/* Fenster öffnen und etwas reinschreiben */

for (i = 0;  i <= 1;  i++)
{
  wind_open (whandle[i], wx[i], wy[i], ww[i], wh[i]);
  output_text (i);
}

/*************************************************************/
/*                Jetzt kommt das Wichtigste:                */
/*   Wir warten auf Benutzeraktivitäten und werten sie aus   */
/*************************************************************/

do
{
  evnt_mesag (puffer);     /* Warten auf eine Nachricht */
  /* Index des Fensters ermitteln, von dem die Nachricht stammt */
  /* puffer[3] enthält dazu das Window-Handle */

  for (window = 0;
        !(puffer[3] == whandle[window] || window == 2);  window++);

  if (window != 2)   /* Handle gefunden bzw. unser Fenster? */      {
    switch (puffer[0])
    {
      case 20:              /* Redraw */
        redraw (window);
        break;

      case 21:              /* Topped */
        wind_set(whandle[window], 10, 0, 0, 0, 0);  /* 10 -> Top */
        break;

      case 23:              /* Full-Box */
        wx[window] = xdesk + 2;
        wy[window] = ydesk + 2;
        ww[window] = wdesk - 6;
        wh[window] = hdesk - 6;
        wind_set (whandle[window], 5, wx[window],
             wy[window], ww[window], wh[window]); /* neue Größe */
        break;
```

```
        case 27:                /* Size-Box */
          ww[window] = puffer[6];
          wh[window] = puffer[7];
          wind_set (whandle[window], 5, wx[window],
              wy[window], ww[window], wh[window]); /* neue Größe */
          break;

        case 28:                /* Move-Bar */
          wx[window] = puffer[4];
          wy[window] = puffer[5];
          wind_set (whandle[window], 5, wx[window],
              wy[window], ww[window], wh[window]); /* neue Größe */
          break;

        case 22:                /* Close-Box */
          wind_close (whandle[window]);
          wind_delete (whandle[window]);
          opened[window] = 0;   /* als geschlossen kennzeichnen */
          break;
      }
    }
  }  while (opened[0] || opened[1]);  /* Bis beide Fenster zu sind */
 }
 gem_exit();
}
```

Assembler

```
        ;
        ; Zwei Fenster gleichzeitig mit Funktionen
        ; Assembler     MP  05-12-88       WIND3.Q
        ;

        INCLUDE 'GEM_INEX.Q'

gemdos  = 1

        TEXT

main:   jsr       gem_init

        ; 'Anständiger' Mauszeiger:

        move.w    #78,control       ;graf_mouse
        move.w    #1,control+2
        move.w    #1,control+4
        move.w    #1,control+6
        clr.w     control+8
        clr.w     int_in            ;0 für Pfeil
```

```
        jsr     aes

        clr.w   d1              ;Größe des Desktop (0) ermitteln
        move.w  #4,d2           ;Unterfunktion
        jsr     wind_get
        move.w  d1,xdesk
        move.w  d2,ydesk
        move.w  d3,wdesk
        move.w  d4,hdesk

        jsr     create_windows
        tst.w   d0              ;Fehler?
        beq.s   no_err

        move.w  #52,control     ;dann Form-Alert
        move.w  #1,control+2    ;für Fehlermeldung
        move.w  #1,control+4
        move.w  #1,control+6
        clr.w   control+8
        move.l  #err_txt,addr_in ;String, der die Box beschreibt
        move.w  #1,int_in       ;erster Knop ist Default-Button
        jsr     aes
        bra     quit

no_err: lea     title1,a0       ;Titelzeile
        move.w  #2,d0           ;2 ist wind_set Unterfunktion
        move.w  whandle,d1      ;erstes Fenster
        jsr     wind_set1

        lea     info1,a0
        move.w  #3,d0           ;3 setzt Info-Zeile
        move.w  whandle,d1
        jsr     wind_set1

        lea     title2,a0
        move.w  #2,d0
        move.w  whandle+2,d1    ;zweites Fenster
        jsr     wind_set1

        lea     info2,a0
        move.w  #3,d0
        move.w  whandle+2,d1
        jsr     wind_set1

        ; Öffnen der Windows:

        clr.w   d7
        clr.w   d3
        lea     whandle,a5
```

```
          lea       wx,a4

open_lp:  move.w    #101,control      ;wind_open
          move.w    #5,control+2
          move.w    #1,control+4
          clr.w     control+6
          clr.w     control+8
          move.w    0(a5,d7.w),int_in   ;Window-Handle
          move.w    0(a4,d7.w),int_in+2
          move.w    4(a4,d7.w),int_in+4
          move.w    8(a4,d7.w),int_in+6
          move.w    12(a4,d7.w),int_in+8
          jsr       aes
          jsr       output_text
          addq.w    #1,d3             ;Index für output_text
          addq.w    #2,d7             ;nächstes Fenster
          cmpi.w    #4,d7             ;schon beide offen?
          bne.s     open_lp           ;nein, dann Schleife

          ;
          ; *** Nachrichtenschleife ***
          ;

          ; evnt_mesag:

loop:     move.w    #23,control
          clr.w     control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          move.l    #puffer,addr_in
          jsr       aes

          ; Index des Fensters ermitteln, das die Nachricht auslöste
          ; und dessen Handle in puffer+6 steht:
          ; Ergebnis ist eigentlich kein Index, sondern ein Offset.

          clr.w     d3
          lea       wx,a4             ;Koordinaten
          lea       whandle,a5        ;Handles
          lea       puffer,a6         ;Nachricht
          move.w    6(a6),d2          ;Window-Handle der 'Quelle'

s_lp:     cmp.w     0(a5,d3.w),d2
          beq.s     s_found
          addq.w    #2,d3
          cmpi.w    #4,d3             ;ist das überhaupt unser Fenster?
          beq       loop              ;scheint nicht so, weiter warten
          bra.s     s_lp
```

```
        ; Jetzt werden die möglichen Ereignisse abgefragt/behandelt:

s_found: cmpi.w   #20,(a6)        ;Redraw?
         bne.s    lab0
         jsr      redraw          ;dann Bereich neuzeichnen
         bra      cont

lab0:    cmpi.w   #21,(a6)        ;Window Topped
         bne.s    lab1
         move.w   d2,d1           ;Window-Handle
         move.w   #10,d0          ;Funktion: Window nach oben bringen
         jsr      wind_set2
         bra      cont

lab1:    cmpi.w   #22,(a6)        ;Close-Box
         bne.s    lab2
         move.w   #102,control    ;wind_close
         move.w   #1,control+2
         move.w   #1,control+4
         clr.w    control+6
         clr.w    control+8
         move.w   0(a5,d3.w),int_in   ;Handle
         jsr      aes
         move.w   #103,control    ;wind_delete
         jsr      aes             ;(nur neue Funktionsnummer)
         lea      opened,a3       ;Fenster nicht mehr geöffnet
         clr.w    0(a3,d3.w)
         bra      cont

lab2:    cmpi.w   #23,(a6)        ;23: Full-Box
         bne.s    lab3
         move.w   xdesk,d0        ;Maximale Fenstergröße einstellen
         addq.w   #2,d0
         move.w   d0,0(a4,d3.w)
         move.w   ydesk,d0
         addq.w   #2,d0
         move.w   d0,4(a4,d3.w)
         move.w   wdesk,d0
         subq.w   #6,d0
         move.w   d0,8(a4,d3.w)
         move.w   hdesk,d0
         subi.w   #6,d0
         move.w   d0,12(a4,d3.w)
         bra      set

lab3:    cmpi.w   #27,(a6)        ;27: Size-Box
         bne.s    lab4
         move.w   12(a6),8(a4,d3.w)
         move.w   14(a6),12(a4,d3.w)
         bra      set
```

```
lab4:   cmpi.w   #28,(a6)        ;28: Move-Bar
        bne.s    cont
        move.w   8(a6),0(a4,d3.w)
        move.w   10(a6),4(a4,d3.w)
set:    move.w   #5,d0           ;Funktion: Position verändern
        move.w   0(a5,d3.w),d1   ;Window-Handle
        move.w   0(a4,d3.w),d2   ;Koordinaten
        move.w   8(a4,d3.w),d4
        move.w   12(a4,d3.w),d5
        move.w   4(a4,d3.w),d3   ;d3 darf erst ganz zum Schluß ver-
        jsr      wind_set2       ;ändert werden (ist ja auch Offset)

cont:
        ; Test, ob noch mindestens ein Fenster offen ist

        tst.w    opened
        bne      loop            ;offen, dann zur Schleife
        tst.w    opened+2        ;2. Fenster
        bne      loop

quit:   jsr      gem_exit

        clr.w    -(sp)
        trap     #gemdos


        ;
        ; *** Unterprogramme ***
        ;

intersect:
        ; Test auf Überschneidung von Rechtecken.
        ; Parameter:
        ;   d1-d4 -> Koordinaten Rechteck 1
        ;   Rechteck zwei kommt aus dem Nachrichten-Puffer
        ;   Dessen Startadresse soll in a6 stehen.
        ; Ausgabe:
        ;   d0 -> 0=keine Überschneidung, 1=Überschneidung
        ;   pxy: VDI-Koordinaten (also nicht Breite und Höhe) des
        ;        Schnittrechtecks

        cmp.w    puffer+8,d1     ;x2 > x1?
        bgt      is1
        move.w   puffer+8,pxy    ;ja, dann x=x1
        bra.s    is2
is1:    move.w   d1,pxy          ;sonst x=x2

is2:    cmp.w    puffer+10,d2    ;y2 > y1?
        bgt      is3
```

```
        move.w    puffer+10,pxy+2  ;wie oben, aber für y
        bra.s     is4
is3:    move.w    d2,pxy+2

is4:    move.w    puffer+8,d0      ;x1...
        add.w     puffer+12,d0     ;... + w1
        add.w     d3,d1            ;und x2 + w2
        cmp.w     d1,d0            ;x2+w1 < x1+w1?
        blt.s     is5
        move.w    d1,pxy+4         ;nein, größer -> x=x1+w1
        bra.s     is6
is5:    move.w    d0,pxy+4         ;ja, dann x=x2+w2
is6:    subi.w    #1,pxy+4         ;Korrektur (x2=x+w-1)

        move.w    puffer+10,d0     ;y1...
        add.w     puffer+14,d0     ;... + h1
        add.w     d4,d2            ;und y2 + h2
        cmp.w     d2,d0            ;Rest: wie oben
        blt       is7
        move.w    d2,pxy+6
        bra.s     is8
is7:    move.w    d0,pxy+6
is8:    subi.w    #1,pxy+6

        clr.w     d0               ;Test: Überschneidung?
        move.w    pxy+4,d1
        cmp.w     pxy,d1           ;x2 < x1?
        blt.s     isrts            ;dann Fehler
        move.w    pxy+6,d1         ;y2 < y1?
        cmp.w     pxy+2,d1         ;ebenfalls Fehler
        blt.s     isrts
        moveq.l   #1,d0            ;sonst kein Fehler
isrts:  rts


redraw:
        ; Zeichnet ein Fenster oder einen Ausschnitt aus einem Fenster
        ; neu. Koordinaten des neu zu zeichnenden Bereichs im
        ; Ereignispuffer (+8,+10,+12,+14). Window (0/1) in d3

        lsr.w     #1,d3            ;Offset in Index umwandeln

        move.w    d3,-(sp)         ;für später retten

        move.w    #107,control     ;wind_update
        move.w    #1,control+2     ;(Mausaktivitäten verbieten,
        move.w    #1,control+4     ;Rechteckliste freigeben)
        clr.w     control+6
        clr.w     control+8
        move.w    #1,int_in        ;Flag: Kontrolle selbst übernehmen
```

```
          jsr       aes

          jsr       wind_calc       ;Wir brauchen noch einmal x- und y-
          move.w    pxy,d6          ;Koordinate des Arbeitsbereichs
          move.w    pxy+2,d7

          move.w    puffer+6,d1     ;Erstes Rechteck aus Liste
          move.w    #11,d2          ;Unterfunktions-Nr. von wind_get
          jsr       wind_get

rdr_lp:   tst.w     d3              ;Breite prüfen
          beq       rdr_end         ;Null, dann aufhören

          jsr       intersect       ;sonst Schnittrechteck bilden
          tst.w     d0              ;Überschneidung?
          beq.s     rdr_lab         ;nicht, dann überspringen
          move.w    #129,contrl     ;vs_clip
          move.w    #2,contrl+2
          clr.w     contrl+4
          move.w    #1,contrl+6
          clr.w     contrl+8
          move.w    handle,contrl+12
          move.w    pxy,ptsin       ;Koordinaten des Arbeitsbereichs
          move.w    pxy+2,ptsin+2
          move.w    pxy+4,ptsin+4
          move.w    pxy+6,ptsin+6
          move.w    #1,intin        ;1: Einschalten
          jsr       vdi

          move.w    d6,d4           ;Koordinaten der linken oberen
          move.w    d7,d5           ;Ecke (nicht immer im Rechteck)
          move.w    (sp),d3         ;Fenster-Index (0/1)
          jsr       draw_text

rdr_lab:  move.w    puffer+6,d1     ;nächstes Rechteck
          move.w    #12,d2
          jsr       wind_get
          bra       rdr_lp          ;und nochmal...

rdr_end:  move.w    #129,contrl     ;Clipping aus...
          move.w    #2,contrl+2
          clr.w     contrl+4
          move.w    #1,contrl+6
          clr.w     contrl+8
          move.w    handle,contrl+12
          clr.w     intin           ;0: Clipping aus
          jsr       vdi

          addq.l    #2,sp           ;Retten von d3 rückgängig machen
```

```
          move.w    #107,control     ;wind_update
          move.w    #1,control+2
          move.w    #1,control+4
          clr.w     control+6
          clr.w     control+8
          clr.w     int_in           ;AES übernimmt Kontrolle
          jmp       aes


wind_set1:
          ; Setzt Infozeile oder Titelzeile
          ; Parameter:
          ;   d0 -> Unterfunktion (2 Title/3 Info)
          ;   d1 -> Window-Handle
          ;   a0 -> String

          move.w    #105,control     ;wind_set
          move.w    #6,control+2
          move.w    #1,control+4
          clr.w     control+6
          clr.w     control+8
          move.w    d1,int_in        ;Window-Handle
          move.w    d0,int_in+2      ;Unterfunktion
          move.l    a0,int_in+4      ;Startadresse
          jmp       aes


wind_set2:
          ; Setzt Koordinaten oder bringt Fenster nach oben
          ; Parameter:
          ;   d0 -> Unterfunktion (5 Koordinaten/10 Info)
          ;   d1 -> Window-Handle
          ;   d2-d5 -> Koordinaten

          move.w    #105,control     ;wind_set
          move.w    #6,control+2
          move.w    #1,control+4
          clr.w     control+6
          clr.w     control+8
          move.w    d1,int_in        ;Window-Handle
          move.w    d0,int_in+2      ;Unterfunktion
          move.w    d2,int_in+4
          move.w    d3,int_in+6
          move.w    d4,int_in+8
          move.w    d5,int_in+10
          jmp       aes


create_windows:
          ; window_create für 2 Fenster
```

```
          ; Ergebnis in d0:
          ;    0 -> OK
          ;    1 -> Fehler
          ; Window-Handles in whandle und whandle+2

          move.l    d7,-(sp)
          moveq.l   #0,d7
          lea       whandle,a5

cw_loop:  move.w    #100,control      ;wind_create
          move.w    #5,control+2
          move.w    #1,control+4
          clr.w     control+6
          clr.w     control+8
          move.w    #63,int_in        ;alles außer Pfeile und Schieber
          move.w    xdesk,int_in+2    ;maximale Größe des Fensters (x)
          move.w    ydesk,int_in+4    ;(y)
          move.w    wdesk,int_in+6    ;(Breite)  die Werte stammen aus
          move.w    hdesk,int_in+8    ;(Höhe)    dem Include-File
          jsr       aes
          tst.w     int_out           ;negativ? Dann Fehler!
          bmi       cw_err
          move.w    int_out,0(a5,d7.w)  ;sonst als Window-Handle merken
          addq.l    #2,d7             ;nächstes Fenster
          cmpi.w    #4,d7             ;schon durch?
          bne.s     cw_loop
          clr.w     d0                ;kein Fehler
          bra.s     cw_rts

cw_err:   move.w    #1,d0             ;Fehler-Flag
cw_rts:   move.l    (sp)+,d7
          rts


draw_text:
          ; Löscht Arbeitsbereich und schreibt Text
          ; Parameter:
          ;   in pxy (.bss-Segment) die Koordinaten des zu löschenden
          ;   Rechtecks
          ;   d4,d5: x/y Koordinaten der linken oberen Ecke des
          ;   Textes (muß nicht im Rechteck liegen, siehe Redraw)

          ; Mauszeiger abschalten

          move.w    #123,contrl       ;v_hide_c
          clr.w     contrl+2
          clr.w     contrl+4
          clr.w     contrl+6
          clr.w     contrl+8
          move.w    handle,contrl+12
```

```
jsr       vdi

; Rechteck in pxy löschen:

; vsf_interior (Füllmuster):

move.w    #23,contrl
clr.w     contrl+2
clr.w     contrl+4
move.w    #1,contrl+6
move.w    #1,contrl+8
move.w    handle,contrl+12
clr.w     intin           ;Füllen mit Hintergrundfarbe
jsr       vdi

; vsf_perimeter (Umrandung ausschalten):

move.w    #104,contrl
clr.w     contrl+2
clr.w     contrl+4
move.w    #1,contrl+6
move.w    #1,contrl+8
move.w    handle,contrl+12
clr.w     intin           ;Flag: Umrahmung ausschalten
jsr       vdi

; v_bar (löscht den Arbeitsbereich):

move.w    #11,contrl      ;Opcode für graphische
move.w    #2,contrl+2     ;Grundfunktionen
clr.w     contrl+4
clr.w     contrl+6
clr.w     contrl+8
move.w    #1,contrl+10    ;Funktionsnummer für v_bar
move.w    handle,contrl+12
move.w    pxy,ptsin
move.w    pxy+2,ptsin+2
move.w    pxy+4,ptsin+4
move.w    pxy+6,ptsin+6
jsr       vdi

; vsf_perimeter (Umrandung wieder einschalten):

move.w    #104,contrl
clr.w     contrl+2
clr.w     contrl+4
move.w    #1,contrl+6
move.w    #1,contrl+8
move.w    handle,contrl+12
move.w    #1,intin        ;Flag: Umrahmung einschalten
```

```
        jsr     vdi

        ; Ausgabe der Texte mit v_gtext

        move.w  #8,contrl
        move.w  #1,contrl+2
        clr.w   contrl+4        ;contrl+6 wird je nach Textlänge
        clr.w   contrl+8        ;gesetzt
        move.w  handle,contrl+12

        move.w  d4,ptsin        ;x-Koordinate
        addi.w  #8,ptsin

        move.w  d5,ptsin+2      ;y-Koordinate
        addi.w  #14,ptsin+2

        lea     zeile1,a0
        jsr     fix_text        ;Text in intin-Array schreiben
        jsr     vdi

        addi.w  #16,ptsin+2     ;nächste Zeile
        lea     zeile2,a0
        jsr     fix_text
        jsr     vdi

        addi.w  #16,ptsin+2
        lea     zeile3,a0
        jsr     fix_text
        jsr     vdi

        ; Mauszeiger wieder zeigen

        move.w  #122,contrl     ;v_show_c
        clr.w   contrl+2
        clr.w   contrl+4
        move.w  #1,contrl+6
        clr.w   contrl+8
        move.w  handle,contrl+12
        move.w  #1,intin
        jsr     vdi
        rts


output_text:
        ; Gibt zu Beginn des Programms einmal den Text aus
        ; Parameter:
        ;   d3: gewünschtes Fenster (0/1)

        jsr     wind_calc       ;Arbeitsbereich berechnen
```

```
        ; Clipping für diesen Arbeitsbereich einschalten

        move.w    #129,contrl       ;Funktions-Opcode
        move.w    #2,contrl+2
        clr.w     contrl+4
        move.w    #1,contrl+6
        clr.w     contrl+8
        move.w    handle,contrl+12
        move.w    pxy,ptsin         ;Koordinaten des (von wind_calc)
        move.w    pxy+2,ptsin+2
        move.w    pxy+4,ptsin+4
        move.w    pxy+6,ptsin+6
        move.w    #1,intin          ;1: Einschalten
        jsr       vdi

        move.w    pxy,d4            ;linke obere Ecke des Textes
        move.w    pxy+2,d5
        jsr       draw_text

        ; Clipping ausschalten

        move.w    #129,contrl       ;Funktions-Opcode
        move.w    #2,contrl+2
        clr.w     contrl+4
        move.w    #1,contrl+6
        clr.w     contrl+8
        move.w    handle,contrl+12
        clr.w     intin             ;0: Clipping aus
        jsr       vdi
        rts


wind_calc:
        ; Errechnet Arbeitsbereich.
        ; Parameter:
        ;   d3: gewünschtes Fenster (0/1)
        ; Ausgabe:
        ;   Bereich in x1/y1/x2/y2 in pxy

        movem.l   d0/d1/d3/a0,-(sp)

        lsl.w     #1,d3             ;Index * 2 als Offset
        lea       wx,a0             ;Basisadresse für "Array"
        move.w    #108,control      ;wind_get
        move.w    #6,control+2
        move.w    #5,control+4
        clr.w     control+6
        clr.w     control+8

        move.w    #1,int_in         ;1 -> Arbeitsbereich ausrechnen
```

```
        move.w    #63,int_in+2     ;Eigenschaften des Fensters
        move.w    0(a0,d3.w),int_in+4 ;äußere Ausmaße (x)
        move.w    4(a0,d3.w),int_in+6 ;(y)
        move.w    8(a0,d3.w),int_in+8 ;(w)
        move.w    12(a0,d3.w),int_in+10        ;(h)

        jsr       aes

        move.w    int_out+2,d0     ;x1 = x aus wind_calc
        move.w    d0,pxy
        move.w    int_out+4,d1     ;y1 = y aus wind_calc
        move.w    d1,pxy+2
        add.w     int_out+6,d0     ;x + Breite...
        subq.w    #1,d0            ;... - 1...
        move.w    d0,pxy+4         ;... = x2
        add.w     int_out+8,d1     ;y + Höhe...
        subq.w    #1,d1            ;... - 1...
        move.w    d1,pxy+6         ;... = y2

        movem.l   (sp)+,d0/d1/d3/a0
        rts


wind_get:
        ; Liefert Rechteckliste oder Größe des Desktop
        ; Parameter:
        ;   d1: Window-Handle (Desktop: 0)
        ;   d2: Funktion:
        ;        4: Größe Arbeitsbereich
        ;       11: Erstes Rechteck
        ;       12: Nächstes Rechteck
        ; Ausgabe:
        ;   Koordinaten in d1-d4

        move.w    #104,control     ;wind_get
        move.w    #2,control+2
        move.w    #5,control+4
        clr.w     control+6
        clr.w     control+8
        move.w    d1,int_in        ;Window-Handle
        move.w    d2,int_in+2      ;Unterfunktion
        jsr       aes
        move.w    int_out+2,d1
        move.w    int_out+4,d2
        move.w    int_out+6,d3
        move.w    int_out+8,d4
        rts
```

```
fix_text:
          ; Unterprogramm, daß einen String (Startadresse in a0
          ; zu übergeben) in das intin-Array schreibt, die Länge
          ; bestimmt und in contrl[3] ablegt

          clr.w     d0              ;Länge
          clr.w     d1              ;Hilfsregister
          lea       intin,a1

fix_loop: move.b    (a0)+,d1        ;ein Byte aus Zielstring holen
          tst.b     d1              ;Stringende?
          beq.s     fix_end
          move.w    d1,(a1)+        ;nein, dann als Wort ins
          addq.w    #1,d0           ;intin-Arrary schreiben
          bra.s     fix_loop

fix_end:  move.w    d0,contrl+6     ;Länge festhalten
          rts


          DATA

opened:   DC.w 1,1 ;Flag ob Fenster offen

wx:       DC.w 20,50              ;Koordinaten der Fenster
wy:       DC.w 20,50
ww:       DC.w 200,200
wh:       DC.w 120,120

title1:   DC.b 'Fenster 1',0
title2:   DC.b 'Fenster 2',0

info1:    DC.b 'Bitte beachten Sie:',0
info2:    DC.b 'Bitte ebenfalls beachten:',0

err_txt:  DC.b '[3][Sorry!¦Kein Window-Handle mehr frei!][OK]',0

zeile1:   DC.b 'Hallo! Das ist eine Fenster-Demonstration!',0
zeile2:   DC.b 'Sie können das Fenster bewegen und seine Größe '
          DC.b 'verändern.',0
zeile3:   DC.b 'Ende durch Klick auf die Close-Box!',0


          BSS

whandle:  DS.w 2   ;Window-Handles

xdesk:    DS.w 1   ;Koordinaten des Desktop
ydesk:    DS.w 1
wdesk:    DS.w 1
```

```
hdesk:    DS.w 1

puffer:   DS.w 8   ;Nachrichtenpuffer

pxy:      DS.w 4   ;Koordinaten

          END      ;Ganz schön lang, was?
```

5.9 Resource-Files

Die Fenster aus dem letzten Kapitel sind ja nur ein grundsätzlicher Bestandteil eines typischen GEM-Programms. Zwei weitere kommen noch hinzu: Menüleisten und Dialogboxen. Was das ist, brauche ich Ihnen wohl nicht zu erklären. Sie könnten jetzt aber fragen, warum beides unter der gleichen Überschrift behandelt wird, die dazu auch noch nichts mit Menüs und Dialogen zu tun zu haben scheint.

Nun, Menüleisten und Dialogboxen gehören zu den Dingen, die intern etwas mehr Aufwand für den Rechner bedeuten. Die Möglichkeiten dieser Hilfsmittel sind so komplex, daß es zu kompliziert wäre, etwa eine ganze Dialogbox im Programm selbst aufzubauen (das sähe etwa so aus: Mache da einen Kasten mit dem Text und dort zwei Knöpfe, und da stell noch ein Bildchen hin...). Deshalb werden solche Boxen mit einem separaten Hilfsprogramm entworfen, das ähnlich einem objektorientierten Grafikprogramm wie GEM-DRAW funktioniert. Für Menüs gilt das gleiche. Dieses Hilfsprogramm erzeugt dann aus Ihren Angaben eine Datei, die die richtige Applikation laden und benutzen kann (wir werden sehen, wie das geht). In dieser Datei sind praktisch alle möglichen Dialogboxen und Menüs in Aussehen und Funktion definiert, die das Programm irgendwann einmal braucht. Weil diese Datei somit eine wichtige 'Quelle' für die Applikation ist, nennt man die Datei selbst ein 'Resource-File' (resource = Hilfsquelle). Der Dateiname endet in der Regel mit der Extension .RSC.

Zwei ganz wesentliche Vorteile bringt diese Datei mit sich: Erstens können mit dem Hilfsprogramm auch große Eingabemasken komfortabel entworfen werden, die dann im Programm mit einer einzigen AES-Funktion abgearbeitet werden können. Und zweitens ist es problemlos möglich, z.B. ein amerikanisches Programm ins Deutsche zu übersetzen: Alle Änderungen können mit dem Hilfsprogramm am Resource-File vorgenommen werden; das Programm selbst muß weder editiert noch neu compiliert werden, wenn die Bedienung über Menüs und Dialogboxen abgewickelt wird.

Sie benötigen im folgenden unbedingt ein solches Hilfsprogramm zur Erstellung von Resource-Dateien; diese Programme heißen Resource-Construction-Sets oder ähnlich. Da sich für den Atari ST kein besonderes RCS durchgesetzt hat, kann ich Ihnen leider nicht allgemein sagen, wie man mit solchen Programmen umgeht. Manchmal gehört ein RCS zum Lieferumfang einer Programmiersprache und wenn nicht, dann müssen Sie halt etwas probieren; sehr schwer ist die Arbeit ohnehin nicht, denn es ähnelt wie gesagt einem Zeichenprogramm.

Grundsätzlich müssen Sie folgendes wissen: Jedes Resource-File besteht aus mindestens einem Baum. Ein Baum ist folglich die gröbste Struktur innerhalb eines Resource-Files. Jeder Baum enthält, wenn das File fertig ist, alle Objekte (das sind also Teile eines Baums), die gezeichnet werden müssen, um z.B. eine Dialogbox auf den Bildschirm zu bringen. Die Objekte können auf verschiedenen Ebenen in diesem Baum liegen: Jedes Objekt, das sich ganz innerhalb eines anderen Objekts befindet, ist ein Unter-Objekt dieses anderen Objekts. Beispiel: Ein Text in einem Rechteck ist ein Unterobjekt dieses Rechtecks. Der äußere Rahmen einer Dialogbox ist also das Objekt, das allen anderen Objekten des gleichen Baumes übergeordnet ist (es enthält ja alle anderen Objekte). Man nennt es deshalb auch Wurzelobjekt. Diese Hierarchie ist für Sie zunächst einmal nur deshalb wichtig, weil es Auswirkungen auf Bewegungen der Objekte innerhalb des RCS hat: Wenn Sie nämlich beim Editieren eines Resource-Files ein Objekt an eine neue Position bewegen, dann werden alle diesem Objekt untergeordneten Objekte automatisch mitbewegt. Das ist eine praktische Sache. Wofür die verschiedenen Ebenen sonst noch gut sind, werden wir etwas später sehen.

Zurück zu den Bäumen: Jedes gute RCS bietet verschiedene Baumtypen. Dazu gehören mindestens die Typen Dialog und Menü (es dürfte klar sein, wozu). Oft wird auch der Baumtyp Free angeboten, der eigentlich nur eine Unterabteilung der Dialog-Bäume ist. Diese Unterscheidung verschiedener Bäume gibt es allerdings nur scheinbar. Sie erleichtert Ihnen "nur" das Erstellen der Bäume mit dem RCS. So werden die Rechtecke, die die einzelnen Menü-Einträge in einem Menü-Baum enthalten, vom RCS automatisch erzeugt usw. Deutlicher wird dieser Pseudo-Baumtyp am Beispiel der Dialog- und Free-Bäume: Ein Dialog-Baum ist mit dem Free-Baum völlig identisch. Allerdings können Sie im RCS die Objekte nur dann in ganz feinen Schritten (Pixelbreite) bewegen, wenn Sie den Baum als Free-Baum deklariert haben; im Dialog-Baum werden alle Objekte auf glatte Koordinaten (durch 8 teilbar) gerundet.

Wichtig ist noch, daß wir später in der Applikation die einzelnen Bäume und die Objekte auseinanderhalten können, um z.B. nicht nur irgendeine, sondern eine ganz bestimmte Dialogbox auf den Bildschirm zaubern zu lassen. Dafür gibt es eine Funktion im RCS, die meist Name heißt. Mit dieser Funktion können Sie ein Objekt, das vorher angeklickt wurde, benennen. Das bedeutet nichts weiter, als daß Sie der Nummer eines Baumes oder eines Objektes (diese sind nämlich durchnumeriert) einen Namen geben. Wenn Sie Ihr Resource-File abspeichern, dann wird neben dem .RSC-File auch noch eine andere Datei gesichert, die Sie in Ihren Programmtext einbinden oder per Include-Anweisung lesen lassen können. Dort werden dann alle Namen der Objekte, die Sie im RCS angegeben haben, als Konstanten deklariert und können über diese angesprochen werden.

So, jetzt kann ich Ihnen eigentlich nicht mehr viel über das RCS sagen. Sie sollten an dieser Stelle die Lektüre unterbrechen, um mit Ihrem RCS ein wenig zu spielen. Machen Sie eine Sicherheitskopie vom Resource-File eines Programms und untersuchen Sie diese Datei mit dem RCS (es ist möglich, daß Sie beim Versuch, eine fertige RSC-Datei zu bearbeiten, eine Fehlermeldung erhalten - ignorieren Sie diese!). Versuchen Sie auch einmal, eine eigene RSC-Datei ganz neu zu erstellen; erforschen Sie dabei, soweit es geht, die Möglichkeiten des RCS. Das ist die Bedingung für das Verständnis der folgenden Abschnitte.

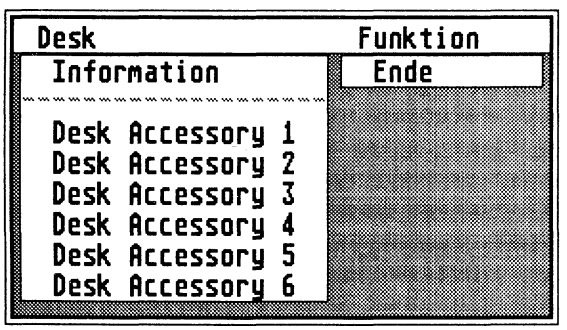
5.9.1 Die Menüleiste

Ich will hier nicht die Vor- und Nachteile einer mausgesteuerten Menüleiste aufzeigen, sondern Ihnen in einem kleinen Beispielprogramm zeigen, wie einfach sich die Anwendung einer fertigen Menüleiste gestaltet, wenn sie erst einmal mit dem RCS erstellt wurde. Mit dem RCS haben Sie sich inzwischen hoffentlich angefreundet.

Unsere Menüleiste soll zwei Oberbegriffe erhalten, man spricht von Menütiteln (englisch: Title): Desk und Funktio'. Den Desk-Titel muß jede Menüleiste haben, obwohl er nicht Desk heißen muß. Hier findet man später die Accessories wieder. Der einzige für Sie freie Menü-Eintrag (englisch: Entry) würde gewöhnlich für eine Copyright-Meldung, zum Anzeigen der Versionsnummer und ähnliches verwendet; bei uns soll er Information heißen. Merken Sie sich: Man unterscheidet zwischen Menütitel und Menü-Eintrag (das, was unter dem Titel ausgefahren wird, wenn der Mauszeiger den Titel berührt). Der zweite Titel soll nur einen Eintrag erhalten: Ende. Mit ihm soll das Programm später verlassen

werden können. Laden Sie also Ihr RCS, richten Sie einen Menübaum ein (das RCS fragt dabei nach dem Namen, nennen Sie den Baum MENUE), und legen Sie die Einträge an. Der Text eines Titels oder eines Eintrags wird übrigens gewöhnlich dadurch geändert, daß ein Doppelklick auf dem Titel bzw. Eintrag ausgeführt wird.

So sollten die beiden Menüs am Ende aussehen:



Die Menütitel sind in dieser Abbildung nur deshalb soweit auseinandergezogen, damit Sie die Einträge besser lesen können; bei Ihnen sollte 'Funktion' also direkt neben 'Desk' stehen.

Nun müssen Sie den beiden Einträgen nur noch zwei Namen geben: nennen wir sie doch 'INFO' und 'ENDE'. Übrigens: Umlaute sollten diese Bezeichner nicht enthalten, weil das später im Programm zu Fehlermeldungen führen würde.

Wenn Sie damit fertig sind, dann können Sie die fertige Resource-Datei abspeichern; sie soll MENU1.RSC heißen. Dabei sichert das RCS auch gleich zwei weitere Dateien: Eine Datei namens MENU1.H und eine namens MENU1.DEF oder MENU1.DFN, das hängt von Ihrem RCS ab. Die H-Datei ist eine Header-Datei für C-Compiler, die drei Konstanten deklarieren wird: MENUE, INFO und ENDE - das sind genau die Namen, die wir dem Baum und seinen Objekten gegeben haben. Damit ist später im Programm die Identifizierung der einzelnen Objekte sichergestellt. Die DEF-Datei ist so etwas Ähnliches. Allerdings wird sie nicht für das Programm benötigt, das das Resource-File lädt; vielmehr dient sie als Gedächtnisstütze für das RCS selbst und sorgt dafür, daß, wenn Sie das Resource-File ein zweites Mal bearbeiten wollen, die bisher vereinbarten Namen noch bekannt sind. Sie können diese Datei ignorieren, sollten sie aber nicht löschen.

Schauen wir uns aber die Header-Datei einmal an:

```
#define MENUE 0
#define INFO 7
#define ENDE 16
```

Wie ich gerade schon sagte, ist diese Datei für C-Programme gedacht. Wenn wir in BASIC oder in Assembler arbeiten, so müßten wir die Zeilen etwas umschreiben: Das '#define' entfällt genauso wie das folgende Leerzeichen, und statt des zweiten Leerzeichens zwischen dem Bezeichner und der Zahl schreiben wir ein Gleichheitszeichen. Mit den Resource-Files wachsen natürlich auch die Header-Dateien, und es ist nicht besonders aufregend, diese Umwandlung von Hand durchzuführen. Lassen wir das also den Computer machen. Das folgende Programm erwartet als Eingabe nur den Dateinamen des Header-Files ohne (!) die Extension H und wird eine Datei mit der Extension H2 erzeugen, die Sie in Ihr Programm mit Merge (BASIC) oder Include (Assembler) einbinden können. Da das Programm keine außergewöhnlichen Tricks benutzt und mehr der Anwendung als der Demonstration dient, habe ich es nur in Omikron-BASIC geschrieben. Es befindet sich aber compiliert auf der Diskette im Buch.

```
'
' Umwandlung von C-Header-Files des RCS in BASIC-Files
' Omikron-BASIC       MP   09-12-88        BASHEAD.BAS
'
PRINT "Umwandlung von C-Header-Files (*.H) vom RCS"
PRINT "Dateiname ohne Extension: "
INPUT Filename$
'
OPEN "I",1,Filename$+".H"'    Quelldatei zum Lesen
OPEN "O",2,Filename$+".H2"'    Zieldatei zum Schreiben
'
WHILE NOT EOF(1)
  INPUT #1,A$'           Originalzeile holen
  A$= MID$(A$,9)'        #define löschen
  P= INSTR(A$," ")'      Leerzeichen suchen
  PRINT #2, LEFT$(A$,P)+"="+ MID$(A$,P)
WEND
'
CLOSE 1: CLOSE 2
END

Die Datei MENU1.H2, die dieses Programm erstellen soll, sieht so aus:
MENUE = 0
INFO = 7
ENDE = 16
```

So, damit hätten wir ein Resource-File und ein Header-File. Aber wir wollten eine Menüleiste haben. Kein Problem! Dafür gibt es die Menu-Library des AES, deren Funktionen, wer hätte es gedacht, alle mit menu_ beginnen. Die Funktionen sollten Sie sich als erstes einmal in Ihrer GEM-Dokumentation anschauen,-aber lassen Sie menu_register bitte aus, die wird nämlich nur von Accessories benutzt.

Was die einzelnen Funktionen machen, dürfte eigentlich leicht zu verstehen sein. Allerdings taucht in jedem Aufruf ein Parameter auf, der laut Beschreibung die Adresse des Menü-Objektbaumes enthalten soll. Dabei wird das Problem deutlich, das wir noch vor uns haben: Der Menü-Objektbaum (das ist einfach ein Baum aus Objekten, so wie Sie ihn mit dem Resource-Construction-Set gebaut haben), befindet sich ja in einer RSC-Datei auf dem Massenspeicher. Und bevor wir die Menüleiste zeichnen lassen können, müssen wir diesen Baum in den Speicher bekommen. Die Adresse des Menü-Objektbaumes ist dann natürlich die Adresse, an der der Baum im Speicher abgelegt wird.

Ich könnte Sie jetzt mit seitenlangen Erklärungen zu Baumstrukturen und dem AES-Objektformat im Speicher langweilen. Falls Sie das interessiert, finden Sie Informationen darüber in jedem guten GEM-Buch. Da diese Informationen aber nicht sehr wichtig sind und Sie wahrscheinlich lieber Ihr erstes eigenes Menü auf dem Bildschirm sehen möchten, lasse ich das aus. Statt dessen beschreibe ich, wie Sie Ihr Resource-File vom Massenspeicher ins RAM bekommen.

Das ist einfacher, als Sie erwarten mögen. Die AES-Funktion rsrc_load führt nämlich alle dazu nötigen Schritte durch. Der einzige Parameter ist der Dateiname des Resource-Files. Als Funktionswert erhalten Sie ein Flag, das, falls es Null ist, einen Fehler signalisiert.

Weil wir aber damit die Startadresse des Menübaumes noch nicht kennen, muß es noch eine andere Funktion geben, um diese Adresse zu ermitteln. Aber was heißt hier des Menübaums? In unserem ersten Beispiel haben wir zwar wirklich nur einen Baum, doch kann es durchaus mehrere Menübaume in einem Resource-File geben. Aber wozu haben wir diesen Bäumen Namen gegeben? Die helfen uns jetzt weiter.

Die Funktion, welche die Startadresse eine Menübaumes ermitteln kann, nennt sich rsrc_gaddr (Resource-Get-Address). Sie kann noch mehr, aber nicht alles auf einmal... Also: Sie übergeben der Funktion einen Baum-Index, damit diese weiß, von welchem der möglicherweise mehreren Bäume Sie die Startadresse erfragen möchten. Dieser Index ist iden-

tisch mit dem Namen, den Sie diesem Baum im RCS gegeben haben, und der auch im Header-File .H oder .H2 (s.o.) enthalten war. In unserem Fall übergeben wir also einfach die Konstante MENUE.

Außerdem will rsrc_gaddr von Ihnen wissen, welcher Art dieses 'Ding', das Sie suchen, ist. Wir suchen einen Baum, und deshalb geben wir hier eine Null an (es kommt selten vor, daß Sie hier etwas anderes als Null angeben müssen). Schließlich ist noch ein Rückgabeparameter zu übergeben, in dem Sie nach dem Aufruf endlich die Startadresse des Baumes finden. Das Ganze darf aber natürlich erst dann geschehen, wenn das Resource-File mit rsrc_load geladen wurde.

So, nehmen wir an, Sie haben die Startadresse unseres Menübaums. Dann wollen Sie natürlich als nächstes die Menüleiste auf den Bildschirm bringen. Dafür gibt es die Funktion menu_bar, die zwei Parameter erhält: die Startadresse des Baumes natürlich und ein Flag, das bestimmt, ob die Menüzeile neu gezeichnet (1) oder vom Bildschirm entfernt (0) werden soll. Nach dem Aufruf ist das Menü am Bildschirm zu sehen und könnte vom Anwender bedient werden.

Dazu müssen Sie ihm allerdings eine Gelegenheit bieten, indem Sie eine der Ereignis-Funktionen (evnt_...) aufrufen. Um auch auf die Wahl des Anwenders reagieren zu können, sollte es allerdings nicht irgendeine, sondern die Funktion evnt_mesag sein (die Windows lassen grüßen); denn auch das Anklicken eines Menüpunktes wird als Ereignis in dem Ihnen bereits bekannten Nachrichtenpuffer gemeldet. Das entsprechende Ereignis hat die Nummer 10 und heißt offiziell MN_SELECTED. Sie finden außerdem in Wort 3 die Nummer des Menütitels und in Wort 4 die Nummer des Menü-Eintrags, der angeklickt wurde. Mit Nummer ist hier wieder die Zahl gemeint, die Ihr RCS als Konstante auswirft, wenn Sie dem Titel oder dem Eintrag einen Namen geben. Eins sei hier gleich klargestellt: Alle Einträge eines Menüs können anhand ihrer Eintrag-Nummern identifiziert werden, das heißt Einträge unter verschiedenen Menütiteln haben nie gleiche Eintragnummern. Deshalb habe ich auch in der Resource-Datei MENU1.RSC, die wir vorhin zusammengestellt haben, den beiden Titeln keinen Namen gegeben, nur die Einträge wurden mit INFO und ENDE bezeichnet.

Wir warten also auf die Nachricht und entscheiden dann mit dem Wort Nr. 4 des Ereignispuffers, was der Benutzer von uns wollte. Also lassen wir entweder eine kleine Alertbox mit einem Copyright-Hinweis erscheinen oder wir verlassen das Programm nach einer Sicherheitsabfrage (ebenfalls durch eine Alertbox). Wenn wir das, was der Benutzer wollte, ausgeführt haben, dann gibt es jedoch noch etwas zu erledigen: Der

Menütitel, der ja bei Mauszeigerkontakt invertiert wird, also weiß auf schwarz erscheint, muß wieder in den normalen Zustand versetzt werden. Das macht die Funktion menu_tnormal (Title-Normal). Als Parameter verlangt sie unter anderem die Nummer des Menütitels, der automatisch im Wort Nr. 3 des Ereignispuffers mitgeliefert wird.

Vor dem Verlassen sind jetzt noch zwei Dinge zu tun, die wir bis jetzt nicht ausführen mußten: Die Menüzeile muß wieder gelöscht werden (ebenfalls mit menu_bar), und der Speicherplatz, den das Resource-File beansprucht, muß offiziell freigegeben werden. Das macht die Funktion rsrc_free, die ohne Parameter aufgerufen wird.

Nun steht einem Beispielprogramm nichts mehr im Wege, mit dem wir die Resource-Datei ausprobieren können. Falls Sie diese noch nicht hergestellt haben, so finden Sie auf der Diskette im Buch auch die fertige Datei MENU1.RSC. Falls Sie Ihre eigene Datei testen möchten, so müssen Sie vorher überprüfen, ob die Nummern der Einträge und die Nummer des Baumes mit meinen Werten übereinstimmen, weil diese unterschiedlich sein können, wenn Sie die Objekte in einer anderen Reihenfolge angelegt haben, was aber durchaus nicht falsch ist. Versuchen Sie auch einmal, aus dem Programm heraus ein Accessory aufzurufen!

GFA-BASIC

```
'
' Laden des Resource-Files und Anzeigen einer Menüleiste
' GFA-BASIC              MP  10-12-88          MENU1.GFA
'
' Für rsrc_load muß Speicher an GEMDOS zurückgegeben werden:
'
RESERVE -33000   ! Das muß reichen, da Resource-Files
'                  nicht größer als 32K werden
'
menue=0   ! Diese Konstanten wurden mit 'Merge' aus der Datei
info=7    ! MENU1.H2 geladen, die mit dem Programm BASHEAD
ende=16   ! erzeugt werden kann (siehe Beschreibung).
'
puffer$=SPACE$(16)   ! Platz für 8 Worte
'
DEFFN p(x)=CVI(MID$(puffer$,x*2+1,2))  ! Funktion holt
'                                        Wort Nr. x aus dem Puffer
'
VOID APPL_INIT()
'
' Laden des Resource-Files (MENU1.RSC):
'
IF RSRC_LOAD("MENU1.RSC")=0
```

```
  VOID FORM_ALERT(1,"[3][Kein Resource-File!][Abbruch]")
ELSE
  '
  ' Adresse des Menü-Objektbaumes erfragen (0 für Baum):
  '
  VOID RSRC_GADDR(0,menue,menue_adresse%)
  '
  ' Anzeigen (1) der Menüleiste:
  '
  VOID MENU_BAR(menue_adresse%,1)
  '
  ' Ereignisschleife:
  '
  abbruch%=0
  '
  REPEAT
    VOID EVNT_MESAG(VARPTR(puffer$))
    '
    IF FN p(0)=10          ! Menüpunkt angeklickt?
      title%=FN p(3)       ! Nummer des Titels merken
      IF FN p(4)=info      ! Information gewünscht?
        VOID FORM_ALERT(1,"[1][Menü-Demoprogramm|(c) 1989 Data...
              ...Becker GmbH][Weiter]")
      ENDIF
      IF FN p(4)=ende      ! oder Programmende?
        IF FORM_ALERT(2,"[2][Wirklich beenden?][Ja|Nein]")=1
          abbruch%=1
        ENDIF
      ENDIF
      '
      ' Menütitel normal (1) stellen:
      '
      VOID MENU_TNORMAL(menue_adresse%,title%,1)
    ENDIF
  UNTIL abbruch%=1
  '
  VOID MENU_BAR(menue_adresse%,0)  ! Menüzeile löschen
  VOID RSRC_FREE()                 ! Platz freigeben
ENDIF
'
RESERVE
VOID APPL_EXIT()
'
END
'
```

Omikron-BASIC

```
'
' Laden des Resource-Files und Anzeigen einer Menüleiste
' Omikron-BASIC          MP  10-12-88          MENU1.BAS
'
' Für rsrc_load muß Speicher an GEMDOS zurückgegeben werden:
'
CLEAR 33000'    Das muß reichen, da Resource-Files
'               nicht größer als 32K werden
'
Menue=0'   Diese Konstanten wurden mit 'Merge' aus der Datei
Info=7'    MENU1.H2 geladen, die mit dem Programm BASHEAD
Ende=16'   erzeugt werden kann (siehe Beschreibung).
'
Puffer$= SPACE$(16)' Platz für 8 Worte
'
DEF FN P(X)= CVI( MID$(Puffer$,X*2+1,2))'  Funktion holt
'                                          Wort Nr. x aus dem Puffer
'
Appl_Init
'
' Laden des Resource-Files (MENU1.RSC):
'
Rsrc_Load("MENU1.RSC",Back)
IF Back=0 THEN
 FORM_ALERT (1,"[3][Kein Resource-File!][Abbruch]")
ELSE
 '
 ' Adresse des Menü-Objektbaumes erfragen (0 für Baum):
 '
 Rsrc_Gaddr(0,Menue,Menue_Adresse)
 '
 ' Anzeigen der Menüleiste:
 '
 Menu_Bar(Menue_Adresse)
 '
 ' Ereignisschleife:
 '
 Abbruch=0
 '
 REPEAT
   Evnt_Mesag(Puffer$)
   '
   IF FN P(0)=10 THEN '       Menüpunkt angeklickt?
     Title=FN P(3)'           Nummer des Titels merken
     IF FN P(4)=Info THEN '   Information gewünscht?
       FORM_ALERT (1,"[1][Menü-Demoprogramm|(c) 1989 Data...
              ...Becker GmbH][Weiter]")
     ENDIF
     IF FN P(4)=Ende THEN '   oder Programmende?
```

```
      FORM_ALERT (2,"[2][Wirklich beenden?][Ja|Nein]",Back)
      IF Back=1 THEN Abbruch=1
    ENDIF
    '
    ' Menütitel normal (1) stellen:
    '
    Menu_Tnormal(Title,1)
  ENDIF
 UNTIL Abbruch=1
 '
 Menu_Bar(Menue_Adresse)'    Menüzeile löschen
 Rsrc_Free' Platz freigeben
ENDIF
'
Appl_Exit
'
END
'
```

Denken Sie an die geteilten Zeilen!

C

```
/*************************************************************/
/*   Laden des REC-Files und Anzeigen einer Menüleiste   */
/*   Megamax Laser C       MP  10-12-88        MENU1.C   */
/*************************************************************/

#include "menu1.h"    /* Konstanten des RSC-Files */
/* Zur Erinnerung: Die Konstanten heißen MENUE, INFO und ENDE */

int  puffer[8],
     title,
     abbruch = 0;
long menue_adresse;

main()
{
  appl_init();
  graf_mouse (0, 0L);   /* Pfeil als Mauszeiger */

  /* Resource-File laden */

  if (rsrc_load ("MENU1.RSC") == 0)  /* Fehler beim Laden? */
    form_alert (1, "[3][Kein Resource-File!][Abbruch]");
  else
  {
    /* Adresse des Menübaumes beschaffen */
    rsrc_gaddr (0, MENUE, &menue_adresse);
```

```
    menu_bar (menue_adresse, 1);  /* Menüleiste zeigen */

    /* Ereignis-Warteschleife: */

    do
    {
      evnt_mesag (puffer);

      if (puffer[0] == 10)   /* Menüpunkt angeklickt? */
      {
        title = puffer[3];   /* für später merken */

        switch (puffer[4])
        {
          case INFO:  form_alert (1,
                        "[1][Menü-Demoprogramm|(c) 1989 Data...
                             ...Becker GmbH][Weiter]");
                      break;

          case ENDE:  if (form_alert (2,
                        "[2][Wirklich beenden?][Ja|Nein]") == 1)
                          abbruch = 1;
                      break;
        }
        menu_tnormal (menue_adresse, title, 1);
      }
    }
    while (!abbruch);

    menu_bar (menue_adresse, 0);  /* Menü wieder löschen */
    rsrc_free();
  }
  appl_exit();
}
```

Assembler

```
          ;
          ; Resource-File laden und Menüleiste anzeigen
          ; Assembler       MP  11-12-88       MENU1.IS
          ;

          .INCLUDE 'GEM_INEX.IS'

gemdos    = 1

MENUE     = 0       ;Konstanten für den Menübaum und die beiden
INFO      = 7       ;Einträge (aus der Datei MENU1.H2, mit
ENDE      = 16      ;BASHEAD.TOS erzeugt)
```

```
        .TEXT

main:   jsr     gem_init

        ; Pfeil als Mauszeiger:

        move.w  #78,control     ;graf_mouse
        move.w  #1,control+2
        move.w  #1,control+4
        move.w  #1,control+6
        clr.w   control+8
        clr.w   int_in          ;0 für Pfeil
        jsr     aes

        ; rsrc_load:

        move.w  #110,control    ;Opcode
        clr.w   control+2
        move.w  #1,control+4
        move.w  #1,control+6
        clr.w   control+8
        move.l  #rsc_name,addr_in
        jsr     aes

        tst.w   int_out         ;Fehler aufgetreten?
        beq     rsc_error

        ; Adresse des Menü-Objektbaums feststellen (rsrc_gaddr)

        move.w  #112,control
        move.w  #2,control+2
        move.w  #1,control+4
        clr.w   control+6
        move.w  #1,control+8
        clr.w   int_in          ;0 -> wir suchen einen Baum
        move.w  #MENUE,int_in+2 ;Index des Menü-Baumes
        jsr     aes
        move.l  addr_out,menu_adr   ;Wert merken

        ; Anzeigen der Menüleiste (menu_bar):

        move.w  #30,control
        move.w  #1,control+2
        move.w  #1,control+4
        move.w  #1,control+6
        clr.w   control+8
        move.w  #1,int_in       ;1 für Menü zeigen
        move.l  menu_adr,addr_in ;Menüobjektbaum
        jsr     aes
```

```
        ; evnt_mesag-Warteschleife

loop:   move.w   #23,control
        clr.w    control+2
        move.w   #1,control+4
        move.w   #1,control+6
        clr.w    control+8
        move.l   #puffer,addr_in
        jsr      aes

        cmpi.w   #10,puffer       ;Menüpunkt angeklickt?
        bne      loop             ;nein, dann warten

        cmpi.w   #INFO,puffer+8   ;sonst: Info gewünscht?
        bne.s    no_info
        move.w   #1,d0            ;Default-Button
        lea.l    info_txt,a0
        jsr      form_alert       ;Alert-Box zeigen
        bra      tnormal          ;und Titel normal stellen
no_info: cmpi.w  #ENDE,puffer+8   ;oder Ende angeklickt?
        bne      loop             ;nein, dann warten
        move.w   #2,d0            ;Default-Button
        lea.l    ende_txt,a0      ;Sicherheitsabfrage
        jsr      form_alert
        cmpi.w   #1,d0            ;ja angeklickt?
        beq      ende             ;sonst Programm abschließen

tnormal: move.w  #33,control      ;menu_tnormal (zeigt Menütitel
        move.w   #2,control+2     ;normal, also nicht mehr
        move.w   #1,control+4     ;invertiert)
        move.w   #1,control+6
        clr.w    control+8
        move.w   puffer+6,int_in  ;Titelnummer (Teil der Nachricht)
        move.w   #1,int_in+2      ;1: Titel normal zeigen
        move.l   menu_adr,addr_in
        jsr      aes
        bra      loop             ;und auf den nächsten Klick warten


        ; Programmende:
        ; Menüzeile wieder löschen (menu_bar):

ende:   move.w   #30,control
        move.w   #1,control+2
        move.w   #1,control+4
        move.w   #1,control+6
        clr.w    control+8
        clr.w    int_in           ;0 für Menü löschen
        move.l   menu_adr,addr_in ;Menüobjektbaum
```

```
          jsr       aes

          ; Resource-File aus dem Speicher entfernen (rsrc_free)

          move.w    #111,control
          clr.w     control+2
          move.w    #1,control+4
          clr.w     control+6
          clr.w     control+8
          jsr       aes

quit:     jsr       gem_exit

          clr.w     -(sp)
          trap      #gemdos


rsc_error:
          move.w    #1,d0           ;Default-Button
          lea.l     err_txt,a0
          jsr       form_alert      ;Fehlermeldung
          bra       quit            ;und Programm abbrechen

form_alert:
          ; Zeigt Alert-Box.
          ; Parameter:
          ;   d0: Default-Button (0..3)
          ;   a0: String, der die Box beschreibt [.][...][...]
          ; Ausgabe:
          ;   d0: angeklickter Knopf (0..3)

          move.w    #52,control     ;Opcode form_alert
          move.w    #1,control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          move.w    d0,int_in       ;Default-Button
          move.l    a0,addr_in      ;String
          jsr       aes
          move.w    int_out,d0      ;angeklickter Knopf
          rts


          .DATA

info_txt: .DC.b "[1][Menü-Demoprogramm|(c) 1989 Data Becker GmbH]"
          .DC.b "[Weiter]",0

ende_txt: .DC.b "[2][Wirklich beenden?][Ja|Nein]",0
```

```
err_txt:  .DC.b "[3][Kein Resource-File!][Abbruch]",0

rsc_name: .DC.b "MENU1.RSC",0

          .BSS

menu_adr: .DS.l 1

puffer:   .DS.w 8

          .END
```

Sie haben gesehen, wie einfach es ist, eine kleine Menüleiste nicht nur am Bildschirm darzustellen, sondern auch auf die Wahl des Anwenders einzugehen, und bei größeren Menüs mit mehr Einträgen wird die Sache nicht komplizierter. Die Menüs sind überhaupt eines der schönsten Beispiele dafür, wie einfach sich komplizierte und langweilige Standard-Dinge unter GEM programmieren lassen. Sie haben ferner sehen können, daß die Accessories (falls geladen) im Desk-Menü vertreten waren und auch funktionierten; das heißt, wir müssen uns um die gar nicht kümmern; GEM regelt die Angelegenheit sehr gut und für uns praktisch. Das einzige, was Sie in diesem Zusammenhang beachten müssen, ist folgendes: Selbst, wenn Sie nur ein Fenster offen haben, so müssen Sie, wenn Sie eine Menüleiste verwenden wollen, eine komplette Redraw-Routine für dieses Fenster installieren (siehe 5.8.2), weil ein Accessory jederzeit das zweite Window öffnen könnte.

So, was gibt es zu den Menüs noch zu sagen? Es gibt aktive und nicht aktive Menüpunkte. Ein Menüpunkt ist genau dann aktiv, wenn er im Menü nicht heller als die anderen Einträge erscheint. Ein heller oder grauer Eintrag wird nicht invertiert, wenn der Mauszeiger ihn berührt, und kann auch nicht angeklickt werden (kann schon, doch erhält das Programm keine Nachricht, und evnt_mesag wird nicht beendet). Es ist immer dann sinnvoll, einen Menüpunkt inaktiv zu machen, wenn die augenblickliche Situation es nicht erlaubt, diesen Punkt zu bearbeiten. So sollte ein Malprogramm alle Menüpunkte, die mit Farben zu tun haben, inaktiv machen, wenn es in der hohen Auflösung auf einem monochromen Monitor gefahren wird. Das Umschalten zwischen aktiv und nicht aktiv geschieht über die Funktion menu_ienable (Entry-Enable, Eintrag anklickbar machen). Als Parameter müssen Sie den Index (die Nummer) des Eintrages angeben, der verändert werden soll. Die Angabe des Titels dieses Eintrages ist nicht erforderlich.Eine zweite Möglichkeit, einzelne Einträge zu verändern, besteht im sogenannten Check-Mark. Das ist ein Häkchen, das links neben einem Menü-Eintrag erscheinen kann. Es wird gewöhnlich dann gesetzt, wenn das, was der Menüpunkt bewirkt, bereits

eingestellt ist. Beispiel: Zwei Einträge lauten Anfänger und Fortgeschrittener, dann sollte der Eintrag 'Anfänger' ein Häkchen erhalten, wenn das Programm mit einem niedrigen Schwierigkeitsgrad spielt. Die Funktion, die einen Haken löscht oder setzt, heißt menu_ickeck. Deren Anwendung ist so einfach, daß ich dafür kein eigenes Beispielprogramm vorgeben möchte. Gleiches gilt auch für menu_ienable.

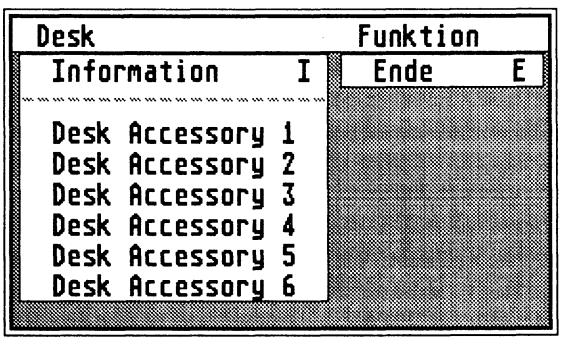
Etwas anderes möchte ich Ihnen aber noch zeigen: Da es oft nötig ist, die Hand von der Tastatur zur Maus und zurück zu bewegen, um einen Menüpunkt anklicken zu können, ziehen es viele Anwender vor, Ihre Befehle dem Programm nur über die Tastatur mitzuteilen. Anwenderfreundliche Programme sollten es also dem Benutzer überlassen, ob er einen Menüpunkt mit der Maus anklickt oder ob er die gleiche Aktion durch einen Tastendruck auslöst.

Die Wahl, welche Taste welchem Menüpunkt entsprechen soll, ist Ihnen überlassen. Dabei ist selbstverständlich zu beachten, daß die Menüpunkte Directory und Drucken nicht beide mit dem D aufgerufen werden können. Außerdem müssen Sie wählen, ob die Menüs über normale Buchstaben oder in Verbindung mit der Control-Taste aufgerufen werden sollen. Letzteres ist immer dann notwendig, wenn die normalen Buchstaben andere Funktionen haben (zum Beispiel in einer Textverarbeitung).

Die Taste, die optional einen Menüpunkt auslösen soll, wird gewöhnlich mit in das Menü geschrieben, und zwar rechts neben den Eintrag. Wenn zusätzlich zu der Taste auch noch Control gedrückt werden muß, dann schreibt man ein Potenzzeichen oder Dach (^) direkt vor den Buchstaben. Das kann schon bei der Erstellung des Resource-Files gemacht werden.

Im Programm selbst wird dann statt des evnt_mesag-Aufrufs die Funktion evnt_multi aufgerufen, die auf Ereignisse (Nachrichten) und Tastendrücke warten soll. Wenn wir eine Nachricht erhalten, verfahren wir wie gewöhnlich. Erhalten wir einen Tastendruck, dann wird die gewünschte Routine aufgerufen. Es ist aber üblich, vor diesem Aufruf den Menütitel des zugehörigen Eintrags zu invertieren, damit der Anwender wenigstens ungefähr weiß, ob er die richtige Taste getroffen hat. Das geht mit der Funktion menu_tnormal, die ja auch dazu benutzt wird, das Invertieren wieder rückgängig zu machen. Allerdings müssen Sie dazu wissen, unter welchem Titel der Eintrag steht (menu_tnormal erwartet ja die Nummer dieses Titels als Parameter). Deshalb geben wir also ab jetzt nicht nur den Menü-Einträgen im Resource-Construction-Set Namen, sondern auch den Titeln.

Um beide Verfahren (mit und ohne Tastendruck) besser vergleichen zu können, nehmen wir doch für unser Beispielprogramm wieder die gleiche Menüleiste. Die Titel nennen wir DESK und FUNKTION; die Einträge behalten ihre alten Namen. Information soll mit I, Ende mit E aufgerufen werden können. Die Menüleiste sieht dann so aus:



Die Header Datei MENU2.H (bzw. MENU2.H2 für BASIC und Assembler) besitzt dann natürlich zwei neue Konstanten. An diesem Programm läßt sich auch sehr schön demonstrieren, was ich im Kapitel 5.8.2 (mehrere Fenster) schon einmal erwähnt habe: Wenn Sie das Kontrollfeld öffnen, so können Sie beide Menüpunkte (Ende und Information) zwar weiterhin per Maus anwählen; die Tastenbefehle I und E funktionieren jedoch nicht, weil das Kontrollfeld ein Fenster geöffnet hat, das gerade aktiv, also im Vordergrund ist. Alle Tastendrücke werden deshalb nur an das Kontrollfeld-Programme weitergereicht. Erst, wenn das Kontrollfeld geschlossen wird oder eines unserer Fenster in den Vordergrund gebracht wird (was hier nicht geht, weil unser Programm keine Fenster hat), können Sie die Tastenbefehle wieder benutzen.

Das Programm sieht dem ersten recht ähnlich. Unterschiedlich sind lediglich die Ereignisabfrage (evnt_multi erfordert etwas mehr Aufwand) und natürlich die Ereignisauswertung. Außerdem habe ich die wirklichen Menüfunktionen in Unterprogramme gepackt. Eine letzte Information: Das Ergebnis von evnt_keybd (erhalten wir jetzt also auch bei evnt_multi) ist ein Wort, dessen unteres Byte den ASCII-Code der gedrückten Taste enthält (bzw. 0 bei Cursor- und Funktionstasten), während im hochwertigen Byte (Bits 8 bis 15) der Scan-Code zu finden ist, mit dem man z.B. unterscheiden kann, ob eine Ziffer des Ziffernblocks oder des Hauptblocks gedrückt wurde. Nur: Diese Information ist für uns ohne Belang; uns genügt der ASCII-Code der Taste. Wir blenden dazu

das obere Wort aus (AND-Verknüpfung). Damit das Programm gleichermaßen auf Groß- und Kleinbuchstaben reagieren kann, was in unserem Fall sinnvoll ist, wird der Buchstabe vor der Abfrage in einen Großbuchstaben umgewandelt.

GFA-BASIC

```
'
' Aktivieren von Menüs per Maus und per Tastatur
' GFA-BASIC        MP  11-12-88        MENU2.GFA
'
' Für rsrc_load muß Speicher an GEMDOS zurückgegeben werden:
'
RESERVE -33000   ! Das muß reichen, da Resource-Files
'                  nicht größer als 32K werden
'
menue=0          ! Konstanten wie gehabt, aber zwei neue für die
desk=3           ! Menütitel
funktion=4
info=7
ende=16
'
puffer$=SPACE$(16)   ! Platz für 8 Worte
'
DEFFN p(x)=CVI(MID$(puffer$,x*2+1,2))  ! Funktion holt
'                                        Wort Nr. x aus dem Puffer
'
VOID APPL_INIT()
'
' Laden des Resource-Files (MENU1.RSC):
'
IF RSRC_LOAD("MENU2.RSC")=0
  VOID FORM_ALERT(1,"[3][Kein Resource-File!][Abbruch]")
ELSE
  '
  ' Adresse des Menü-Objektbaumes erfragen (0 für Baum):
  '
  VOID RSRC_GADDR(0,menue,menue_adresse%)
  '
  ' Anzeigen (1) der Menüleiste:
  '
  VOID MENU_BAR(menue_adresse%,1)
  '
  ' Ereignisschleife:
  '
  abbruch%=0
  '
  REPEAT
    ' evnt_multi:  17 = evnt_keybd + evnt_mesag
```

```
    '              d% ist dummy-Variable
    w%=EVNT_MULTI(17,0,0,0,0,0,0,0,0,0,0,0,0,0,VARPTR(puffer$),0,d%,...
                              ...d%,d%,d%,key%,d%)
    '
    IF (w% AND 16) AND (FN p(0)=10)  ! Menüpunkt angeklickt?
      title%=FN p(3)                 ! Nummer des Titels merken
      IF FN p(4)=info                ! Information gewünscht?
        sub_info
      ENDIF
      IF FN p(4)=ende                ! oder Programmende?
        sub_ende
      ENDIF
      '
      ' Menütitel normal (1) stellen:
      '
      VOID MENU_TNORMAL(menue_adresse%,title%,1)
    ENDIF
    '
    IF w% AND 1                            ! Taste gedrückt?
      taste$=UPPER$(CHR$(key% AND 255))    ! nur ASCII-Code
      IF taste$="I"                        ! (niederwertiges Byte)
        ' Menütitel invers
        VOID MENU_TNORMAL(menue_adresse%,desk,0)
        ' Funktion aufrufen:
        sub_info
        ' Menütitel wieder normal:
        VOID MENU_TNORMAL(menue_adresse%,desk,1)
      ENDIF
      IF taste$="E"
        VOID MENU_TNORMAL(menue_adresse%,funktion,0)
        sub_ende
        VOID MENU_TNORMAL(menue_adresse%,funktion,1)
      ENDIF
    ENDIF
  UNTIL abbruch%=1
  '
  VOID MENU_BAR(menue_adresse%,0)  ! Menüzeile löschen
  VOID RSRC_FREE()                 ! Platz freigeben
ENDIF
'
RESERVE
VOID APPL_EXIT()
'
END
'
'
PROCEDURE sub_info
  VOID FORM_ALERT(1,"[1][Menü-Demoprogramm|(c) 1989 Data Becker...
                        ... GmbH][Weiter]")
RETURN
```

```
'
PROCEDURE sub_ende
  IF FORM_ALERT(2,"[2][Wirklich beenden?][Ja|Nein]")=1
    abbruch%=1
  ENDIF
RETURN
'
```

Omikron-BASIC

```
'
' Anzeigen einer Menüleiste, Funktionen auf Tastendruck
' Omikron-BASIC          MP  11-12-88         MENU2.BAS
'
' Für rsrc_load muß Speicher an GEMDOS zurückgegeben werden:
'
CLEAR 33000'    Das muß reichen, da Resource-Files
'               nicht größer als 32K werden
'
Menue=0'        Kennen Sie schon, jetzt aber aus MENU2.H2
Desk=3'         (mit Titelnummern)
Funktion=4
Info=7
Ende=16
'
Puffer$= SPACE$(16)' Platz für 8 Worte
'
DEF FN P(X)= CVI( MID$(Puffer$,X*2+1,2))'  Funktion holt
'                                          Wort Nr. x aus dem Puffer
'
Appl_Init
'
' Laden des Resource-Files (MENU1.RSC):
'
Rsrc_Load("MENU2.RSC",Back)
IF Back=0 THEN
 FORM_ALERT (1,"[3][Kein Resource-File!][Abbruch]")
ELSE
 '
 ' Adresse des Menü-Objektbaumes erfragen (0 für Baum):
 '
 Rsrc_Gaddr(0,Menue,Menue_Adresse)
 '
 ' Anzeigen der Menüleiste:
 '
 Menu_Bar(Menue_Adresse)
 '
 ' Ereignisschleife:
 '
 Abbruch=0
```

```
'
REPEAT
  Evnt_Multi(17,0,0,0,0,0,0,0,0,0,0,0,0,0,0,Puffer$,W,D,D,D,...
                        ...D,K,D)
  '
  IF (W AND 16) AND (FN P(0)=10) THEN '       Menüpunkt angeklickt?
    Title=FN P(3)'                          Nummer des Titels merken
    IF FN P(4)=Info THEN '                    Information gewünscht?
      Sub_Info
    ENDIF
    IF FN P(4)=Ende THEN '                          oder Programmende?
      Sub_Ende
    ENDIF
    '
    ' Menütitel normal (1) stellen:
    '
    Menu_Tnormal(Title,1)
  ENDIF
  '
  IF (W AND 1) THEN '                    Tastendruck
    Taste$= UPPER$( CHR$(K AND 255))'  Zeichen steht im Low-Byte
    IF Taste$="I" THEN
      Menu_Tnormal(Desk,0)' Menütitel invertieren
      Sub_Info
      Menu_Tnormal(Desk,1)' und wieder normal
    ENDIF
    '
    IF Taste$="E" THEN
      Menu_Tnormal(Funktion,0)
      Sub_Ende
      Menu_Tnormal(Funktion,1)
    ENDIF
  ENDIF
UNTIL Abbruch=1
'
Menu_Bar(Menue_Adresse)'    Menüzeile löschen
Rsrc_Free' Platz freigeben
ENDIF
'
Appl_Exit
'
END
'
DEF PROC Sub_Info
  FORM_ALERT (1,"[1][Menü-Demoprogramm¦(c) 1989 Data Becker GmbH]...
                            ...[Weiter]")
RETURN
'
DEF PROC Sub_Ende
  FORM_ALERT (2,"[2][Wirklich beenden?][Ja¦Nein]",Back)
```

```
    IF Back=1 THEN Abbruch=1
RETURN
'
```

C

```
/*******************************************************/
/*   Anzeigen einer Menüleiste/Menüpunkt über Taste    */
/*   Megamax Laser C     MP  11-12-88       MENU2.C    */
/*******************************************************/

#include "menu2.h"    /* Konstanten des RSC-Files */
/* Die beiden neuen Konstanten heißen DESK und FUNKTION */

int  puffer[8],
     key,
     dummy,
     title,
     abbruch = 0,
     which;
long menue_adresse;


void sub_info()
{
  form_alert (1,
      "[1][Menü-Demoprogramm¦(c) 1989 Data Becker GmbH][Weiter]");
}


void sub_ende()
{
 if (form_alert (2, "[2][Wirklich beenden?][Ja¦Nein]") == 1)
   abbruch = 1;
}


main()
{
  appl_init();
  graf_mouse (0, 0L);   /* Pfeil als Mauszeiger */

  /* Resource-File laden */

  if (rsrc_load ("MENU2.RSC") == 0)  /* Fehler beim Laden? */
    form_alert (1, "[3][Kein Resource-File!][Abbruch]");
  else
  {
    /* Adresse des Menübaumes beschaffen */
```

```
rsrc_gaddr (0, MENUE, &menue_adresse);

menu_bar (menue_adresse, 1);  /* Menüleiste zeigen */

/* Ereignis-Warteschleife: */

do
{
  which = evnt_multi (17, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          puffer, 0, 0, &dummy, &dummy, &dummy, &dummy,
          &key, &dummy);

  if ((puffer[0] == 10) &&
     (which && 16))         /* Menüpunkt angeklickt? */
  {
    title = puffer[3];      /* für später merken */

    switch (puffer[4])
    {
      case INFO:  sub_info();
                  break;

      case ENDE:  sub_ende();
                  break;
    }
    menu_tnormal (menue_adresse, title, 1);
  }

  if (which && 1)       /* Taste gedrückt */
  {
    key &= 255;         /* Nur Low-Byte berücksichtigen */
    key = key > 'Z' ? key-('z'-'Z') : key;  /* Großbuchstabe */
    switch (key)
    {
      case 'I': menu_tnormal (menue_adresse, DESK, 0);
                sub_info();
                menu_tnormal (menue_adresse, DESK, 1);
                break;

     case 'E':  menu_tnormal (menue_adresse, FUNKTION, 0);
                sub_ende();
                menu_tnormal (menue_adresse, FUNKTION, 1);
                break;
     }
  }
}
while (!abbruch);

menu_bar (menue_adresse, 0);  /* Menü wieder löschen */
rsrc_free();
```

```
  }
  appl_exit();
}
```

Assembler

```
          ;
          ; Menüleiste anzeigen, Funktionen auf Tasten
          ; Assembler       MP  11-12-88       MENU2.Q
          ;

          INCLUDE 'GEM_INEX.Q'

gemdos    = 1

MENUE     = 0       ;Konstanten aus MENU2.H2
DESK      = 3
FUNKTION  = 4
INFO      = 7
ENDE      = 16

          TEXT

main:     jsr       gem_init

          ; Pfeil als Mauszeiger:

          move.w    #78,control      ;graf_mouse
          move.w    #1,control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          clr.w     int_in           ;0 für Pfeil
          jsr       aes

          ; rsrc_load:

          move.w    #110,control     ;Opcode
          clr.w     control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          move.l    #rsc_name,addr_in
          jsr       aes

          tst.w     int_out          ;Fehler aufgetreten?
          beq       rsc_error

          ; Adresse des Menü-Objektbaums feststellen (rsrc gaddr)
```

```
        move.w    #112,control
        move.w    #2,control+2
        move.w    #1,control+4
        clr.w     control+6
        move.w    #1,control+8
        clr.w     int_in          ;0 -> wir suchen einen Baum
        move.w    #MENUE,int_in+2 ;Index des Menü-Baumes
        jsr       aes
        move.l    addr_out,menu_adr;Wert merken

        ; Anzeigen der Menüleiste (menu_bar):

        move.w    #30,control
        move.w    #1,control+2
        move.w    #1,control+4
        move.w    #1,control+6
        clr.w     control+8
        move.w    #1,int_in       ;1 für Menü zeigen
        move.l    menu_adr,addr_in ;Menüobjektbaum
        jsr       aes

        ; evnt_multi-Warteschleife

loop:   move.w    #25,control     ;evnt_multi
        move.w    #16,control+2   ;(in Assembler relativ kurz!)
        move.w    #7,control+4
        move.w    #1,control+6
        clr.w     control+8
        move.w    #17,int_in      ;evnt_mesag + evnt_keybd
        move.l    #puffer,addr_in
        jsr       aes             ;int_out: 1 (Taste) / 16 (Menü)
        move.w    int_out,d7      ;für später aufheben

        btst      #4,d7           ;Nachricht erhalten?
        beq       no_menu

        cmpi.w    #10,puffer      ;wirklich Menüpunkt angeklickt?
        bne       loop            ;nein, dann warten

        cmpi.w    #INFO,puffer+8  ;sonst: Info gewünscht?
        bne.s     no_info
        move.w    #1,d0           ;Default-Button
        lea       info_txt,a0
        jsr       form_alert      ;Alert-Box zeigen
        bra       tnormal         ;und Titel normal stellen

no_info: cmpi.w   #ENDE,puffer+8  ;oder Ende angeklickt?
        bne       loop            ;nein, dann warten
        move.w    #2,d0           ;Default-Button
        lea       ende_txt,a0     ;Sicherheitsabfrage
```

```
          jsr       form_alert
          cmpi.w    #1,d0             ;ja angeklickt?
          beq       fine              ;sonst Programm abschließen

tnormal:  move.w    #33,control       ;menu_tnormal (zeigt Menütitel)
          move.w    #2,control+2      ;normal, also nicht mehr
          move.w    #1,control+4      ;invertiert)
          move.w    #1,control+6
          clr.w     control+8
          move.w    puffer+6,int_in   ;Titelnummer (Teil der Nachricht)
          move.w    #1,int_in+2       ;1: Titel normal zeigen
          move.l    menu_adr,addr_in
          jsr       aes
          bra       loop              ;und auf den nächsten Klick warten

no_menu:  btst      #0,d7             ;Taste gedrückt?
          beq       loop
          move.w    int_out+10,d0     ;betätigte Taste
          andi.w    #$ff,d0           ;nur Low-Byte betrachten
          cmpi.b    #'Z',d0           ;Kleinbuchstabe
          ble       gross
          subi.b    #'z'-'Z',d0       ;ja, dann Groß machen

gross:    cmpi.b    #'I',d0           ;Information gefragt?
          bne       no_info2

          ; Menütitel invertieren:

          move.w    #33,control       ;menu_tnormal
          move.w    #2,control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          move.w    #DESK,int_in      ;Titelnummer
          clr.w     int_in+2          ;0: Titel invers zeigen
          move.l    menu_adr,addr_in
          jsr       aes

          move.w    #1,d0             ;Default-Button
          lea       info_txt,a0
          jsr       form_alert        ;Alert-Box zeigen

          move.w    #33,control       ;menu_tnormal
          move.w    #2,control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          move.w    #DESK,int_in      ;Titelnummer
          move.w    #1,int_in+2       ;1: Titel normal zeigen
          move.l    menu_adr,addr_in
```

```
          jsr       aes
          bra       loop

no_info2: cmpi.b    #'E',d0
          bne       loop
          move.w    #33,control       ;menu_tnormal
          move.w    #2,control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          move.w    #FUNKTION,int_in ;Titelnummer
          clr.w     int_in+2          ;0: Titel invers zeigen
          move.l    menu_adr,addr_in
          jsr       aes

          move.w    #2,d0             ;Default-Button
          lea       ende_txt,a0       ;Sicherheitsabfrage
          jsr       form_alert
          cmpi.w    #1,d0             ;ja angeklickt?
          beq       fine              ;sonst Programm abschließen

          move.w    #33,control       ;menu_tnormal
          move.w    #2,control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          move.w    #FUNKTION,int_in ;Titelnummer
          move.w    #1,int_in+2       ;1: Titel normal zeigen
          move.l    menu_adr,addr_in
          jsr       aes
          bra       loop


          ; Programmende:
          ; Menüzeile wieder löschen (menu_bar):

fine:     move.w    #30,control
          move.w    #1,control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          clr.w     int_in            ;0 für Menü löschen
          move.l    menu_adr,addr_in ;Menüobjektbaum
          jsr       aes

          ; Resource-File aus dem Speicher entfernen (rsrc_free)

          move.w    #111,control
          clr.w     control+2
          move.w    #1,control+4
```

```
          clr.w     control+6
          clr.w     control+8
          jsr       aes

quit:     jsr       gem_exit

          clr.w     -(sp)
          trap      #gemdos

rsc_error:
          move.w    #1,d0           ;Default-Button
          lea       err_txt,a0
          jsr       form_alert      ;Fehlermeldung
          bra       quit            ;und Programm abbrechen

form_alert:
          ; Zeigt Alert-Box.
          ; Parameter:
          ;   d0: Default-Button (0..3)
          ;   a0: String, der die Box beschreibt [.][...][...]
          ; Ausgabe:
          ;   d0: angeklickter Knopf (0..3)

          move.w    #52,control     ;Opcode form_alert
          move.w    #1,control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          move.w    d0,int_in       ;Default-Button
          move.l    a0,addr_in      ;String
          jsr       aes
          move.w    int_out,d0      ;angeklickter Knopf
          rts

          DATA

info_txt: DC.b "[1][Menü-Demoprogramm|(c) 1989 DATA BECKER GmbH]"
          DC.b "[Weiter]",0

ende_txt: DC.b "[2][Wirklich beenden?][Ja|Nein]",0

err_txt:  DC.b "[3][Kein Resource-File!][Abbruch]",0

rsc_name: DC.b "MENU2.RSC",0

          BSS

menu_adr: DS.l 1
puffer:   DS.w 8
          END
```

5.9.2 Dialoge

Jeder kennt sie, und jeder hält sie für praktisch: die Dialogboxen. Sie sind neben den Fenstern und den Menüleisten das dritte Erkennungszeichen für GEM-Applikationen. So stehen Ihnen am Ende dieses Kapitels alle Möglichkeiten zur Verfügung, selbst zur Tastatur zu greifen und eine echte GEM-Applikation zu programmieren.

Vor die Dialogboxen haben die Götter allerdings die Objektbäume gestellt. Diese Bäume wurden schon einmal im vorigen Abschnitt über Menüs erwähnt, aber nicht richtig erklärt; das war zum Verständnis der Menüs auch nicht erforderlich. Sie haben jetzt die Wahl: Entweder holen Sie den Stoff nach, was langweilig wird; oder Sie überschlagen die nächsten Seiten, dann müssen Sie auf meine Unterprogramme in den Beispielapplikationen vertrauen, die alles Wichtige automatisch erledigen.

Gut, ich habe Sie gewarnt; Objektbäume sind ein trockenes Thema. Dafür dürfen Sie sich bald zu den wenigen zählen, die sich in dieser Materie wirklich auskennen. Ich werde hier nicht alle Einzelheiten ausleuchten, das ist Sache des ersten Intern-Bandes; doch Sie erhalten alle wesentlichen Informationen.

Was genau ist eigentlich ein Objektbaum? Das ist eine Datenstruktur, mit deren Hilfe logisch zusammengehörende Objekte (gemeint sind grafische Objekte) im Speicher dargestellt werden, d.h. wir betrachten hier die Objekte einer Dialogbox, nachdem sie mit rsrc_load in den Speicher geladen wurde. Logisch zusammengehörend sind alle Objekte eines Resource-Files dann, wenn sie zur gleichen Dialogbox (oder auch zur gleichen Menüleiste) gehören; diese Objekte zusammen bilden dann einen Objektbaum.

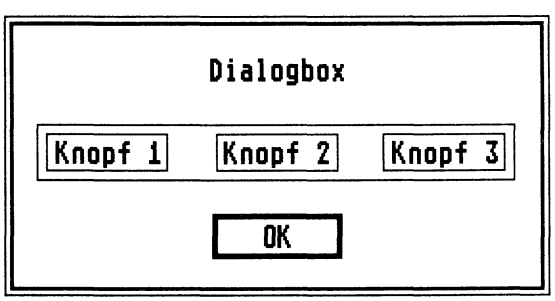
Wichtig für uns als Programmierer ist es, die Startadresse eines solchen Objektbaums herauszufinden. Das geht sehr einfach über die Funktion rsrc_gaddr, die wir ja vorhin schon benutzt haben. Diese Funktion kann selbstverständlich erst dann aufgerufen werden, wenn mit rsrc_load bereits Objekt-Bäume in den Speicher geladen wurden. Zu übergeben sind der Typ des Dings, das wir suchen; in unserem Fall ist das eine Null, die besagt, daß wir die Startadresse eines ganzen Objektbaumes haben möchten (Startadressen von anderen Dingen werden kaum benötigt). Zum andern ist der Index des Baumes anzugeben, dessen Adresse wir brauchen; ein Resource-File kann schließlich mehrere Objektbäume ent-

halten. Den Index bekommen Sie (wie schon bei den Menüleisten erwähnt) über die Name-Funktion des RCS, mit dem Sie den Objektbaum erstellt haben.

An dieser Startadresse finden wir nun alle Objekte, die zu dem entsprechenden Objektbaum gehören. Jedes Objekt belegt dabei einen Speicherplatz von 24 Bytes. Außerdem sind alle Objekte durchnumeriert; man sagt auch, die Objekte haben Indizes. Das Objekt mit dem Index Null liegt immer an der mit rsrc_gaddr erfragten Startadresse, das mit dem Index 1 liegt 24 Bytes dahinter usw. Auch die Indizes der einzelnen Objekte finden Sie im Header-File (.H oder .H2), wenn Sie Option Name des RCS benutzt haben.

Diese Objekte liegen nun zwar physikalisch hintereinander in einer Reihe im Speicher. Logisch gesehen gibt es aber eine Anordnung der Objekte, die nicht ihrer Reihenfolge im Speicher entspricht. Um das zu erreichen, sind die Objekte mit vielen Zeigern ausgestattet, die alle auf irgendwelche anderen Objekte zeigen, so daß eine richtige Verästelung der Objekte entsteht. So ist denn auch der Begriff des Objektbaums zu erklären.

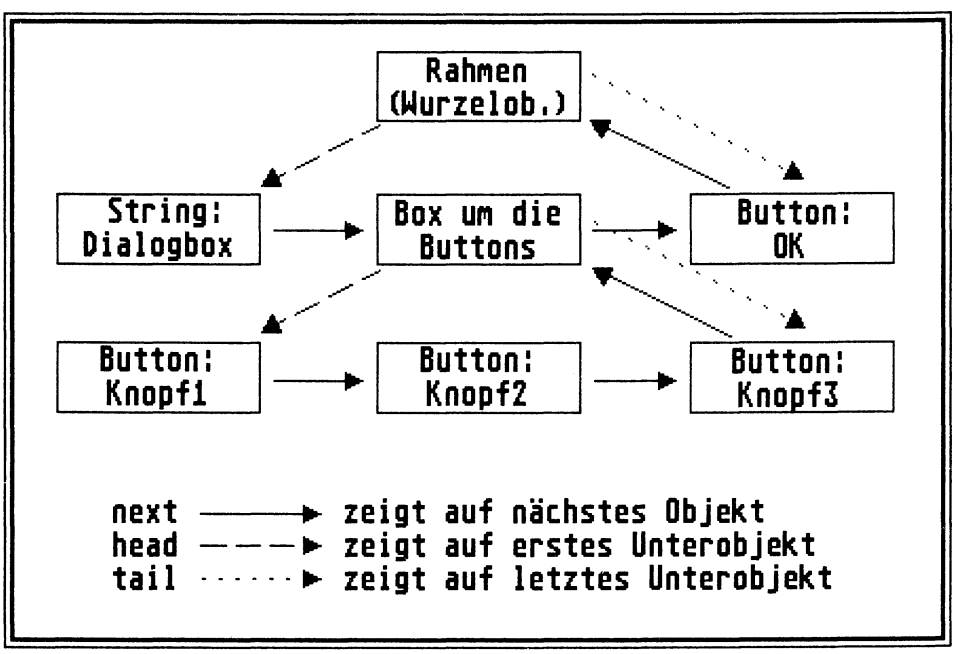
Betrachten wir dieses Zeigersystem einmal genauer: Jedes Objekt besteht im Speicher, das wurde gerade gesagt, aus 24 Bytes oder 12 Worten. Die ersten drei Worte eines Objekts sind Zeiger, die der internen Verwaltung dienen und den logischen Aufbau des Baumes erkennen lassen. Alle Zeiger zusammen beschreiben also die Anordnung der Objekte im Baum. Strenggenommen sind es übrigens gar keine Zeiger (die würden ja Langworte benötigen); in einem solchen Wort steht immer nur der Index des Objektes, auf das gezeigt werden soll. An dieser Stelle muß ein Beispiel her, um die weiteren Erläuterungen verständlich zu halten.



Diese kleine Dialogbox besteht aus folgenden Teilen: einem äußeren Rahmen, wie ihn jede Dialogbox besitzt, einem String (Dialogbox), einem Kasten (Box), der drei Knöpfe (Buttons) enthält, und einem Button, in

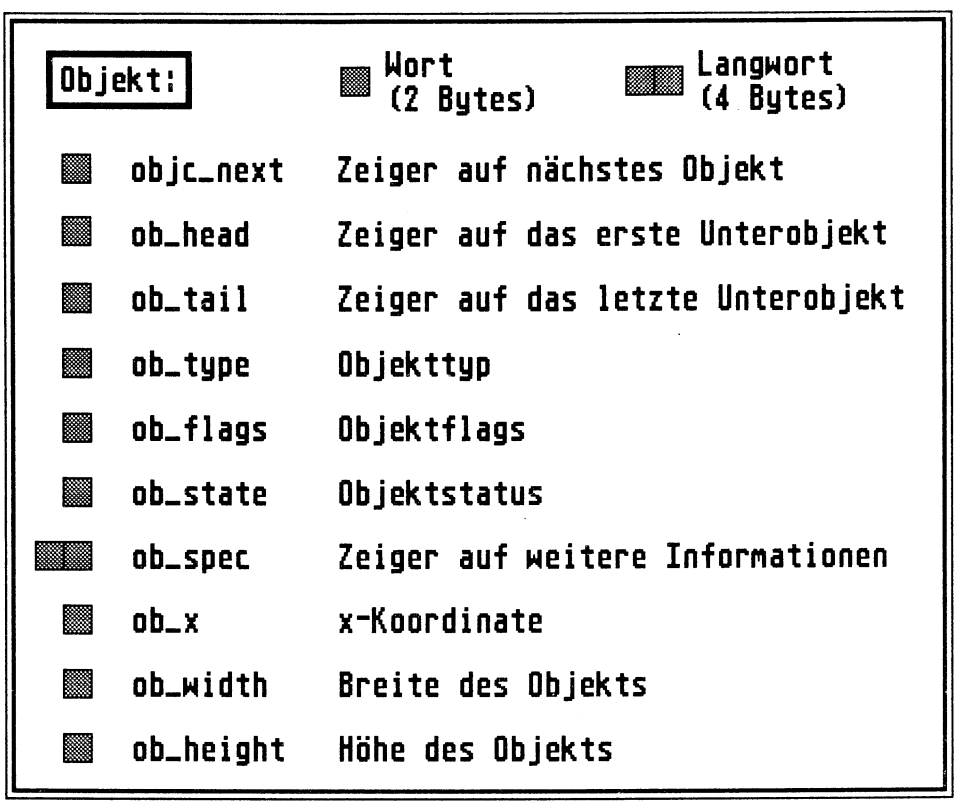
dem OK steht. Hier ist die Hierarchie der Objekte gut zu erkennen: Wir haben ein Objekt, das alle anderen umfaßt, nämlich den äußeren Rahmen. Wenn wir schon von Objekt-Bäumen reden, so bietet sich für dieses oberste Objekt die Bezeichnung Wurzelobjekt an. Drei Objekte sind direkte Unterobjekte dieses Rahmens, nämlich der String Dialogbox, der Button OK und die Box. Letztere enthält wiederum drei Unterobjekte: drei Buttons. Damit hätten wir auch gleich den Begriff des Unterobjekts erklärt: Ein Objekt ist einem anderen Objekt untergeordnet, wenn dieses andere Objekt das untergeordnete Objekt ganz umfaßt. Ein Objekt ist demnach anderen Objekten übergeordnet, wenn es die anderen Objekte ganz enthält.

Klären wir noch, was ein direktes Unterobjekt ist: Die Buttons Knopf1 bis Knopf3 aus unserer Beispielbox sind alle Unterobjekte des Wurzelobjekts, also des Rahmens, allerdings nicht unmittelbar; denn sie sind ja auch Unterobjekte der Box (und zwar direkte), die ihrerseits direktes Unterobjekt des Rahmens ist. Diese Box besteht demnach aus drei Hierarchie-Ebenen: Der Wurzel, den Unterobjekten String, Box und Button sowie den drei weiteren Unterobjekten (Buttons).



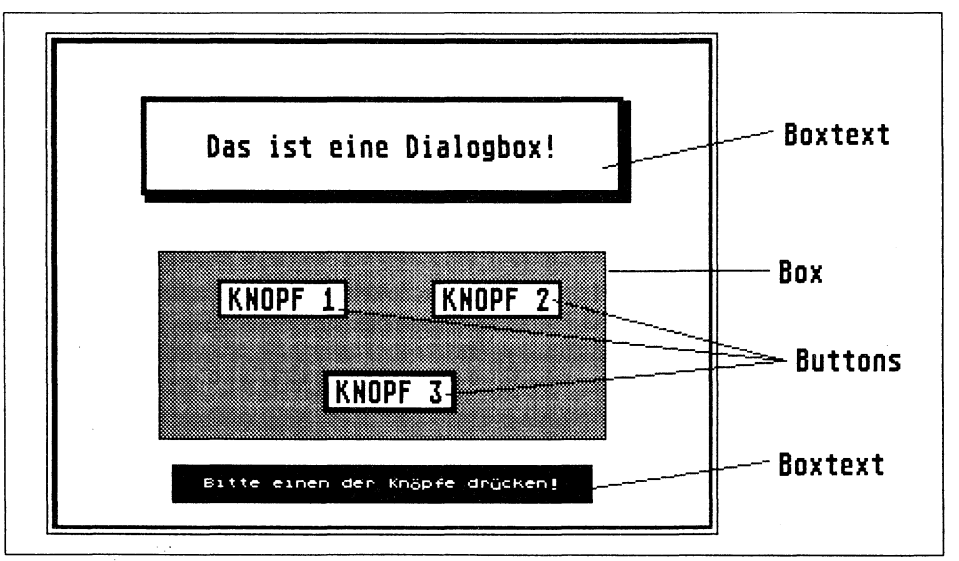
Zur Klärung dieser Anordnung: Wenn wir das System auf den Speicher übertragen, dann führen wir drei verschiedene Zeiger ein, mit denen Objekte auf andere Objekte zeigen: einen Zeiger namens next, der auf

das nächste Objekt der gleichen Ebene zeigen soll, einen namens head, der auf das erste der direkt untergeordneten Objekte zeigen soll, und einen namens tail, der auf das letzte der direkt untergeordneten Objekte zeigen soll. Wir vereinbaren ferner, daß der Next-Zeiger des letzten Objekts einer Ebene auf das direkt übergeordnete Element zurückzeigen soll. Die letzten Unklarheiten werden per Definiton beseitigt: Das Wurzelobjekt hat keine übergeordneten Objekte; deshalb nimmt dessen Next-Zeiger den Wert -1 an (den Objektindex -1 kann es ja nicht geben). Das gleiche gilt für Objekte, die keine Unterobjekte mehr haben: head und tail dieser Objekte werden ebenfalls auf -1 gesetzt. Nun ist es interessant, wie jedes Objekt im Speicher aussieht. Das zeigt folgendes Bild:



Die ersten drei Einträge erkennen Sie sicherlich wieder: Es sind die drei Zeiger next, head und tail. Wie ich bereits sagte, werden hier nicht wirklich Zeiger, sondern die Indizes der Objekte, auf die gezeigt werden soll, abgelegt. Das nächste Wort, ob_type, gibt an, von welchem Typ das Objekt ist (also z.B. Button, Box, Text, Icon...). Daran schließen sich die

Objektflags (ob_flags) an: In diesem Wort geben die einzelnen Bits Auskunft darüber, was mit dem Objekt alles geschehen kann, ob es z.B. angeklickt oder editiert werden kann. Das nächste Wort heißt ob_state und gibt den Status des Objekts wieder; hiermit kann z.B. geprüft werden, ob ein Button angeklickt (selektiert) ist oder nicht. Das einzige Langwort des Objekts, ob_spec, ist ein wirklicher Zeiger. Er zeigt nicht auf ein weiteres Objekt, sondern auf zusätzliche Daten, deren Bedeutung allerdings vom Objekttyp abhängig sind und daher nicht allgemein erklärt werden können. Schließlich folgen noch vier Worte für die Koordinaten, die relativ zum direkt übergeordneten Objekt angegeben sind. Beim Wurzelobjekt erfolgt diese Angabe relativ zum Bildschirm-Ursprung. Mehr dazu können Sie dem ersten Intern-Band entnehmen, der zum Beispiel die Objekttypen und die Bedeutung von ob_spec genau erläutert. Wir werden uns jetzt darum bemühen, eine solche Dialogbox auf den Bildschirm zu bekommen. Dazu werden wird ein Resource-File konstruieren und in den Speicher laden. Der erste Punkt, das Arbeiten mit einem RCS, ist schwierig zu erklären (darauf habe ich ja schon im Kapitel 5.9.1 hingewiesen); deshalb sollen hier nur allgemeine Informationen gegeben werden. Die Bedienung ihres RCS entnehmen Sie dann bitte dessen Bedienungsanleitung. Zunächst wollen wir eine Dialogbox kreieren, die außer Boxen und Strings, auf die der Benutzer keinen Einfluß hat, noch drei Buttons enthält, die angeklickt werden können. Das sähe also so aus:



Wenn Sie diesen Objektbaum mit einem RCS selbst erstellen möchten, dann sollten Sie schon jetzt beachten, welche Namen ich den Objekten

gegeben habe. Der ganze Baum heißt DIALOG (brauchen wir für rsrc_gaddr), und die Knöpfe hören auf die Konstanten KNOPF1, KNOPF2 und KNOPF3. Übrigens: In C gehört es sich, daß Bezeichner von Konstanten großgeschrieben werden; in Assembler mache ich das nur, wenn es sich um Konstanten von Objekt-Bäumen handelt, um sie von normalen Konstanten unterscheiden zu können. In BASIC geht das leider nicht.

Aus der obigen Abbildung können Sie auch erkennen, welche Objekt-Typen für die Dialogbox verwendet wurden. Was Sie nicht sehen können, ist, daß die Buttons die Eigenschaften (Objektflags) SELECTABLE und EXIT besitzen. Die erste Eigenschaft erhält automatisch jeder Button. Sie bewirkt, daß das entsprechende Objekt später selektiert (d.h. angewählt) oder deselektiert wird, wenn der Benutzer es mit dem Mauszeiger anklickt. Ein selektiertes Objekt erscheint dabei invertiert. Ein Button, der EXIT-Status besitzt, bewirkt zudem, daß die Abarbeitung des Formulars (dazu später mehr) beendet ist, sobald der Benutzer diesen Button angeklickt hat. Zusätzlich wurde für den Button KNOPF3 DEFAULT aktiviert. Die Bedeutung ist die gleiche wie in einer Alert-Box: Der Button, der diesen Status besitzt, gilt als angeklickt, wenn der Anwender die Return-Taste drückt. Es sollte klar sein, daß in jedem Objektbaum nur ein Button diesen Status haben kann. Falls Sie es noch nicht wissen: All diese Dinge können Sie in einem RCS einstellen, wenn Sie mit dem Mauszeiger auf das gewünschte Objekt fahren und doppelklicken.Wenn Sie den Objektbaum fertiggestellt haben, dann speichern Sie ihn ab (Sie können natürlich auch das RSC-File auf der Diskette im Buch nehmen). BASIC- und Assembler-Programmierer sollten dann das Programm BAS-HEAD.TOS benutzen, um eine Header-Datei (.H2) zu erstellen, die Sie in Ihrer Programmiersprache benutzen können (habe ich bereits auf der Diskette gemacht). Jetzt können wir uns an das Programm machen, das die Resource-Datei (sie soll übrigens DIALOG1.RSC heißen) benutzt.

Die ersten beiden Funktionen, die in diesem Zusammenhang benutzt werden müssen, kennen Sie bereits aus dem Beispiel der Menüleisten. Es sind rsrc_load und rsrc_gaddr, die genauso verwendet werden müssen, wie wir es auch bisher gemacht haben. Die nächsten Schritte sind vier Aufrufe von AES-Funktionen. Da diese immer zum Darstellen einer Dialogbox auf dem Bildschirm verwendet werden, habe ich sie in den folgenden Beispielprogrammen show_dialog zusammengefaßt. Es handelt sich um folgende Funktionen:

1. form_center

Diese Funktion muß mindestens einmal vor dem ersten Anzeigen der Dialogbox aufgerufen werden. Sie berechnet die Koordinaten (x, y; w und h stehen ja fest), an denen die Box erscheinen muß, um in der Mitte des Bildschirms zu stehen. Diese Koordinaten hängen von der momentanen Bildschirmauflösung ab und sind daher beim Erstellen des Resource-Files noch nicht bekannt. Zu übergeben sind die Startadresse des Objektbaums und vier Variablen, in denen die Koordinaten (x, y, w, h) zurückgegeben werden.

2. form_dial

Dies ist eine Zusammenfassung von vier Funktionen, die von 0 bis 3 durchnumeriert sind. Wir brauchen zunächst einmal die Unterfunktion Nr. 0. Diese meldet dem AES, daß wir, also die Applikation, vorhaben, einen bestimmten Bildschirmbereich, dessen Koordinaten zu übergeben sind, für eigene Zwecke zu benutzen. AES rettet dann Randelemente von Fenstern in einem Puffer, um sie wieder herstellen zu können, sobald die Dialogbox nicht mehr gebraucht wird. Parameter: Die Nummer der Unterfunktion (0) und die Koordinaten des Bereichs (haben wir von form_center erhalten) und zwar - es lebe die Bürokratie - in zweifacher Ausfertigung (also: form_dial (0,x,y,w,h,x,y,w,h)).

3. form_dial

Jetzt können wir auch gleich die Unterfunktion Nr. 1 von form_dial benutzen. Diese zeichnet ein größer werdendes Rechteck, auch Zoom-Rechteck genannt. Das sieht ganz nett aus. Zu übergeben sind die Unterfunktionsnummer (1), die Koordinaten des kleinen Rechtecks und die des großen Rechtecks. Letztere sind mit den Koordinaten von form_center identisch, während Sie das kleine Rechteck frei wählen können. Beispiele: linke obere Bildschirmecke, Bildschirmmitte, linke obere Ecke der (zukünftigen) Dialogbox usw. Die Routine zeichnet dann zuerst das kleine Rechteck und läßt es wachsen.

4. objc_draw

Nach dieser Menge Verwaltungs-Zoom-Theorie wird es nun Zeit, die Dialogbox auch wirklich zu zeichnen. Das erledigt die Funktion objc_draw. Als Parameter sind zu übergeben: die Adresse des Objektbaums, ein Rechteck (x, y, w, h), das als Zeichengrenze (Clipping) dient (gewöhnlich gibt man hier die Koordinaten von form_center an); ferner der Index des Objektes, mit dem die Zeichnung begonnen wird (gewöhnlich Null = Wurzelobjekt/Rahmen) sowie eine maximale Zeichentiefe.

Letztere gibt an, wie viele Ebenen von Unterobjekten zusätzlich zu dem ersten zu zeichnenden Objekt gezeichnet werden sollen, d.h. bei 0 wird nur das angegebene erste zu zeichnende Objekt ausgegeben, bei 1 auch dessen direkte Unterobjekte, bei 2 auch die direkten Unterobjekte dieser Unterobjekte usw. Ein Beispiel für die seltene Anwendung der beiden zuletzt genannten Parameter werden Sie im Beispielprogramm DIALOG4 finden.

Mit diesen vier Funktionsaufrufen steht nun die Dialogbox auf dem Bildschirm. Was jetzt noch fehlt, ist die Abarbeitung dieses Dialogs, also die Reaktion auf den Anwender. Aber auch das haben die GEM-Entwickler für uns schon vorbereitet. So genügt ein Aufruf der Routine form_do, um zu warten, bis der Anwender einen der drei Exit-Knöpfe betätigt hat. Der Index des Buttons, durch den die Verarbeitung der Box abgebrochen wurde, wird von form_do als Funktionswert zurückgegeben. Als Parameter werden die Adresse des Objektbaumes erwartet und der Index des Edit-Feldes, auf dem der Text-Cursor zuerst stehen soll. Da wir aber noch keine Edit-Felder in unserem Dialog haben, geben wir hier einfach eine Null an.

Wenn der Benutzer also einen Knopf gedrückt hat, dann kann die Dialogbox wieder verschwinden. Dazu dient mein Unterprogramm hide_dialog, das noch einmal die Koordinaten der Box ausrechnet (form_center) und die letzten beiden Unterfunktionen von form_dial (2 und 3) bemüht. Nummer 2 zeichnet ein sich verkleinerndes Rechteck, während Nummer 3 die Fensterränder und das graue Desktop wiederherstellt. Außerdem erhalten Fenster, die sich möglicherweise unter der Dialogbox befanden, eine Redraw-Meldung.

Eins gibt es noch zu beachten: Wenn der Benutzer einen Button anklickt, so wird dieser selektiert und erscheint invers. Soll die gleiche Dialogbox noch einmal verwendet werden (ohne das Programm neu zu laden), so muß dieser Knopf vorher wieder deselektiert werden. Das machen wir am besten gleich nach dem form_do-Aufruf.

Aber wie geht das eigentlich: Knöpfe selektieren und deselektieren? Das Prinzip sieht so aus: Wir berechnen die Startadresse des Objekts, das (de)selektiert werden soll. Diese Adresse ist die Startadresse des Objektbaums (erfahren wir durch rsrc_gaddr) + 24 * Index des Objektes, da ja jedes Objekt 24 Bytes = 12 Worte beansprucht. Zu dieser Zahl addieren wir noch einmal 10 (siehe Abbildung zur Objektstruktur) und erhalten die Adresse, an der der Objektstatus des betreffenden Objekts steht. In diesem Wort ist nun das Bit 0 zu setzen (selektieren) oder zu löschen (deselektieren). Alles klar? Sie sehen schon: Mit Dialogboxen zu arbeiten,

heißt, mit Zeigern und Adressen zu hantieren. Daß dies auch gute Möglichkeiten mit sich bringt, den Rechner abstürzen zu lassen, sei am Rande bemerkt...

In GFA-BASIC funktioniert dies im Prinzip genauso. Allerdings gibt es dort die Anweisungen:

```
OB_STATE(adresse,index)=...
```

bzw.

```
x=OB_STATE(adresse,index)
```

Damit ist es möglich, den Objektstatus zu lesen oder zu schreiben. adresse ist die Startadresse des Objektbaums und index der Index des betroffenen Objekts.

In C sieht die Sache wieder anders aus: Hier gibt es eine Standard-Header-Datei namens OBDEFS.H (OBject DEFinitionS), in der ein Struct OBJECT definiert wird.

```
typedef struct object {
 int  ob_next;
 int  ob_head;
 int  ob_tail;
 unsigned int ob_type;
 unsigned int ob_flags;
 unsigned int ob_state;
 char *ob_spec;
 int  ob_x;
 int  ob_y;
 int  ob_width;
 int  ob_height;
} OBJECT;
```

Wenn wir nun ein ARRAY mit Elementen vom Typ OBJECT bilden und diesem Array die Startadresse des Objektbaums geben, dann können wir durch

```
object[index].ob_state
```

bequem auf den Objektstatus zugreifen.

Das sollte als Erklärung für das erste Beispiel genügen.

GFA-BASIC

```
'
' Erste Dialogbox laden, anzeigen und verarbeiten
' GFA-BASIC        MP  16-12-88       DIALOG1.GFA
'
DEFINT "a-z"   ! alle Variablen 4-Byte-Integer
'
dialog=0       ! Konstanten aus DIALOG1.H2
knopf3=4
knopf1=5
knopf2=6
'
VOID APPL_INIT()
'
IF RSRC_LOAD("DIALOG1.RSC")=0
  VOID FORM_ALERT(1,"[3][Kein Rsc-File!][Ende]")
ELSE
  '
  ' Startadresse des (0 =) Baumes 'dialog' suchen:
  '
  VOID RSRC_GADDR(0,dialog,baum_adr)
  '
  ' Dialogbox anzeigen lassen:
  '
  GOSUB show_dialog(baum_adr)
  '
  ' Dialogbox abarbeiten lassen, d.h. auf 'Knopfdruck' warten
  '
  knopf=FORM_DO(baum_adr,0)      ! 0 = kein Edit-Feld
  '
  ' Status 'selected' des gewählten Knopfes aufheben:
  '
  GOSUB deselect(baum_adr,knopf)
  '
  ' Dialogbox wieder verschwinden lassen:
  '
  GOSUB hide_dialog(baum_adr)
  '
  ' Meldung an den Anwender, welcher Knopf gedrückt wurde:
  '
  SELECT knopf
  CASE knopf1
    nr=1
  CASE knopf2
    nr=2
  CASE knopf3
    nr=3
  ENDSELECT
  '
  a$="[1][Sie haben den Knopf Nr. "+STR$(nr)+" gedrückt!][Richtig!]"
```

```
  VOID FORM_ALERT(1,a$)
  '
  ' Resource-File aus dem Speicher entfernen:
  '
  VOID RSRC_FREE()
  '
ENDIF
'
VOID APPL_EXIT()
'
END
'
'
PROCEDURE select(baum,index)
  ' Bit 0 setzen:
  OB_STATE(baum,index)=OB_STATE(baum,index) OR 1
RETURN
'
'
PROCEDURE deselect(baum,index)
  ' Bit 0 löschen:
  OB_STATE(baum,index)=OB_STATE(baum,index) AND -2
RETURN
'
'
PROCEDURE show_dialog(baum)
  LOCAL x,y,w,h
  '
  ' Formular auf dem Bildschirm zentrieren (wird dabei noch
  ' nicht gezeichnet)
  '
  VOID FORM_CENTER(baum,x,y,w,h)
  '
  ' Fensterränder etc. retten lassen:
  '
  VOID FORM_DIAL(0,x,y,w,h,x,y,w,h)
  '
  ' Zeichnen eines 'Zoom'-Rechtecks:
  '
  VOID FORM_DIAL(1,x,y,1,1,x,y,w,h)
  '
  ' Zeichnen des Formulars:
  ' Start bei Objekt Nr. 0 (Wurzelobjekt, äußerer Kasten/Rahmen)
  ' Tiefe: max. 12 Ebenen (willkürlich)
  '
  VOID OBJC_DRAW(baum,0,12,x,y,w,h)
  '
RETURN
'
'
```

```
PROCEDURE hide_dialog(baum)
  LOCAL x,y,w,h
  '
  ' Nochmal die Koordinaten erfragen:
  '
  VOID FORM_CENTER(baum,x,y,w,h)
  '
  ' Zeichnen eines kleiner werdenden Rechtecks:
  '
  VOID FORM_DIAL(2,x,y,1,1,x,y,w,h)
  '
  ' Fensterränder wiederherstellen und Redraw-Meldung
  ' an alle zerstörten Fenster veranlassen:
  '
  VOID FORM_DIAL(3,x,y,w,h,x,y,w,h)
  '
RETURN
'

Omikron-BASIC:
'
' Erste Dialogbox laden, anzeigen und verarbeiten
' MIKRON-BASIC      MP  16-12-88      DIALOG1.BAS
'
Dialog%L=0'         Konstanten aus DIALOG1.H2
Knopf3%L=4
Knopf1%L=5
Knopf2%L=6
'
Appl_Init
'
Rsrc_Load("DIALOG1.RSC",Ret%L)
IF Ret%L=0 THEN
 FORM_ALERT (1,"[3][Kein Rsc-File!][Ende]",Dummy%L)
ELSE
 '
 ' Startadresse des (0 =) Baumes 'dialog' suchen:
 '
 Rsrc_Gaddr(0,Dialog%L,Baum_Adr%L)
 '
 ' Dialogbox anzeigen lassen:
 '
 Show_Dialog(Baum_Adr%L)
 '
 ' Dialogbox abarbeiten lassen, d.h. auf 'Knopfdruck' warten
 '
 Form_Do(0,Baum_Adr%L,Knopf%L)'          0 = kein Edit-Feld
 '
 ' Status 'selected' des gewählten Knopfes aufheben:
 '
```

```
 Deselect(Baum_Adr%L,Knopf%L)
 '
 ' Dialogbox wieder verschwinden lassen:
 '
 Hide_Dialog(Baum_Adr%L)
 '
 ' Meldung an den Anwender, welcher Knopf gedrückt wurde:
 '
 IF Knopf%L=Knopf1%L THEN Nr%L=1
 IF Knopf%L=Knopf2%L THEN Nr%L=2
 IF Knopf%L=Knopf3%L THEN Nr%L=3
 '
 A$="[1][Sie haben den Knopf Nr."+ STR$(Nr%L)+" gedrückt!][Richtig!]"
 FORM_ALERT (1,A$,Dummy%L)
 '
 ' Resource-File aus dem Speicher entfernen:
 '
 Rsrc_Free
 '
ENDIF
'
Appl_Exit
'
END
'
'
DEF PROC Select(Baum%L,Index%L)
 ' Bit 0 setzen:
 WPOKE Baum%L+24*Index%L+10, WPEEK(Baum%L+24*Index%L+10) OR 1
RETURN
'
'
DEF PROC Deselect(Baum%L,Index%L)
 ' Bit 0 löschen:
 WPOKE Baum%L+24*Index%L+10, WPEEK(Baum%L+24*Index%L+10) AND -2
RETURN
'
'
DEF PROC Show_Dialog(Baum%L)
 LOCAL X%L,Y%L,W%L,H%L
 '
 ' Formular auf dem Bildschirm zentrieren (wird dabei noch
 ' nicht gezeichnet)
 '
 Form_Center(Baum%L,X%L,Y%L,W%L,H%L)
 '
 ' Fensterränder etc. retten lassen:
 '
 Form_Dial(0,X%L,Y%L,W%L,H%L)
 '
```

```
 ' Zeichnen eines 'Zoom'-Rechtecks:
 '
 Form_Dial(1,X%L,Y%L,W%L,H%L)
 '
 ' Zeichnen des Formulars:
 ' Start bei Objekt Nr. 0 (Wurzelobjekt, äußerer Kasten/Rahmen)
 ' Tiefe: max. 12 Ebenen (willkürlich)
 '
 Objc_Draw(0,12,X%L,Y%L,W%L,H%L,Baum%L)
 '
RETURN
'
'
DEF PROC Hide_Dialog(Baum%L)
 LOCAL X%L,Y%L,W%L,H%L
 '
 ' Nochmal die Koordinaten erfragen:
 '
 Form_Center(Baum%L,X%L,Y%L,W%L,H%L)
 '
 ' Zeichnen eines kleiner werdenden Rechtecks:
 '
 Form_Dial(2,X%L,Y%L,W%L,H%L)
 '
 ' Fensterränder wiederherstellen und Redraw-Meldung
 ' an alle zerstörten Fenster veranlassen:
 '
 Form_Dial(3,X%L,Y%L,W%L,H%L)
 '
RETURN
'
```

C

```
/*******************************************************/
/*   Erste Dialogbox laden, anzeigen und verarbeiten   */
/*   Megamax Laser C     MP  14-11-88      DIALOG1.C   */
/*******************************************************/

#include <obdefs.h>     /* GEM-Objekt-Definitionen */

#include "gem_inex.c"
#include "dialog1.h"    /* Header-File der Resource-Datei */

OBJECT *baum_adr;       /* pointer auf das erste Onjekt eines Baumes */
int     knopf;
char    str[50],
        ziffer[2];
```

```
select (baum, index)          /* schaltet Button 'object' ein */
OBJECT baum[];
int    index;
{
  baum[index].ob_state |= 1;      /* Bit 0 in ob_state setzen */
}


deselect (baum, index)        /* schaltet Button 'object' aus */
OBJECT baum[];
int    index;
{
  baum[index].ob_state &= -2;   /* Bit 0 in ob_state löschen */
}


show_dialog (baum)              /* Darstellen einer Dialogbox */
OBJECT *baum;
{
int  x, y, w, h;

  /* Formular auf dem Bildschirm zentrieren. Dabei werden nur */
  /* Koordinaten intern an die Bildschirmauflösung angepaßt,  */
  /* es wird also noch nichts gezeichnet. Außerdem erhalten   */
  /* wir die zukünftigen Koordinaten der Dialogbox.           */

  form_center (baum, &x, &y, &w, &h);

  /* Fensterränder etc. retten lassen: */
  form_dial (0, x, y, w, h, x, y, w, h);

  /* Zeichnen eines 'Zoom'-Rechtecks */
  form_dial (1, x, y, 1, 1, x, y, w, h);

  /* Zeichnen des Objektbaumes selbst */
  /* Start bei Objekt Nr. 0 (Wurzel, äußerer Kasten) */
  /* Tiefe: max. 12 Ebenen (willkürlich) */
  objc_draw (baum, 0, 12, x, y, w, h);
}


hide_dialog (baum)
OBJECT *baum;
{
int  x, y, w, h;

  /* Nochmal die Koordinaten erfragen: */
  form_center (baum, &x, &y, &w, &h);

  /* Zeichnen eines kleiner werdenden Rechtecks */
```

```
  form_dial (2, x, y, 1, 1, x, y, w, h);

  /* Fensterränder wieder herstellen lassen und Redraw- */
  /* Meldung an alle zerstörten Fenster veranlassen     */
  form_dial (3, x, y, w, h, x, y, w, h);
}


main()
{
  gem_init();

  /* Resource-File (DIALOG1.RSC) laden */

  if (rsrc_load ("DIALOG1.RSC") == 0)
    form_alert (1, "[3][Kein RSC-File!][Ende]");
  else
  {
    /* Startadresse des (0 =) Baums DIALOG festellen */
    rsrc_gaddr (0, DIALOG, &baum_adr);

    /* Dialogbox anzeigen lassen: */
    show_dialog (baum_adr);

    /* Dialog abarbeiten lassen, d.h. auf 'Knopfdruck' warten */
    knopf = form_do (baum_adr, 0);   /* 0 weil kein Edit-Feld */

    /* Status 'SELECTED' des gedrückten Buttons aufheben */
    deselect (baum_adr, knopf);

    /* Dialogbox wieder verschwinden lassen */
    hide_dialog (baum_adr);

    /* Meldung an den Anwender, welcher Knopf gedrückt wurde: */

    switch (knopf)
    {
      case KNOPF1: strcpy (ziffer, "1");
                   break;

      case KNOPF2: strcpy (ziffer, "2");
                   break;

      case KNOPF3: strcpy (ziffer, "3");
                   break;
    }

    strcpy (str, "[1][Sie haben den Knopf Nr. ");
    strcat (str, ziffer);
```

```
    strcat (str, " gedrückt![Richtig!]");

    form_alert (1, str);

    /* Resource-File aus dem Speicher werfen: */
    rsrc_free();
  }

  gem_exit();
}
```

Assembler

```
          ;
          ; Erste Dialogbox laden, anzeigen und verarbeiten
          ; Assembler          MP  17-11-88       DIALOG1.Q
          ;

gemdos    = 1

DIALOG    = 0       ;Konstanten aus DIALOG1.H2
KNOPF3    = 4
KNOPF1    = 5
KNOPF2    = 6

          INCLUDE 'GEM_INEX.Q'

          TEXT

main:     jsr       gem_init

          ; Pfeil als Mauszeiger:

          move.w    #78,control      ;graf_mouse
          move.w    #1,control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          clr.w     int_in           ;0 für Pfeil
          jsr       aes

          ; rsrc_load:

          move.w    #110,control
          clr.w     control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          move.l    #rscname,addr_in
```

```
        jsr     aes

        tst.w   int_out         ;Fehler?
        beq     rscerr

        ; rsrc_gaddr ermittelt Startadresse des Dialogs:

        move.w  #112,control
        move.w  #2,control+2
        move.w  #1,control+4
        clr.w   control+6
        move.w  #1,control+8
        clr.w   int_in          ;0 für 'Baum gesucht'
        move.w  #DIALOG,int_in+2 ;Index des Baumes
        jsr     aes
        move.l  addr_out,baum_adr   ;Ergebnis: die Startadresse

        jsr     show_dialog     ;Anzeigen des Baumes

        ; form_do läßt Dialog abarbeiten:

        move.w  #50,control
        move.w  #1,control+2
        move.w  #1,control+4
        move.w  #1,control+6
        clr.w   control+8
        clr.w   int_in          ;kein Edit-Feld
        move.l  baum_adr,addr_in
        jsr     aes
        move.w  int_out,knopf   ;gedrückter Knopf

        jsr     deselect        ;Selected-Status löschen

        jsr     hide_dialog     ;Dialogbox vom Bildschirm entfernen

        ; Meldung je nach gedrücktem Knopf:

        move.w  knopf,d1
        moveq.l #1,d0
        cmpi.w  #KNOPF2,d1
        bne.s   not2
        moveq.l #2,d0
not2:   cmpi.w  #KNOPF3,d1
        bne.s   not3
        moveq.l #3,d0

not3:   addi.b  #'0',d0         ;Aus Zahl eine ASCII-Ziffer machen
        move.b  d0,ziffer

        lea     form_str,a0     ;Alert-Box anzeigen
```

```
        jsr     form_alert

        ; rsrc_free:

        move.w  #111,control
        clr.w   control+2
        move.w  #1,control+4
        clr.w   control+6
        clr.w   control+8
        jsr     aes

quit:   jsr     gem_exit

        clr.w   -(sp)
        trap    #gemdos


rscerr: lea     errtxt,a0
        jsr     form_alert      ;Warnhinweis ausgeben

        bra     quit


show_dialog:
        ; Dieses Unterprogramm malt einen Objektbaum auf den
        ; Bildschirm. Dazu muß sich dessen Startadresse unter
        ; 'baum_adr' befinden.

        ; form_center

        move.w  #54,control
        clr.w   control+2
        move.w  #5,control+4
        move.w  #1,control+6
        clr.w   control+8
        move.l  baum_adr,addr_in
        jsr     aes
        move.w  int_out+2,d4    ;Koordinaten sichern
        move.w  int_out+4,d5
        move.w  int_out+6,d6
        move.w  int_out+8,d7

        ; form_dial rettet Fensterränder etc. (0)

        move.w  #51,control
        move.w  #9,control+2
        move.w  #1,control+4
        clr.w   control+6
        clr.w   control+8
        clr.w   int_in          ;Unterfunktion 0
```

```
        ; keine Werte für das kleine Rechteck fo_dilittlx/y/w/h
        move.w    d4,int_in+10      ;großes Rechteck do_dibigx/y/w/h
        move.w    d5,int_in+12
        move.w    d6,int_in+14
        move.w    d7,int_in+16
        jsr       aes

        ; form_dial zeichnet 'Zoom'-Rechteck (1):

        move.w    #51,control
        move.w    #9,control+2
        move.w    #1,control+4
        clr.w     control+6
        clr.w     control+8
        move.w    #1,int_in         ;Unterfunktion 1
        move.w    d4,int_in+2       ;Größe des kleinen Rechtecks
        move.w    d5,int_in+4
        move.w    #1,int_in+6
        move.w    #1,int_in+8
        move.w    d4,int_in+10      ;großes Rechteck do_dibigx/y/w/h
        move.w    d5,int_in+12
        move.w    d6,int_in+14
        move.w    d7,int_in+16
        jsr       aes

        ; Dialog zeichnen mit objc_draw:

        move.w    #42,control
        move.w    #6,control+2
        move.w    #1,control+4
        move.w    #1,control+6
        clr.w     control+8
        clr.w     int_in            ;0=Wurzelobjekt zuerst zeichnen
        move.w    #12,int_in+2      ;max. 12 Ebenen (willkürlich)
        move.w    d4,int_in+4
        move.w    d5,int_in+6
        move.w    d6,int_in+8
        move.w    d7,int_in+10
        move.l    baum_adr,addr_in
        jmp       aes


hide_dialog:
        ; Entfernt das Formular vom Bildschirm.
        ; Adresse des Objektbaums wieder in baum_adr

        ; form_center

        move.w    #54,control
        clr.w     control+2
```

```
        move.w    #5,control+4
        move.w    #1,control+6
        clr.w     control+8
        move.l    baum_adr,addr_in
        jsr       aes
        move.w    int_out+2,d4      ;Koordinaten sichern
        move.w    int_out+4,d5
        move.w    int_out+6,d6
        move.w    int_out+8,d7

        ; form_dial zeichnet schrumpfendes-Rechteck (2):

        move.w    #51,control
        move.w    #9,control+2
        move.w    #1,control+4
        clr.w     control+6
        clr.w     control+8
        move.w    #2,int_in         ;Unterfunktion 2
        move.w    d4,int_in+2       ;Größe des kleinen Rechtecks
        move.w    d5,int_in+4
        move.w    #1,int_in+6
        move.w    #1,int_in+8
        move.w    d4,int_in+10      ;großes Rechteck do_dibigx/y/w/h
        move.w    d5,int_in+12
        move.w    d6,int_in+14
        move.w    d7,int_in+16
        jsr       aes

        ; form_dial sendet Redraw-Meldungen an Fenster (3)

        move.w    #51,control
        move.w    #9,control+2
        move.w    #1,control+4
        clr.w     control+6
        clr.w     control+8
        move.w    #3,int_in         ;Unterfunktion 3
        ; keine Werte für das kleine Rechteck fo_dilittlx/y/w/h
        move.w    d4,int_in+10      ;großes Rechteck do_dibigx/y/w/h
        move.w    d5,int_in+12
        move.w    d6,int_in+14
        move.w    d7,int_in+16
        jmp       aes


select:
        ; Schaltet den Button 'knopf' auf 'selected'-Status
        ; Objektbaum muß in baum_adr stehen

        movea.l   baum_adr,a0
        move.w    knopf,d0          ;Index des Objekts
```

```
        mulu    #24,d0          ;* 24 (jedes Objekt hat 24 Bytes)
        ori.w   #1,10(a0,d0.w)  ;Bit 0 (selected) setzen
        rts


deselect:
        ; Schaltet den Button 'knopf' auf 'nicht-selected'-Status
        ; Objektbaum muß in baum_adr stehen

        movea.l baum_adr,a0
        move.w  knopf,d0        ;Index des Objekts
        mulu    #24,d0          ;* 24 (jedes Objekt hat 24 Bytes)
        andi.w  #-2,10(a0,d0.w) ;Bit 0 (selected) löschen
        rts


form_alert:
        ; Zeigt Alert-Box. Beschreibender String steht ab a0.
        ; Knopf Nr. 1 ist Default-Button.

        move.w  #52,control
        move.w  #1,control+2
        move.w  #1,control+4
        move.w  #1,control+6
        clr.w   control+8
        move.w  #1,int_in       ;Default-Button
        move.l  a0,addr_in
        jmp     aes



        DATA

rscname: DC.b 'DIALOG1.RSC',0

form_str: DC.b '[1][Sie haben den Knopf Nr. '
ziffer:  DC.b '0 gedrückt!][Richtig!]',0

errtxt:  DC.b '[3][Kein RSC-File!][Ende]',0


        BSS

baum_adr: DS.l 1
knopf:  DS.w 1

        END
```

Vielleicht haben Sie einmal versucht, auf einem Button einen Doppelklick auszuführen und wunderten sich, daß das Programm entweder "hängen"

blieb oder sogar abstürzte. Nun, GEM übermittelt uns mit der form-do-Routine nicht nur die Indexnummer des gewählten Objektes, sondern setzt im Falle eines Doppelklicks zusätzlich das höchste Bit. Wenn Sie auf einem Doppelklick nicht reagieren wollen, so müssen Sie das Bit durch

```
knopf = knopf and 32767
```

löschen. Erst danach darf die Deselektier-Routine aufgerufen werden. Dieses erste Beispielprogramm zeigt noch nicht, wie leistungsfähig die Dialogboxen unter GEM sind; es war ja auch mehr eine bessere Alert-Box, in der ein Button angeklickt werden mußte.

Oft gibt es aber ganz andere Arten von Abfragen: Zahlen und Strings müssen eingegeben werden. Kein Problem unter GEM! Dazu bauen Sie im RCS ein Edit-Feld in Ihren Dialog mit ein. Wenn Sie später im Programm die Routine form_do aufrufen, dann wird nicht nur auf das Betätigen eines Exit-Buttons gewartet; der Rechner nimmt auch vollautomatisch Eingaben im Edit-Feld entgegen, und zwar so, wie Sie es von Dialogboxen gewöhnt sind, d.h. die Taste Esc löscht das aktuelle Eingabefeld, und auch die Cursortasten sowie Delete und Backspace funktionieren! Und das alles mit einem einzigen form_do-Aufruf- Sie werden sehen!

Zwei Probleme sind nun zu diskutieren: Erstens ist es nicht ganz leicht, die Edit-Felder im RCS anzulegen, und zweitens müssen wir innerhalb des Programms Zugriff auf den Inhalt dieses Feldes haben; denn was nützt uns die Eingabe des Benutzers, wenn unser Programm sie nicht lesen kann.

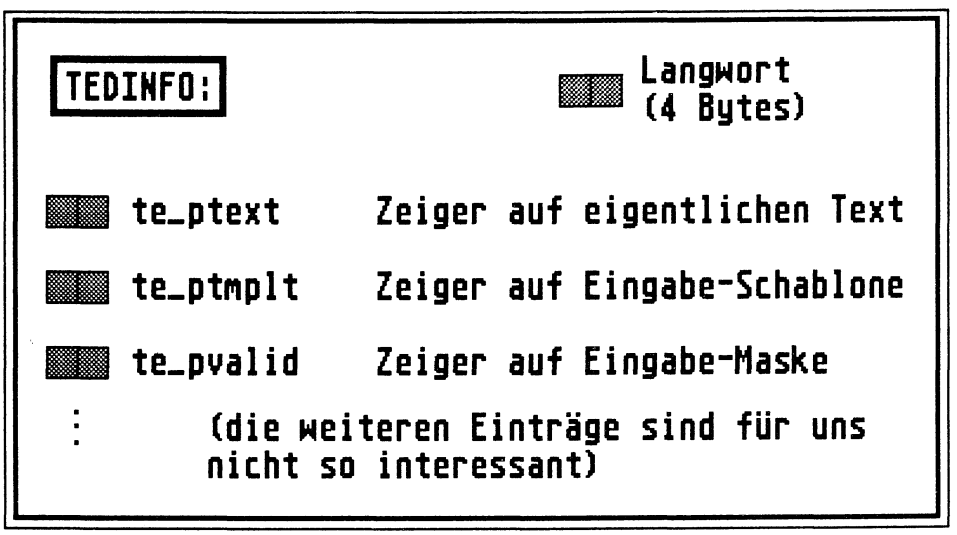
Betrachten wir das erste Problem: Wenn Sie im RCS eine Dialogbox basteln, dann haben Sie (wahrscheinlich) in dem Fenster mit den verschiedenen Bauteilen auch ein Teil mit dem Namen EDIT. Das bewegen Sie wie gewohnt an die gewünschte Stelle in Ihrer Dialogbox. Dann doppelklicken Sie auf diesem neuen Feld. Es sollte eine riesige Box erscheinen, in der, meist ganz unten, drei Zeilen zur Text-Eingabe stehen. Diese Zeilen haben furchtbare Namen:

```
te_ptext, te_ptmplt und te_pvalid
```

Ein wenig Objektkunde kann ich Ihnen auch hier nicht ersparen: In einem Objekt zeigt der ob_spec-Zeiger bekanntlich auf unterschiedliche Dinge (das hängt vom Objekt-Typ ab). In einem Edit-Objekt zeigt er auf eine neue Datenstruktur, die weitere Informationen über das Objekt enthält; man hätte diese nicht allein in den zwölf Worten des Objekts unter-

bringen können. Diese Struktur heißt TEDINFO, das heißt Text/EDit-INFOrmation. Diesem Namen können Sie auch schon entnehmen, daß eine TEDINFO-Struktur nicht nur bei EDIT-, sondern auch bei TEXT-Objekten benutzt wird (nicht zu verwechseln mit STRING!).

Die TEDINFO-Struktur ist nicht allzu interessant; nur die ersten drei Langworte sind für uns wichtig:



Die Namen sollten Ihnen bekannt vorkommen. Also: te steht für TEDINFO, und p bedeutet Pointer. Es sind also drei Zeiger, und zwar Zeiger auf Strings. Und genau diese Strings sind die geheimnisvollen Einträge, die Sie im RCS für ein EDIT-Objekt ausfüllen müssen - was Sie hier hineinschreiben, steht nach rsrc_load im Speicher, und te_ptext, te_ptmplt und te_pvalid zeigen auf diese Strings. Was müssen Sie da also im RCS eintragen?

Beginnen wir mit te_ptmplt. Die letzten fünf Buchstaben heißen template. Das Wort ist schwer zu übersetzen; es bezeichnet gewöhnlich Führungssysteme für Maschinen. In unserem Fall wird dieser String die Eingabe quasi führen, aber darunter können Sie sich bestimmt auch nicht viel mehr vorstellen. Vielleicht sollten wir deshalb im Deutschen die Bezeichnung Maske bzw. Eingabemaske verwenden. Das ist ein String, der später in der Dialogbox ausgegeben wird. Allerdings hat ein Zeichen innerhalb dieses Strings eine Besonderheit: der Unterstrich (_). Wo immer dieses Zeichen in der Maske auftaucht, kann der Benutzer später, also wenn der Dialog innerhalb einer Applikation wirklich abgearbeitet wird,

ein Zeichen eingeben. Wenn der Benutzer mehr als nur ein Zeichen eingeben soll, so müssen Sie entsprechend viele Unterstriche in die Maske schreiben. Aber die Maske hat noch eine weitere Aufgabe, die wieder stärker mit der eigentlichen Bedeutung des Wortes template zusammenhängt: Wenn in einer Maske nämlich mehrere Gruppen von diesen Strichen stehen, d.h. wenn Striche durch normale Zeichen getrennt werden, dann kann der Benutzer innerhalb dieses einen Edit-Objektes zum nächsten Unterfeld springen, also den Cursor an den Anfang der nächsten Strichgruppe setzen, indem er das Zeichen auf der Tastatur tippt, das die beiden Strichgruppen trennt.

Dazu zwei sinnvolle Beispiele für templates:

```
1. Datum: __/__/__
Wenn der Benutzer eingibt: 2/5/80, so erscheint auf dem Bildschirm:
Datum: 2 /5 /80


2. Dateiname: ________.___
Gibt der Benutzer hier ein: BTEXT.PRG, so erscheint:
Dateiname: BTEXT    .PRG
```

Mit dem template-String ist also eine formatierte Eingabe möglich.

Betrachten wir den nächsten Zeiger: te_pvalid. Valid heißt gültig, und das trifft den Punkt einmal ganz genau: Mit dem valid-String müssen Sie für jeden einzelnen Unterstrich in einem template bestimmen, welche Zeichen der Benutzer hier eintragen darf, d.h. welche Zeichen gültig oder erlaubt sind. Dazu geben Sie für jeden Unterstrich der Eingabemaske ein Zeichen an, das alle hier erlaubten Eingabezeichen repräsentiert. Diese Zeichen sind durch einen Code wie folgt festgelegt:

valid-Code	erlaubte Zeichen
9	alle Ziffern (0-9)
A	alle Großbuchstaben und das Leerzeichen
a	alle Buchstaben und Leerzeichen
N	Großbuchstaben, Ziffern und Leerzeichen
n	alle Buchstaben, Ziffern und Leerzeichen
F	File-Namen einschließlich ? * :
P (groß)	Pfadname einschließlich ? * :
p (klein)	Pfadname einschließlich :
X	alle Zeichen

Der Unterschied zwischen File- und Pfadnamen liegt übrigens nur darin, daß ein Pfadname zusätzlich den Backslash (\) enthalten darf.

Beachten Sie bitte, daß Ihnen die meisten Resource Construction Sets zur Eingabe dieses valid-Strings eine Hilfe anbieten: Da Sie ja nur für die Unterstriche der Maske, nicht aber für den erklärenden Text (z.B. Datum: oder Dateiname:) valid-Zeichen eingeben müssen, wäre es praktisch, wenn Sie die valid-Codes immer direkt unter den Strichen der Maske eingeben könnten. Das geht, indem Sie das Proportionalitätszeichen, die Tilde, als Füllzeichen verwenden; es wird vom RCS als nicht vorhanden angesehen (Beispiel s.u.).

Schließlich müssen Sie noch den dritten String vorgeben: te_ptext. Das ist, wie der Name vermuten läßt, der Text, der beim Anzeigen der Dialogbox in die Maske eingebaut wird, und zwar nur an den Stellen, wo innerhalb der Maske ein Strich steht. Hier müssen Sie im RCS auf jeden Fall so viele Zeichen eingeben, wie auch Striche in der Maske vorhanden sind; denn bei einer Benutzer-Eingabe im laufenden Programm werden die Zeichen direkt in diesem String, also in den Resource-Daten, gespeichert (durch Nullbyte abgeschlossen). Wenn also hier nicht genug Platz für die Eingaben ist, dann überschreibt der Anwender womöglich wichtige andere Daten des Objektbaums. Übrigens können Sie bei der Eingabe des text-Strings im RCS auch den gleichen Trick benutzen, wie ich ihn schon für valid empfohlen habe: das Zeichen ~, das nicht mit im RSC-File abgespeichert wird. Und noch etwas: Mit welchem Zeichen Sie den text-String füllen, ist ganz egal; es kommt nur auf die Länge an. Wenn Sie möchten, daß hier bei der Anzeige der Dialogbox in Ihrem Programm ein bestimmter String steht (z.B. ein Leerstring), dann sollten Sie diesen String erst zur Laufzeit, d.h. in Ihrem eigenen Programm, dorthin schreiben.

Auch dazu ein Beispiel:

```
te_ptmplt:   Ihren Namen, bitte: _____
te_pvalid:   ~~~~~~~~~~~~~~~~~~~Aaaaaaaaaaaaaaaaaaa
te_ptext:    ~~~~~~~~~~~~~~~~~~~auhohfshdjkldhuoeblf
```

Damit ist sichergestellt, daß der Benutzer seinen Namen mit einem Großbuchstaben beginnt; außerdem ist genügend Platz für die Eingabe des Benutzers reserviert (durch den Namen, den Sie besser nicht aussprechen sollten).

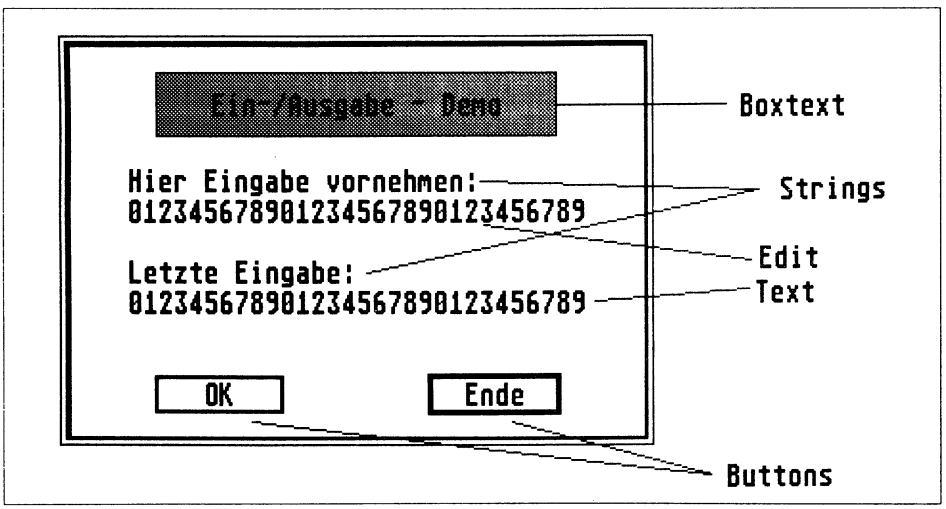
Punkt 2: Wie geht das Ganze denn jetzt in meinem Programm? Betrachten wir zunächst einmal den einfachen Teil, nämlich die wirkliche Eingabe in dieses Edit-Feld durch den Benutzer. Die erledigt das AES so nebenher, wenn Sie mit form_do auf das Betätigen eines Exit-Buttons warten. Wenn Sie mehrere Edit-Objekte in einer Box haben, dann kann

der Anwender auch automatisch mit Cursor Up/Down bzw. Tab zwischen den einzelnen Feldern hin und her springen; auch alle anderen Cursor-Aktivitäten werden vom AES überwacht. Sie müssen allerdings beim Aufruf von form_do einen zusätzlichen Parameter angeben, den wir bisher einfach auf Null gesetzt haben, nämlich den Index des Edit-Objektes, an dem sich der Textcursor zu Anfang befinden soll. Wenn Sie nur ein Edit-Objekt in Ihrem Objektbaum haben, dann geben Sie einfach den Index dieses Objekts an.

Bei der letzten Frage wird es noch einmal problematisch: Das Programm muß Zugang zu dem String te_ptext haben, um Strings vor der Eingabe vorgeben (oder das Eingabefeld durch einen Leerstring zu löschen) und nach der Eingabe den Text auslesen zu können. Hier ist wieder ein wenig Zeigerakrobatik nötig: Bei einem Edit-Objekt zeigt der ob_spec-Zeiger auf eine TEDINFO-Struktur. In dieser TEDINFO-Struktur ist wiederum das erste Langwort ein Zeiger auf te_ptext - all das können Sie den Abbildungen der Objekt- und TEDINFO-Struktur entnehmen! Die eigentliche Berechnung der te_ptext-Adresse ist allerdings (wie schon bei ob_state, Stichwort: selected) in den verschiedenen Programmiersprachen unterschiedlich zu lösen: In GFA-BASIC haben Sie mit OB_SPEC(adresse,index) Zugriff auf diesen Zeiger; in Omikron-BASIC müssen einfache Peeks und Pokes herhalten. In C arbeiten wir wieder mit den C-Objekt-Definitionen der Datei OBDEFS.H, die neben dem Ihnen schon bekannten OBJECT-Struct auch eine TEDINFO-Struktur enthält. In Assembler gibt es diesen Luxus natürlich nicht, doch auch hier kann man das Problem kurz und elegant lösen. Am besten schauen Sie sich jeweils im Listing an, wie die Unterprogramme in "Ihrer" Sprache aussehen. Ach ja, die Unterprogramme heißen übrigens read_text und write_text, mit denen Sie Strings lesen und schreiben können. Diese Routinen können Sie aber nicht nur für Edits, sondern auch für Text-Objekte (nicht Strings!) benutzen, da auch diese, vielleicht erinnern Sie sich, mit TEDINFO-Strukturen arbeiten. Das bietet sich immer dann an, wenn Sie innerhalb einer Dialogbox auch reine Ausgaben machen möchten, die der Benutzer nicht editieren können soll.

Im Programm erhalten die Unterprogramme read_text und write_text jeweils drei Parameter: die Adresse des Objektbaums, den Index des Edit-/Text-Objekts und eine Stringvariable/einen String für den zu lesenden/schreibenden Text. In BASIC fehlt einem String üblicherweise das abschließende Nullbyte; deshalb hängen wir es im Unterprogramm einfach an (+CHR$(0)).

Und nun zu meinem zweiten Dialog-Beispielprogramm: DIALOG2. Dem Programm liegt folgende Dialogbox zu Grunde:



Folgende Namen habe ich den Objekten gegeben: Das Edit-Objekt heißt EINGABE, das Text-Objekt AUSGABE. Die Buttons heißen OK und ENDE. Der Objektbaum als Ganzes trägt wieder den Namen DIALOG.

Das Programm soll eine Eingabe beliebigen Typs entgegennehmen, d.h. alle Zeichen sind erlaubt (valid-String besteht aus XXXX...). Die Eingabe darf maximal 30 Zeichen lang sein. Auch das Text-Objekt muß schon im RCS mit einem Text gleicher Länge gefüllt werden (ich habe Ziffern genommen, um die Länge leichter zählen zu können). Wenn der Benutzer die Eingabe mit OK bestätigt, dann wird das Programm die Box erneut auf den Bildschirm bringen und die letzte Eingabe im Text-Objekt anzeigen. Mit Klick auf Ende oder Druck auf die Return-Taste ist das Programm beendet; ENDE ist also der Default-Button.

GFA-BASIC

```
'
' Dialogbox laden, anzeigen und Edits verarbeiten
' GFA-BASIC         MP  16-12-88        DIALOG2.GFA
'
DEFINT "a-z"   ! alle Variablen 4-Byte-Integer
'
dialog=0       ! Konstanten aus DIALOG.H2
eingabe=3
ausgabe=5
ok=6
ende=7
'
```

```
VOID APPL_INIT()
'
IF RSRC_LOAD("DIALOG2.RSC")=0
  VOID FORM_ALERT(1,"[3][Kein Rsc-File!][Ende]")
ELSE
  '
  ' Startadresse des Baumes erfragen:
  '
  VOID RSRC_GADDR(0,dialog,baum_adr)
  '
  ' Ausgabefeld initialisieren:
  '
  GOSUB write_text(baum_adr,ausgabe,"*** keine ***")
  '
  REPEAT
    '
    ' Eingabefeld löschen:
    '
    GOSUB write_text(baum_adr,eingabe,"")
    '
    ' Dialogbox anzeigen lassen:
    '
    GOSUB show_dialog(baum_adr)
    '
    ' Dialogbox abarbeiten lassen; 'eingabe' erstes Edit-Feld
    '
    knopf=FORM_DO(baum_adr,eingabe)
    '
    ' Status 'selected' des gewählten Knopfes aufheben:
    '
    GOSUB deselect(baum_adr,knopf)
    '
    ' Dialogbox wieder verschwinden lassen:
    '
    GOSUB hide_dialog(baum_adr)
    '
    ' Eingabe auslesen und in Ausgabefeld schreiben:
    '
    read_text(baum_adr,eingabe,a$)
    write_text(baum_adr,ausgabe,a$)
    '
  UNTIL knopf=ende
  '
  ' Resource-File aus dem Speicher entfernen:
  '
  VOID RSRC_FREE()
  '
ENDIF
'
VOID APPL_EXIT()
```

```
'
END
'
'
PROCEDURE select(baum,index)
  ' Bit 0 setzen:
  OB_STATE(baum,index)=OB_STATE(baum,index) OR 1
RETURN
'
'
PROCEDURE deselect(baum,index)
  ' Bit 0 löschen:
  OB_STATE(baum,index)=OB_STATE(baum,index) AND -2
RETURN
'
'
PROCEDURE show_dialog(baum)
  LOCAL x,y,w,h
  '
  ' Formular auf dem Bildschirm zentrieren (wird dabei noch
  ' nicht gezeichnet)
  '
  VOID FORM_CENTER(baum,x,y,w,h)
  '
  ' Fensterränder etc. retten lassen:
  '
  VOID FORM_DIAL(0,x,y,w,h,x,y,w,h)
  '
  ' Zeichnen eines 'Zoom'-Rechtecks:
  '
  VOID FORM_DIAL(1,25,25,25,25,x,y,w,h)
  '
  ' Zeichnen des Formulars:
  ' Start bei Objekt Nr. 0 (Wurzelobjekt, äußerer Kasten/Rahmen)
  ' Tiefe: max. 12 Ebenen (willkürlich)
  '
  VOID OBJC_DRAW(baum,0,12,x,y,w,h)
  '
RETURN
'
'
PROCEDURE hide_dialog(baum)
  LOCAL x,y,w,h
  '
  ' Nochmal die Koordinaten erfragen:
  '
  VOID FORM_CENTER(baum,x,y,w,h)
  '
  ' Zeichnen eines kleiner werdenden Rechtecks:
  '
```

```
  VOID FORM_DIAL(2,25,25,25,25,x,y,w,h)
  '
  ' Fensterränder wiederherstellen und Redraw-Meldung
  ' an alle zerstörten Fenster veranlassen:
  '
  VOID FORM_DIAL(3,x,y,w,h,x,y,w,h)
  '
RETURN
'
'
PROCEDURE write_text(baum,index,text$)
  LOCAL adr,i,a$
  '
  ' Startadresse des reinen Textes (te_ptext) ermitteln:
  '
  adr=LPEEK(OB_SPEC(baum,index))
  '
  ' Nullbyte anhängen:
  '
  a$=text$+CHR$(0)
  '
  FOR i=1 TO LEN(a$)
    POKE adr+i-1,ASC(MID$(a$,i,1))
  NEXT i
RETURN
'
'
PROCEDURE read_text(baum,index,VAR text$)
  LOCAL adr,i
  '
  ' te_ptext ermitteln:
  '
  adr=LPEEK(OB_SPEC(baum,index))
  '
  text$=""
  i=0
  '
  WHILE PEEK(adr+i)<>0    ! bis zum Nullbyte lesen
    text$=text$+CHR$(PEEK(adr+i))
    INC i
  WEND
RETURN
'
```

Omikron-BASIC

```
'
' Dialogbox laden, anzeigen und Edits verarbeiten
' Omikron-BASIC      MP  16-12-88     DIALOG2.BAS
'
```

```
'
Dialog%L=0'          Konstanten aus DIALOG.H2
Eingabe%L=3
Ausgabe%L=5
Ok%L=6
Ende%L=7
'
Appl_Init
'
Rsrc_Load("DIALOG2.RSC",Ret%L)
 IF Ret%L=0 THEN
 FORM_ALERT (1,"[3][Kein Rsc-File!][Ende]",Dummy%L)
ELSE
 '
 ' Startadresse des Baumes erfragen:
 '
 Rsrc_Gaddr(0,Dialog%L,Baum_Adr%L)
 '
 ' Ausgabefeld initialisieren:
 '
 Write_Text(Baum_Adr%L,Ausgabe%L,"*** keine ***")
 '
 REPEAT
   '
   ' Eingabefeld löschen:
   '
   Write_Text(Baum_Adr%L,Eingabe%L,"")
   '
   ' Dialogbox anzeigen lassen:
   '
   Show_Dialog(Baum_Adr%L)
   '
   ' Dialogbox abarbeiten lassen; 'eingabe' erstes Edit-Feld
   '
   Form_Do(Eingabe%L,Baum_Adr%L,Knopf%L)
   '
   ' Status 'selected' des gewählten Knopfes aufheben:
   '
   Deselect(Baum_Adr%L,Knopf%L)
   '
   ' Dialogbox wieder verschwinden lassen:
   '
   Hide_Dialog(Baum_Adr%L)
   '
   ' Eingabe auslesen und in Ausgabefeld schreiben:
   '
   Read_Text(Baum_Adr%L,Eingabe%L,A$)
   Write_Text(Baum_Adr%L,Ausgabe%L,A$)
   '
 UNTIL Knopf%L=Ende%L
```

```
 '
 ' Resource-File aus dem Speicher entfernen:
 '
 Rsrc_Free
 '
ENDIF
'
Appl_Exit
'
END
'
'
DEF PROC Select(Baum%L,Index%L)
 ' Bit 0 setzen:
 WPOKE Baum%L+24*Index%L+10, WPEEK(Baum%L+24*Index%L+10) OR 1
RETURN
'
'
DEF PROC Deselect(Baum%L,Index%L)
 ' Bit 0 löschen:
 WPOKE Baum%L+24*Index%L+10, WPEEK(Baum%L+24*Index%L+10) AND -2RETURN
'
'
DEF PROC Show_Dialog(Baum%L)
 LOCAL X%L,Y%L,W%L,H%L
 '
 ' Formular auf dem Bildschirm zentrieren (wird dabei noch
 ' nicht gezeichnet)
 '
 Form_Center(Baum%L,X%L,Y%L,W%L,H%L)
 '
 ' Fensterränder etc. retten lassen:
 '
 Form_Dial(0,X%L,Y%L,W%L,H%L)
 '
 ' Zeichnen eines 'Zoom'-Rechtecks:
 '
 Form_Dial(1,X%L,Y%L,W%L,H%L)
 '
 ' Zeichnen des Formulars:
 ' Start bei Objekt Nr. 0 (Wurzelobjekt, äußerer Kasten/Rahmen)
 ' Tiefe: max. 12 Ebenen (willkürlich)
 '
 Objc_Draw(0,12,X%L,Y%L,W%L,H%L,Baum%L)
 '
RETURN
'
'
DEF PROC Hide_Dialog(Baum%L)
 LOCAL X%L,Y%L,W%L,H%L
```

```
'
' Nochmal die Koordinaten erfragen:
'
Form_Center(Baum%L,X%L,Y%L,W%L,H%L)
'
' Zeichnen eines kleiner werdenden Rechtecks:
'
Form_Dial(2,X%L,Y%L,W%L,H%L)
'
' Fensterränder wiederherstellen und Redraw-Meldung
' an alle zerstörten Fenster veranlassen:
'
Form_Dial(3,X%L,Y%L,W%L,H%L)
'
RETURN
'
'
DEF PROC Write_Text(Baum%L,Index%L,Text$)
 LOCAL Adr%L,I%L,A$
 '
 ' Startadresse des reinen Textes (te_ptext) ermitteln:
 '
 Adr%L= LPEEK( LPEEK(Baum%L+24*Index%L+12))
 '
 ' Nullbyte anhängen:
 '
 A$=Text$+ CHR$(0)
 '
 FOR I%L=1 TO LEN(A$)
   POKE Adr%L+I%L-1, ASC( MID$(A$,I%L,1))
 NEXT I%L
RETURN
'
'
DEF PROC Read_Text(Baum%L,Index%L,R Text$)
 LOCAL Adr%L,I%L
 '
 ' te_ptext ermitteln:
 '
 Adr%L= LPEEK( LPEEK(Baum%L+24*Index%L+12))
 '
 Text$=""
 I%L=0
 '
 WHILE PEEK(Adr%L+I%L)<>0'          bis zum Nullbyte lesen
   Text$=Text$+ CHR$( PEEK(Adr%L+I%L))
   I%L=I%L+1
 WEND
RETURN
'
```

C

```
/*********************************************************/
/*   Dialogbox laden, anzeigen und Edits verarbeiten   */
/*   Megamax Laser C     MP  14-11-88      DIALOG2.C   */
/*********************************************************/

#include <obdefs.h>    /* GEM-Objekt-Definitionen */

#include "gem_inex.c"
#include "dialog2.h"   /* Header-File der Resource-Datei */

OBJECT *baum_adr;      /* pointer auf das erste Onjekt eines Baumes */
int    knopf;
char   eingabe[80];


select (baum, index)          /* schaltet Button 'object' ein */
OBJECT baum[];
int    index;
{
  baum[index].ob_state |= 1;     /* Bit 0 in ob_state setzen */
}


deselect (baum, index)       /* schaltet Button 'object' aus */OBJECT baum[];
int    index;
{
  baum[index].ob_state &= -2;    /* Bit 0 in ob_state löschen */
}


write_text (baum, index, string)        /* Ändert TEXT-Objekt */
OBJECT baum[];
int    index;
char   string[];
{
TEDINFO *ted;

  ted = (TEDINFO *) baum[index].ob_spec;
  strcpy (ted->te_ptext, string);

  /* ted->te_ptext   entspricht:   (*ted).te_ptext */
}


read_text (baum, index, string)         /* Liest TEXT-Objekt */
OBJECT baum[];
int    index;
char   string[];
```

```
TEDINFO *ted;

  ted = (TEDINFO *) baum[index].ob_spec;
  strcpy (string, ted->te_ptext);
}


show_dialog (baum)              /* Darstellen einer Dialogbox */
OBJECT *baum;
{
int  x, y, w, h;

  /* Formular auf dem Bildschirm zentrieren. Dabei werden nur */
  /* Koordinaten intern an die Bildschirmauflösung angepaßt,  */
  /* es wird also noch nichts gezeichnet. Außerdem erhalten   */
  /* wir die zukünftigen Koordinaten der Dialogbox.           */

  form_center (baum, &x, &y, &w, &h);

  /* Fensterränder etc. retten lassen: */
  form_dial (0, x, y, w, h, x, y, w, h);

  /* Zeichnen eines 'Zoom'-Rechtecks */
  form_dial (1, 25, 25, 25, 25, x, y, w, h);

  /* Zeichnen des Objektbaumes selbst */
  /* Start bei Objekt Nr. 0 (Wurzel, äußerer Kasten) */
  /* Tiefe: max. 12 Ebenen (willkürlich) */
  objc_draw (baum, 0, 12, x, y, w, h);
}


hide_dialog (baum)
OBJECT *baum;
{
int  x, y, w, h;

  /* Nochmal die Koordinaten erfragen: */
  form_center (baum, &x, &y, &w, &h);

  /* Zeichnen eines kleiner werdenden Rechtecks */
  form_dial (2, 25, 25, 25, 25, x, y, w, h);

  /* Fensterränder wieder herstellen lassen und Redraw- */
  /* Meldung an alle zerstörten Fenster veranlassen     */
  form_dial (3, x, y, w, h, x, y, w, h);
}


main()
```

```
{
  gem_init();

  /* Resource-File (DIALOG2.RSC) laden */

  if (rsrc_load ("DIALOG2.RSC") == 0)
    form_alert (1, "[3][Kein RSC-File!][Ende]");
  else
  {
    /* Startadresse des (0 =) Baums DIALOG festellen */
    rsrc_gaddr (0, DIALOG, &baum_adr);

    /* Ein- und Ausgabefeld initialisieren */
    write_text (baum_adr, AUSGABE, "*** keine ***");

    do
    {
      /* Eingabefeld löschen: */
      write_text (baum_adr, EINGABE, "");

      /* Dialogbox anzeigen lassen: */
      show_dialog (baum_adr);

      /* Dialog abarbeiten lassen, EINGABE erstes EDIT-Element */
      knopf = form_do (baum_adr, EINGABE);

      /* Status 'SELECTED' des gedrückten Buttons aufheben */
      deselect (baum_adr, knopf);

      /* Dialogbox wieder verschwinden lassen */
      hide_dialog (baum_adr);

      /* Eingabe auslesen und in Ausgabefeld schreiben */
      read_text (baum_adr, EINGABE, eingabe);
      write_text (baum_adr, AUSGABE, eingabe);

    }  while (knopf != ENDE);  /* Bis ENDE angeklickt wurde */
    /* Resource-File aus dem Speicher werfen: */
    rsrc_free();
  }

  gem_exit();
}
```

Assembler

```
        ;
        ; Erste Dialogbox laden, anzeigen und Edits verarbeiten
        ; Assembler             MP  17-11-88          DIALOG2.Q
        ;
```

```
gemdos    = 1

DIALOG    = 0       ;Konstanten aus DIALOG2.H2
EINGABE   = 3
AUSGABE   = 5
OK        = 6
ENDE      = 7

          INCLUDE 'GEM_INEX.Q'

          TEXT

main:     jsr       gem_init

          ; Pfeil als Mauszeiger:

          move.w    #78,control      ;graf_mouse
          move.w    #1,control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          clr.w     int_in           ;0 für Pfeil
          jsr       aes

          ; rsrc_load:

          move.w    #110,control
          clr.w     control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          move.l    #rscname,addr_in
          jsr       aes

          tst.w     int_out          ;Fehler?
          beq       rscerr

          ; rsrc_gaddr ermittelt Startadresse des Dialogs:

          move.w    #112,control
          move.w    #2,control+2
          move.w    #1,control+4
          clr.w     control+6
          move.w    #1,control+8
          clr.w     int_in           ;0 für 'Baum gesucht'
          move.w    #DIALOG,int_in+2 ;Index des Baumes
          jsr       aes
          move.l    addr_out,baum_adr   ;Ergebnis: die Startadresse
```

```
        ; Ausgabefeld initialisisren:

        lea     inittxt,a0      ;Anfangstext
        move.w  #AUSGABE,d0     ;Index des Text-Feldes
        jsr     write_text

loop:
        ; Eingabefeld löschen:

        lea     empty,a0        ;Leerstring
        move.w  #EINGABE,d0     ;Index des Edit-Feldes
        jsr     write_text

        jsr     show_dialog     ;Anzeigen des Baumes

        ; form_do läßt Dialog abarbeiten:

        move.w  #50,control
        move.w  #1,control+2
        move.w  #1,control+4
        move.w  #1,control+6
        clr.w   control+8
        move.w  #EINGABE,int_in ;Edit-Feld
        move.l  baum_adr,addr_in
        jsr     aes
        move.w  int_out,knopf   ;gedrückter Knopf

        jsr     deselect        ;Selected-Status löschen

        jsr     hide_dialog     ;Dialogbox vom Bildschirm entfernen

        ; Eingabe ins Ausgabefeld kopieren:

        lea     str,a0
        move.w  #EINGABE,d0
        jsr     read_text

        lea     str,a0
        move.w  #AUSGABE,d0
        jsr     write_text

        ; Abbruch, wenn Ende geklickt wurde:

        cmpi.w  #ENDE,knopf
        bne     loop

        ; rsrc_free:

        move.w  #111,control
        clr.w   control+2
```

```
          move.w    #1,control+4
          clr.w     control+6
          clr.w     control+8
          jsr       aes

quit:     jsr       gem_exit

          clr.w     -(sp)
          trap      #gemdos


rscerr:   lea       errtxt,a0
          jsr       form_alert      ;Warnhinweis ausgeben

          bra       quit


show_dialog:
          ; Dieses Unterprogramm malt einen Objektbaum auf den
          ; Bildschirm. Dazu muß sich dessen Startadresse unter
          ; 'baum_adr' befinden.

          ; form_center

          move.w    #54,control
          clr.w     control+2
          move.w    #5,control+4
          move.w    #1,control+6
          clr.w     control+8
          move.l    baum_adr,addr_in
          jsr       aes
          move.w    int_out+2,d4    ;Koordinaten sichern
          move.w    int_out+4,d5
          move.w    int_out+6,d6
          move.w    int_out+8,d7

          ; form_dial rettet Fensterränder etc. (0)

          move.w    #51,control
          move.w    #9,control+2
          move.w    #1,control+4
          clr.w     control+6
          clr.w     control+8
          clr.w     int_in          ;Unterfunktion 0
          ; keine Werte für das kleine Rechteck fo_dilittlx/y/w/h
          move.w    d4,int_in+10    ;großes Rechteck do_dibigx/y/w/h
          move.w    d5,int_in+12
          move.w    d6,int_in+14
          move.w    d7,int_in+16
```

```
        jsr     aes

        ; form_dial zeichnet 'Zoom'-Rechteck (1):

        move.w  #51,control
        move.w  #9,control+2
        move.w  #1,control+4
        clr.w   control+6
        clr.w   control+8
        move.w  #1,int_in       ;Unterfunktion 1
        move.w  d4,int_in+2     ;Größe des kleinen Rechtecks
        move.w  d5,int_in+4
        move.w  #1,int_in+6
        move.w  #1,int_in+8
        move.w  d4,int_in+10    ;großes Rechteck do_dibigx/y/w/h
        move.w  d5,int_in+12
        move.w  d6,int_in+14
        move.w  d7,int_in+16
        jsr     aes

        ; Dialog zeichnen mit objc_draw:

        move.w  #42,control
        move.w  #6,control+2
        move.w  #1,control+4
        move.w  #1,control+6
        clr.w   control+8
        clr.w   int_in          ;0=Wurzelobjekt zuerst zeichnen
        move.w  #12,int_in+2    ;max. 12 Ebenen (willkürlich)
        move.w  d4,int_in+4
        move.w  d5,int_in+6
        move.w  d6,int_in+8
        move.w  d7,int_in+10
        move.l  baum_adr,addr_in
        jmp     aes


hide_dialog:
        ; Entfernt das Formular vom Bildschirm.
        ; Adresse des Objektbaums wieder in baum_adr

        ; form_center

        move.w  #54,control
        clr.w   control+2
        move.w  #5,control+4
        move.w  #1,control+6
        clr.w   control+8
        move.l  baum_adr,addr_in
        jsr     aes
```

```
        move.w    int_out+2,d4      ;Koordinaten sichern
        move.w    int_out+4,d5
        move.w    int_out+6,d6
        move.w    int_out+8,d7

        ; form_dial zeichnet schrumpfendes-Rechteck (2):

        move.w    #51,control
        move.w    #9,control+2
        move.w    #1,control+4
        clr.w     control+6
        clr.w     control+8
        move.w    #2,int_in         ;Unterfunktion 2
        move.w    d4,int_in+2       ;Größe des kleinen Rechtecks
        move.w    d5,int_in+4
        move.w    #1,int_in+6
        move.w    #1,int_in+8
        move.w    d4,int_in+10      ;großes Rechteck do_dibigx/y/w/h
        move.w    d5,int_in+12
        move.w    d6,int_in+14
        move.w    d7,int_in+16
        jsr       aes

        ; form_dial sendet Redraw-Meldungen an Fenster (3)

        move.w    #51,control
        move.w    #9,control+2
        move.w    #1,control+4
        clr.w     control+6
        clr.w     control+8
        move.w    #3,int_in         ;Unterfunktion 3
        ; keine Werte für das kleine Rechteck fo_dilittlx/y/w/h
        move.w    d4,int_in+10      ;großes Rechteck do_dibigx/y/w/h
        move.w    d5,int_in+12
        move.w    d6,int_in+14
        move.w    d7,int_in+16
        jmp       aes


select:
        ; Schaltet den Button 'knopf' auf 'selected'-Status
        ; Objektbaum muß in baum_adr stehen

        movea.l   baum_adr,a0
        move.w    knopf,d0          ;Index des Objekts
        mulu      #24,d0            ;* 24 (jedes Objekt hat 24 Bytes)
        addi.w    #10,d0            ;+10 als Offset für ob_state
        ori.w     #1,0(a0,d0.w)     ;Bit 0 (selected) setzen
```

```
        rts


deselect:
        ; Schaltet den Button 'knopf' auf 'nicht-selected'-Status
        ; Objektbaum muß in baum_adr stehen

        movea.l   baum_adr,a0
        move.w    knopf,d0          ;Index des Objekts
        mulu      #24,d0            ;* 24 (jedes Objekt hat 24 Bytes)
        addi.w    #10,d0            ;+10 als Offset für ob_state
        andi.w    #-2,0(a0,d0.w)    ;Bit 0 (selected) löschen
        rts


form_alert:
        ; Zeigt Alert-Box. Beschreibender String steht ab a0.
        ; Knopf Nr. 1 ist Default-Button.

        move.w    #52,control
        move.w    #1,control+2
        move.w    #1,control+4
        move.w    #1,control+6
        clr.w     control+8
        move.w    #1,int_in         ;Default-Button
        move.l    a0,addr_in
        jmp       aes


write_text:
        ; Schreibt String in Text- oder Edit-Objekt (Index in d0)
        ; String in a0, Objektbaum in baum_adr

        movea.l   baum_adr,a1
        mulu      #24,d0
        movea.l   12(a1,d0.w),a1    ;Adresse des TEDINFO-Blocks
        movea.l   (a1),a1           ;te_ptext enthält den Textwrt_lp:
        move.b    (a0)+,(a1)+
        bne.s     wrt_lp
        rts


read_text:
        ; Gegenstück zu write_text. Zieladresse ist in a0
        ; zu übergeben.

        movea.l   baum_adr,a1
        mulu      #24,d0
        movea.l   12(a1,d0.w),a1
```

```
          movea.l   (a1),a1

rd_lp:    move.b    (a1)+,(a0)+
          bne.s     rd_lp
          rts


          DATA

rscname:  DC.b 'DIALOG2.RSC',0
errtxt:   DC.b '[3][Kein RSC-File!][Ende]',0
inittxt:  DC.b '*** keine ***',0
empty:    DC.b 0

          BSS

baum_adr: DS.l 1
knopf:    DS.w 1
str:      DS.b 40

          END
```

Mit den beiden Unterprogrammen read_text und write_text steht Ihnen eine einfache Möglichkeit zur Verfügung, Ein- und Ausgaben über Dialogboxen vorzunehmen, übrigens eine Sache, bei der sich viele Anfänger schwertun. Eine Information möchte ich Ihnen noch zur Vorsicht geben: Wenn Sie te_ptmplt wirklich als Eingabe-Führung benutzen, etwa zur Eingabe eines Filenamens (Name: _____.___), dann wird der Punkt nicht in te_ptext mit gespeichert, sondern nur als Cursorsteuer-Anweisung des Benutzers gewertet. Wenn Sie also solche Eingaben als Dateinamen an eine GEMDOS-Routine übergeben, so müssen Sie den Punkt vorher selbst einfügen, und zwar hinter dem achten Zeichen in te_ptext; denn die bei der Eingabe übersprungenen Striche werden automatisch mit Leerzeichen gefüllt.

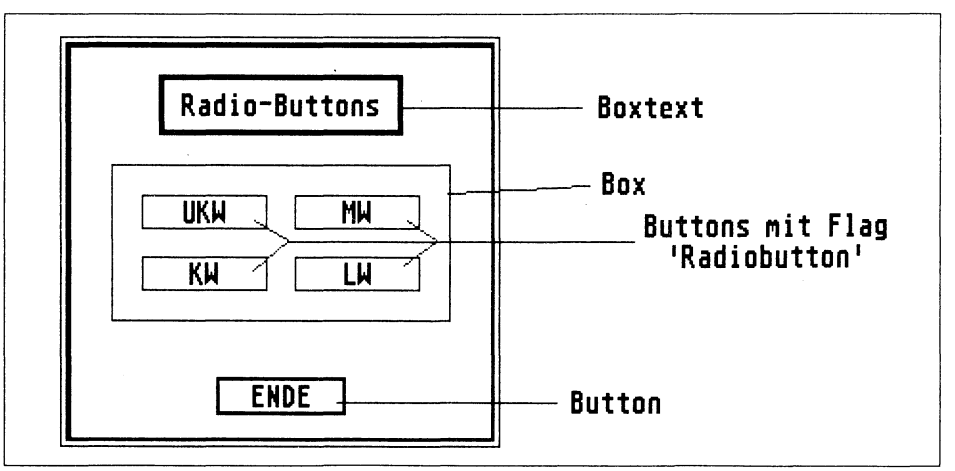
Jetzt zu einem weiteren oft benötigten Bestandteil der Dialogboxen: Radio-Buttons. Das sind Knöpfe (mindestens zwei), von denen immer nur einer gleichzeitig selektiert sein darf. Der Name Radio-Button kommt daher, daß Sie sich ja auch bei jedem Radiogerät für einen Frequenzbereich entscheiden müssen, und wenn Sie UKW gehört haben und MW drücken, dann wird der UKW-Knopf automatisch ausgerastet. In einer Dialogbox ist es ähnlich. Sobald der Benutzer einen Radio-Knopf anklickt, werden alle anderen Radio-Knöpfe automatisch deselektiert.

Es gibt natürlich auch Fälle, in denen solche Buttons in einer Dialogbox sinnvoll angewendet werden können; ich nehme nicht an, daß Sie Ihren Atari nur zum Radiohören verwenden. Aber denken Sie an einen Dialog

in einem Grafik-Programm, mit dem die Zeichenfarbe bestimmt werden soll - mehr als eine Farbe gleichzeitig geht nicht! Oder stellen Sie sich eine Adreßverwaltung mit Radio-Buttons für die Anrede vor: Herrn, Frau, Fräulein, Familie und Firma - immer nur eine Auswahl ist sinnvoll.

Wenn Sie im RCS einen Radio-Button anlegen, dann verwenden Sie einen ganz gewöhnlichen Button. Doppelklicken Sie auf diesen Knopf und wählen Sie zusätzlich zum Flag SELECTABLE (sollte schon aktiviert sein) noch das Flag RADIO BUTTON. Achten Sie außerdem darauf, daß sich alle logisch zusammengehörenden Radioknöpfe innerhalb eines übergeordneten Kastens (darf das Wurzelobjekt, also der äußere Rahmen, sein) befinden. Das ist dann wichtig, wenn Sie mehr als eine Gruppe von Radio-Buttons im selben Objektbaum verwenden, wenn also von jeder Knopfgruppe jeweils genau ein Knopf aktiv sein muß; denn sonst weiß ja der Rechner nicht mehr, welche Knöpfe er ausschalten muß, wenn Sie einen Button anklicken.

Als Beispielprogramm muß noch einmal beziehungsreich das Radio mit seinen Frequenzbereichen herhalten. Um nach der Anzeige des Dialogs überprüfen zu können, ob ein Button selektiert ist oder nicht, habe ich die Funktion selected zusätzlich zu den bisherigen Unterprogrammen eingeführt. Die Dialogbox sieht so aus:



Der Baum heißt wieder einmal DIALOG. Die vier Radio-Buttons habe ich UKW, MW, KW und LW genannt, während der Exit-Button auf den Namen ENDE hört.

GFA-BASIC

```
'
' Dialogbox laden, anzeigen, Radiobuttons verarbeiten
' GFA-BASIC          MP  16-12-88         DIALOG3.GFA
'
DEFINT "a-z"   ! alle Variablen 4-Byte-Integer
'
dialog=0       ! Kontanten aus DIALOG3.H2
mw=3
ukw=2
kw=4
lw=5
ende=6
'
VOID APPL_INIT()
'
IF RSRC_LOAD("DIALOG3.RSC")=0
  VOID FORM_ALERT(1,"[3][Kein Rsc-File!][Ende]")
ELSE
  '
  ' Startadresse des Baumes erfragen:
  '
  VOID RSRC_GADDR(0,dialog,baum_adr)
  '
  ' Einen Default-Button setzen (hier: UKW)
  '
  GOSUB select(baum_adr,ukw)
  '
  ' Dialogbox anzeigen lassen:
  '
  GOSUB show_dialog(baum_adr)
  '
  ' Dialogbox abarbeiten lassen
  '
  knopf=FORM_DO(baum_adr,0)
  '
  ' Status 'selected' des gewählten Knopfes aufheben:
  '
  GOSUB deselect(baum_adr,knopf)
  '
  ' Dialogbox wieder verschwinden lassen:
  '
  GOSUB hide_dialog(baum_adr)
  '
  ' Knöpfe auswerten:
  '
  IF FN selected(baum_adr,ukw)
    a$="Ultrakurzwelle"
  ENDIF
  '
```

```
  IF FN selected(baum_adr,mw)
    a$="Mittelwelle"
  ENDIF
  '
  IF FN selected(baum_adr,kw)
    a$="Kurzwelle"
  ENDIF
  '
  IF FN selected(baum_adr,lw)
    a$="Langwelle"
  ENDIF
  '
  VOID FORM_ALERT(1,"[1][Sie haben den Frequenzbereich|"+a$+...
...." ausgewählt!][Ja!]")
  '
  ' Resource-File aus dem Speicher entfernen:
  '
  VOID RSRC_FREE()
  '
ENDIF
'
VOID APPL_EXIT()
'
END
'
'
PROCEDURE select(baum,index)
  ' Bit 0 setzen:
  OB_STATE(baum,index)=OB_STATE(baum,index) OR 1
RETURN
'
'
PROCEDURE deselect(baum,index)
  ' Bit 0 löschen:
  OB_STATE(baum,index)=OB_STATE(baum,index) AND -2
RETURN
'
'
DEFFN selected(baum,index)=OB_STATE(baum,index) AND 1
'
'
PROCEDURE show_dialog(baum)
  LOCAL x,y,w,h
  '
  ' Formular auf dem Bildschirm zentrieren (wird dabei noch
  ' nicht gezeichnet)
  '
  VOID FORM_CENTER(baum,x,y,w,h)
  '
  ' Fensterränder etc. retten lassen:
```

```
  '
  VOID FORM_DIAL(0,x,y,w,h,x,y,w,h)
  '
  ' Zeichnen eines 'Zoom'-Rechtecks:
  '
  VOID FORM_DIAL(1,25,25,25,25,x,y,w,h)
  '
  ' Zeichnen des Formulars:
  ' Start bei Objekt Nr. 0 (Wurzelobjekt, äußerer Kasten/Rahmen)
  ' Tiefe: max. 12 Ebenen (willkürlich)
  '
  VOID OBJC_DRAW(baum,0,12,x,y,w,h)
  '
RETURN
'
'
PROCEDURE hide_dialog(baum)
  LOCAL x,y,w,h
  '
  ' Nochmal die Koordinaten erfragen:
  '
  VOID FORM_CENTER(baum,x,y,w,h)
  '
  ' Zeichnen eines kleiner werdenden Rechtecks:
  '
  VOID FORM_DIAL(2,25,25,25,25,x,y,w,h)
  '
  ' Fensterränder wiederherstellen und Redraw-Meldung
  ' an alle zerstörten Fenster veranlassen:
  '
  VOID FORM_DIAL(3,x,y,w,h,x,y,w,h)
  '
RETURN
'
'
PROCEDURE write_text(baum,index,text$)
  LOCAL adr,i,a$
  '
  ' Startadresse des reinen Textes (te_ptext) ermitteln:
  '
  adr=LPEEK(OB_SPEC(baum,index))
  '
  ' Nullbyte anhängen:
  '
  a$=text$+CHR$(0)
  '
  FOR i=1 TO LEN(a$)
    POKE adr+i-1,ASC(MID$(a$,i,1))
  NEXT i
RETURN
```

```
'
'
PROCEDURE read_text(baum,index,VAR text$)
  LOCAL adr,i
  '
  ' te_ptext ermitteln:
  '
  adr=LPEEK(OB_SPEC(baum,index))
  '
  text$=""
  i=0
  '
  WHILE PEEK(adr+i)<>0    ! bis zum Nullbyte lesen
    text$=text$+CHR$(PEEK(adr+i))
    INC i
  WEND
RETURN
'
```

Omikron-BASIC

```
'
' Dialogbox laden, anzeigen, Radiobuttons verarbeiten
' Omikron-BASIC        MP  16-12-88       DIALOG3.BAS
'
Dialog%L=0'             Kontanten aus DIALOG3.H2
Mw%L=3
Ukw%L=2
Kw%L=4
Lw%L=5
Ende%L=6
'
Appl_Init
'
Rsrc_Load("DIALOG3.RSC",Ret%L)
IF Ret%L=0 THEN
 FORM_ALERT (1,"[3][Kein Rsc-File!][Ende]")
ELSE
 '
 ' Startadresse des Baumes erfragen:
 '
 Rsrc_Gaddr(0,Dialog%L,Baum_Adr%L)
 '
 ' Einen Default-Button setzen (hier: UKW)
 '
 Select(Baum_Adr%L,Ukw%L)
 '
 ' Dialogbox anzeigen lassen:
 '
 Show_Dialog(Baum_Adr%L)
```

```
'
' Dialogbox abarbeiten lassen
'
Form_Do(0,Baum_Adr%L,Knopf%L)
'
' Status 'selected' des gewählten Knopfes aufheben:
'
Deselect(Baum_Adr%L,Knopf%L)
'
' Dialogbox wieder verschwinden lassen:
'
Hide_Dialog(Baum_Adr%L)
'
' Knöpfe auswerten:
'
IF FN Selected%L(Baum_Adr%L,Ukw%L) THEN A$="Ultrakurzwelle"
IF FN Selected%L(Baum_Adr%L,Mw%L) THEN A$="Mittelwelle"
IF FN Selected%L(Baum_Adr%L,Kw%L) THEN A$="Kurzwelle"
IF FN Selected%L(Baum_Adr%L,Lw%L) THEN A$="Langwelle"
'
FORM_ALERT (1,"[1][Sie haben den Frequenzbereich¦"+A$+...
                ..." ausgewählt!][Ja!]")
'
' Resource-File aus dem Speicher entfernen:
'
Rsrc_Free
'
ENDIF
'
Appl_Exit
'
END
'
'
DEF PROC Select(Baum%L,Index%L)
 ' Bit 0 setzen:
 WPOKE Baum%L+24*Index%L+10, WPEEK(Baum%L+24*Index%L+10) OR 1
RETURN
'
'
DEF PROC Deselect(Baum%L,Index%L)
 ' Bit 0 löschen:
 WPOKE Baum%L+24*Index%L+10, WPEEK(Baum%L+24*Index%L+10) AND -2RETURN
'
'
DEF FN Selected%L(Baum%L,Index%L)= WPEEK(Baum%L+24*Index%L+10) AND 1
'
'
DEF PROC Show_Dialog(Baum%L)
 LOCAL X%L,Y%L,W%L,H%L
```

```
'
' Formular auf dem Bildschirm zentrieren (wird dabei noch
' nicht gezeichnet)
'
Form_Center(Baum%L,X%L,Y%L,W%L,H%L)
'
' Fensterränder etc. retten lassen:
'
Form_Dial(0,X%L,Y%L,W%L,H%L)
'
' Zeichnen eines 'Zoom'-Rechtecks:
'
Form_Dial(1,X%L,Y%L,W%L,H%L)
'
' Zeichnen des Formulars:
' Start bei Objekt Nr. 0 (Wurzelobjekt, äußerer Kasten/Rahmen)
' Tiefe: max. 12 Ebenen (willkürlich)
'
Objc_Draw(0,12,X%L,Y%L,W%L,H%L,Baum%L)
'
RETURN
'
'
DEF PROC Hide_Dialog(Baum%L)
 LOCAL X%L,Y%L,W%L,H%L
 '
 ' Nochmal die Koordinaten erfragen:
 '
  Form_Center(Baum%L,X%L,Y%L,W%L,H%L)
 '
 ' Zeichnen eines kleiner werdenden Rechtecks:
 '
  Form_Dial(2,X%L,Y%L,W%L,H%L)
 '
 ' Fensterränder wiederherstellen und Redraw-Meldung
 ' an alle zerstörten Fenster veranlassen:
 '
  Form_Dial(3,X%L,Y%L,W%L,H%L)
 '
RETURN
'
'
DEF PROC Write_Text(Baum%L,Index%L,Text$)
 LOCAL Adr%L,I%L,A$
 '
 ' Startadresse des reinen Textes (te_ptext) ermitteln:
 '
 Adr%L= LPEEK( LPEEK(Baum%L+24*Index%L+12))
 '
 ' Nullbyte anhängen:
```

```
'
A$=Text$+ CHR$(0)
'
FOR I%L=1 TO LEN(A$)
  POKE Adr%L+I%L-1, ASC( MID$(A$,I%L,1))
NEXT I%L
RETURN
'
'
DEF PROC Read_Text(Baum%L,Index%L,R Text$)
 LOCAL Adr%L,I%L
 '
 ' te_ptext ermitteln:
 '
 Adr%L= LPEEK( LPEEK(Baum%L+24*Index%L+12))
 '
 Text$=""
 I%L=0
 '
 WHILE PEEK(Adr%L+I%L)<>0'         bis zum Nullbyte lesen
   Text$=Text$+ CHR$( PEEK(Adr%L+I%L))
   I%L=I%L+1
 WEND
RETURN
'
```

C

```
/*************************************************************/
/*   Dialogbox laden, anzeigen, Radiobuttons verarbeiten   */
/*   Megamax Laser C       MP  14-11-88        DIALOG3.C   */
/*************************************************************/

#include <obdefs.h>     /* GEM-Objekt-Definitionen */

#include "gem_inex.c"
#include "dialog3.h"    /* Header-File der Resource-Datei */

OBJECT *baum_adr;       /* pointer auf das erste Onjekt eines Baumes */
int    knopf;
char   str[80],
       frq[20];


select (baum, index)          /* schaltet Button 'object' ein */
OBJECT baum[];
int    index;
{
```

```
  baum[index].ob_state |= 1;      /* Bit 0 in ob_state setzen */
}


deselect (baum, index)        /* schaltet Button 'object' aus */OBJECT baum[];
int    index;
{
  baum[index].ob_state &= -2;    /* Bit 0 in ob_state löschen */
}


int selected (baum, index)   /* true, wenn objekt selektiert wurde */
OBJECT baum[];
int    index;
{
  return (baum[index].ob_state && 1);    /* Bit 0 zurückgeben */
}


write_text (baum, index, string)        /* Ändert TEXT-Objekt */
OBJECT baum[];
int    index;
char   string[];
{
TEDINFO *ted;

  ted = (TEDINFO *) baum[index].ob_spec;
  strcpy (ted->te_ptext, string);

  /* ted->te_ptext   entspricht:   (*ted).te_ptext */
}


read_text (baum, index, string)          /* Liest TEXT-Objekt */
OBJECT baum[];
int    index;
char   string[];
{
TEDINFO *ted;

  ted = (TEDINFO *) baum[index].ob_spec;
  strcpy (string, ted->te_ptext);
}


show_dialog (baum)              /* Darstellen einer Dialogbox */
OBJECT *baum;
{
int  x, y, w, h;
```

```
  /* Formular auf dem Bildschirm zentrieren. Dabei werden nur */
  /* Koordinaten intern an die Bildschirmauflösung angepaßt,  */
  /* es wird also noch nichts gezeichnet. Außerdem erhalten   */
  /* wir die zukünftigen Koordinaten der Dialogbox.           */

  form_center (baum, &x, &y, &w, &h);

  /* Fensterränder etc. retten lassen: */
  form_dial (0, x, y, w, h, x, y, w, h);

  /* Zeichnen eines 'Zoom'-Rechtecks */
  form_dial (1, 25, 25, 25, 25, x, y, w, h);

  /* Zeichnen des Objektbaumes selbst */
  /* Start bei Objekt Nr. 0 (Wurzel, äußerer Kasten) */
  /* Tiefe: max. 12 Ebenen (willkürlich) */
  objc_draw (baum, 0, 12, x, y, w, h);
}


hide_dialog (baum)
OBJECT *baum;
{
int  x, y, w, h;

  /* Nochmal die Koordinaten erfragen: */
  form_center (baum, &x, &y, &w, &h);

  /* Zeichnen eines kleiner werdenden Rechtecks */
  form_dial (2, 25, 25, 25, 25, x, y, w, h);

  /* Fensterränder wieder herstellen lassen und Redraw- */
  /* Meldung an alle zerstörten Fenster veranlassen     */
  form_dial (3, x, y, w, h, x, y, w, h);
}


main()
{
  gem_init();

  /* Resource-File (DIALOG3.RSC) laden */

  if (rsrc_load ("DIALOG3.RSC") == 0)
    form_alert (1, "[3][Kein RSC-File!][Ende]");
  else
  {
    /* Startadresse des (0 =) Baums DIALOG festellen */
    rsrc_gaddr (0, DIALOG, &baum_adr);
```

```
    /* Einen Dafault-Button setzen (hier: UKW) */
    select (baum_adr, UKW);

    /* Dialogbox anzeigen lassen: */
    show_dialog (baum_adr);

    /* Dialog abarbeiten lassen, kein EDIT-Element */
    knopf = form_do (baum_adr, 0);

    /* Status 'SELECTED' des gedrückten Buttons aufheben */
    deselect (baum_adr, knopf);

    /* Dialogbox wieder verschwinden lassen */
    hide_dialog (baum_adr);

    /* Knöpfe auswerten: */
    if (selected (baum_adr, UKW))
      strcpy (frq, "Ultrakurzwelle");

    if (selected (baum_adr, MW))
      strcpy (frq, "Mittelwelle");

    if (selected (baum_adr, KW))
      strcpy (frq, "Kurzwelle");

    if (selected (baum_adr, LW))
      strcpy (frq, "Langwelle");

    strcpy (str, "[1][Sie haben den Frequenzbereich|");
    strcat (str, frq);
    strcat (str, " ausgewählt!][Ja!]");

    form_alert (1, str);

    /* Resource-File aus dem Speicher werfen: */
    rsrc_free();
  }

  gem_exit();
}
```

Assembler

```
          ;
          ; Dialogbox laden, anzeigen, Radio-Buttons verarbeiten
          ; Assembler            MP  17-11-88          DIALOG3.Q
          ;

gemdos    = 1
```

```
DIALOG    = 0       ;Konstanten aus DIALOG3.H2
MW        = 3
UKW       = 2
KW        = 4
LW        = 5
ENDE      = 6

          INCLUDE 'GEM_INEX.Q'

          TEXT

main:     jsr       gem_init

          ; Pfeil als Mauszeiger:

          move.w    #78,control      ;graf_mouse
          move.w    #1,control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          clr.w     int_in           ;0 für Pfeil
          jsr       aes

          ; rsrc_load:

          move.w    #110,control
          clr.w     control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          move.l    #rscname,addr_in
          jsr       aes

          tst.w     int_out          ;Fehler?
          beq       rscerr

          ; rsrc_gaddr ermittelt Startadresse des Dialogs:

          move.w    #112,control
          move.w    #2,control+2
          move.w    #1,control+4
          clr.w     control+6
          move.w    #1,control+8
          clr.w     int_in           ;0 für 'Baum gesucht'
          move.w    #DIALOG,int_in+2 ;Index des Baumes
          jsr       aes
          move.l    addr_out,baum_adr   ;Ergebnis: die Startadresse

          ; Einen Default-Button (UKW) setzen:
```

```
        move.w    #UKW,knopf
        jsr       select

        jsr       show_dialog       ;Anzeigen des Baumes

        ; form_do läßt Dialog abarbeiten:

        move.w    #50,control
        move.w    #1,control+2
        move.w    #1,control+4
        move.w    #1,control+6
        clr.w     control+8
        clr.w     int_in            ;kein Edit-Feld
        move.l    baum_adr,addr_in
        jsr       aes
        move.w    int_out,knopf     ;gedrückter Knopf

        jsr       deselect          ;Selected-Status löschen

        jsr       hide_dialog       ;Dialogbox vom Bildschirm entfernen

        ; Knöpfe auswerten:

        move.w    #UKW,d0
        jsr       selected          ;Knopf gesetzt?
        beq.s     n_ukw
        moveq.l   #0,d1

n_ukw:  move.w    #MW,d0
        jsr       selected
        beq.s     n_mw
        moveq.l   #1,d1

n_mw:   move.w    #KW,d0
        jsr       selected
        beq.s     n_kw
        moveq.l   #2,d1

n_kw:   move.w    #LW,d0
        jsr       selected
        beq.s     n_lw
        moveq.l   #3,d1

n_lw:   asl.w     #2,d1             ;mal 4
        lea       frqtab,a0         ;Tabelle mit Zeiger auf Strings
        movea.l   0(a0,d1.w),a0     ;String mit Frequenzname an
        lea       form_fort,a1      ;form_alert-String anhängen
cpy_lp: move.b    (a0)+,(a1)+
        bne.s     cpy_lp
```

```
          subq.l    #1,a1           ;Nullbyte überschreiben
          lea       form_end,a0     ;mit dem Rest des Stringscpy_l2:
          move.b    (a0)+,(a1)+
          bne.s     cpy_l2

          lea       form_anf,a0
          jsr       form_alert      ;Box anzeigen

          ; rsrc_free:

          move.w    #111,control
          clr.w     control+2
          move.w    #1,control+4
          clr.w     control+6
          clr.w     control+8
          jsr       aes

quit:     jsr       gem_exit

          clr.w     -(sp)
          trap      #gemdos


rscerr:   lea       errtxt,a0
          jsr       form_alert      ;Warnhinweis ausgeben

          bra       quit


show_dialog:
          ; Dieses Unterprogramm malt einen Objekt-Baum auf den
          ; Bildschirm. Dazu muß sich dessen Startadresse unter
          ; 'baum_adr' befinden.

          ; form_center

          move.w    #54,control
          clr.w     control+2
          move.w    #5,control+4
          move.w    #1,control+6
          clr.w     control+8
          move.l    baum_adr,addr_in
          jsr       aes
          move.w    int_out+2,d4    ;Koordinaten sichern
          move.w    int_out+4,d5
          move.w    int_out+6,d6
          move.w    int_out+8,d7

          ; form_dial rettet Fensterränder etc. (0)
```

```
        move.w    #51,control
        move.w    #9,control+2
        move.w    #1,control+4
        clr.w     control+6
        clr.w     control+8
        clr.w     int_in          ;Unterfunktion 0
        ; keine Werte für das kleine Rechteck fo_dilittlx/y/w/h
        move.w    d4,int_in+10    ;großes Rechteck do_dibigx/y/w/h
        move.w    d5,int_in+12
        move.w    d6,int_in+14
        move.w    d7,int_in+16
        jsr       aes

        ; form_dial zeichnet 'Zoom'-Rechteck (1):

        move.w    #51,control
        move.w    #9,control+2
        move.w    #1,control+4
        clr.w     control+6
        clr.w     control+8
        move.w    #1,int_in       ;Unterfunktion 1
        move.w    d4,int_in+2     ;Größe des kleinen Rechtecks
        move.w    d5,int_in+4
        move.w    #1,int_in+6
        move.w    #1,int_in+8
        move.w    d4,int_in+10    ;großes Rechteck do_dibigx/y/w/h
        move.w    d5,int_in+12
        move.w    d6,int_in+14
        move.w    d7,int_in+16
        jsr       aes

        ; Dialog zeichnen mit objc_draw:

        move.w    #42,control
        move.w    #6,control+2
        move.w    #1,control+4
        move.w    #1,control+6
        clr.w     control+8
        clr.w     int_in          ;0=Wurzelobjekt zuerst zeichnen
        move.w    #12,int_in+2    ;max. 12 Ebenen (willkürlich)
        move.w    d4,int_in+4
        move.w    d5,int_in+6
        move.w    d6,int_in+8
        move.w    d7,int_in+10
        move.l    baum_adr,addr_in
        jmp       aes


hide_dialog:
        ; Entfernt das Formular vom Bildschirm.
```

```
; Adresse des Objektbaums wieder in baum_adr

; form_center

move.w    #54,control
clr.w     control+2
move.w    #5,control+4
move.w    #1,control+6
clr.w     control+8
move.l    baum_adr,addr_in
jsr       aes
move.w    int_out+2,d4      ;Koordinaten sichern
move.w    int_out+4,d5
move.w    int_out+6,d6
move.w    int_out+8,d7

; form_dial zeichnet schrumpfendes-Rechteck (2):

move.w    #51,control
move.w    #9,control+2
move.w    #1,control+4
clr.w     control+6
clr.w     control+8
move.w    #2,int_in         ;Unterfunktion 2
move.w    d4,int_in+2       ;Größe des kleinen Rechtecks
move.w    d5,int_in+4
move.w    #1,int_in+6
move.w    #1,int_in+8
move.w    d4,int_in+10      ;großes Rechteck do_dibigx/y/w/h
move.w    d5,int_in+12
move.w    d6,int_in+14
move.w    d7,int_in+16
jsr       aes

; form_dial sendet Redraw-Meldungen an Fenster (3)

move.w    #51,control
move.w    #9,control+2
move.w    #1,control+4
clr.w     control+6
clr.w     control+8
move.w    #3,int_in         ;Unterfunktion 3
; keine Werte für das kleine Rechteck fo_dilittlx/y/w/h
move.w    d4,int_in+10      ;großes Rechteck do_dibigx/y/w/h
move.w    d5,int_in+12
move.w    d6,int_in+14
move.w    d7,int_in+16
jmp       aes
```

```
select:
        ; Schaltet den Button 'knopf' auf 'selected'-Status
        ; Objektbaum muß in baum_adr stehen

        movea.l   baum_adr,a0
        move.w    knopf,d0          ;Index des Objekts
        mulu      #24,d0            ;* 24 (jedes Objekt hat 24 Bytes)
        ori.w     #1,10(a0,d0.w)    ;Bit 0 (selected) setzen
        rts


deselect:
        ; Schaltet den Button 'knopf' auf 'nicht-selected'-Status
        ; Objektbaum muß in baum_adr stehen

        movea.l   baum_adr,a0
        move.w    knopf,d0          ;Index des Objekts
        mulu      #24,d0            ;* 24 (jedes Objekt hat 24 Bytes)
        andi.w    #-2,10(a0,d0.w)   ;Bit 0 (selected) löschen
        rts


selected:
        ; Zero-Flag wird gesetzt, wenn Objekt (Index in d0) nicht
        ; selektiert ist. Aadresse des Objektbaums in baum_adr.

        movea.l   baum_adr,a0
        mulu      #24,d0            ;Index * 24 = Offset
        btst      #0,11(a0,d0.w)    ;btst testet Byte!!!, daher 11
        rts


form_alert:
        ; Zeigt Alert-Box. Beschreibender String steht ab a0.
        ; Knopf Nr. 1 ist Default-Button.

        move.w    #52,control
        move.w    #1,control+2
        move.w    #1,control+4
        move.w    #1,control+6
        clr.w     control+8
        move.w    #1,int_in         ;Default-Button
        move.l    a0,addr_in
        jmp       aes


write_text:
        ; Schreibt String in Text- oder Edit-Objekt (Index in d0)
        ; String in a0, Objektbaum in baum_adr
```

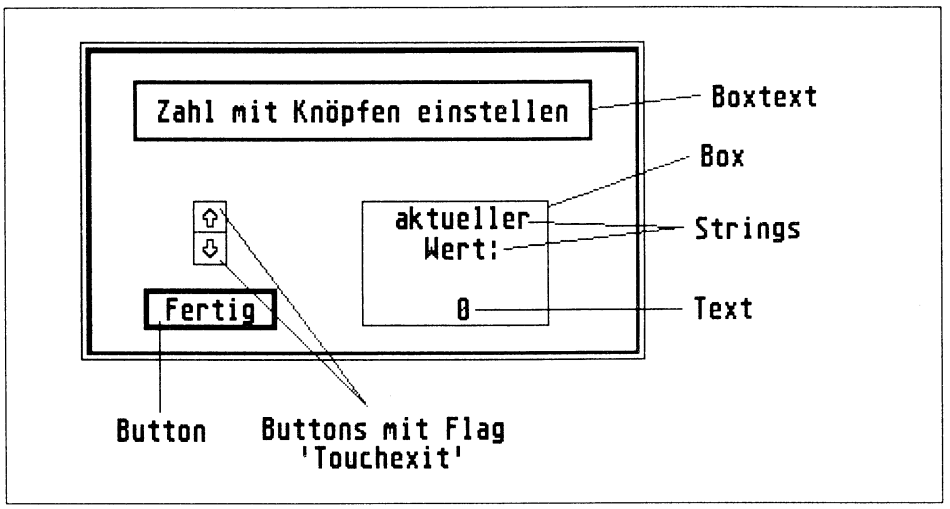
```
          movea.l   baum_adr,a1
          mulu      #24,d0
          movea.l   12(a1,d0.w),a1   ;Adresse des TEDINFO-Blocks
          movea.l   (a1),a1          ;te_ptext enthält den Textwrt_lp:
          move.b    (a0)+,(a1)+
          bne.s     wrt_lp
          rts


read_text:
          ; Gegenstück zu write_text. Zieladresse ist in a0
          ; zu übergeben.

          movea.l   baum_adr,a1
          mulu      #24,d0
          movea.l   12(a1,d0.w),a1
          movea.l   (a1),a1

rd_lp:    move.b    (a1)+,(a0)+
          bne.s     rd_lp
          rts


          DATA

frqtab:   DC.l f_ukw,f_mw,f_kw,f_lw ; Startadressen der Stringsrscname:
          DC.b 'DIALOG3.RSC',0
errtxt:   DC.b '[3][Kein RSC-File!][Ende]',0
f_ukw:    DC.b 'Ultrakurzwelle',0
f_mw:     DC.b 'Mittelwelle',0
f_kw:     DC.b 'Kurzwelle',0
f_lw:     DC.b 'Langwelle',0
form_end: DC.b ' ausgewählt!][Ja!]',0
form_anf: DC.b '[1][Sie haben den Frequenzbereich|'
form_fort:

          BSS

          DS.w 30  ;Platz für das Ende des Alert-Strings
baum_adr: DS.l 1
knopf:    DS.w 1

          END
```

Zum Abschluß möchte ich Ihnen noch eine nette Möglichkeit vorstellen, um die eigenen Programme mit Dialogboxen nicht nur komfortabel zu machen, sondern auch sehr professionell aussehen zu lassen.

Oft müssen in einem Programm irgendwelche Zahlen abgefragt werden. Der Bereich der möglichen Zahlen ist dabei nicht immer groß. Wenn Sie also in Ihrer Textverarbeitung die Zahl der Zeilen pro Seite erfragen wollen, dann wird der Benutzer Werte von vielleicht 30 bis 60 angeben. Sie könnten dazu ein Edit-Objekt für numerische Eingaben (also mit te_pvalid = 99) einrichten und den Benutzer die Eingabe über die Tastatur tätigen lassen. In GEM-Programmen findet man jedoch oft eine andere Möglichkeit, solche Zahlen einzustellen: Mit der Maus können zwei Pfeile angeklickt werden, einer nach oben und einer nach unten. Mit diesen Pfeilen kann die Zahl, die zur Kontrolle immer mit in der Dialogbox erscheint, vergrößert und verkleinert werden. Das hat zwei Vorteile: Erstens kann der Benutzer die Hand auf der Maus lassen, und zweitens ist es einfacher, falsche Eingaben auszuschließen: Wenn nämlich die maximale Zeilenzahl für eine Seite 60 beträgt, dann können Sie jede Vergrößerung über 60 hinaus sperren - bei Edit-Objekten geht das nicht, dort können Sie nur die Zahl der Stellen einer Eingabe begrenzen.

Was brauchen wir für eine solche Dialogbox? Zunächst einmal die Pfeile: Das sind Buttons. Diese Buttons sollen aber, wenn sie angeklickt werden, nicht mehr selektiert werden. Deshalb müssen Sie im RCS das Flag SELECTABLE löschen. Statt dessen müssen diese Objekte einen Exit-Charakter haben, denn beim Klick auf die Buttons muß unsere Applikation kurzzeitig die Kontrolle erhalten; schließlich müssen wir ja die Zahl entsprechend den Wünschen des Anwenders verändern. Allerdings hat das EXIT-Flag, wie wir es bisher kennen, die dumme Eigenschaft, daß der Mausknopf erst wieder losgelassen werden muß, bevor form_do wirklich beendet wird. Außerdem ist ein Exit-Button dicker als normale Knöpfe, und das sieht um die kleinen Pfeile herum nicht gut aus. Deshalb gibt es optional zum EXIT-Flag noch die Eigenschaft TOUCHEXIT, das heißt etwa Ende bei Berührung. Dieses Flag erfüllt all unsere Anforderungen: Der Knopf wird nicht zu dick, und form_do wird beendet, sobald der Mausknopf gedrückt ist (nicht unbedingt wird!). Übrigens: Pfeile in alle Himmelsrichtungen erhalten Sie im RCS dadurch, daß Sie die Buchstaben A, B, C und D zusammen mit der Taste Control drücken.

Auch hierzu möchte ich wieder ein Beispielprogramm geben. Die Dialogbox sieht so aus:



Wie gesagt: Die beiden Pfeil-Knöpfe sind weder EXIT- noch SELECTABLE-Buttons. Ihr einziges Objekt-Flag ist TOUCHEXIT. Die Knöpfe habe ich UP und DOWN genannt. Letzterer Name ist in GFA-BASIC leider nicht zu gebrauchen, deshalb heißt der Button hier DN. Der normale Default- und Exit-Button heißt FERTIG, und das Text-Feld (kein String!), in dem Sie hier eine Null sehen, heißt AKTUELL. Das Text-Objekt hat nur diese eine Ziffer.

Das Programm soll nun die Dialogbox anzeigen und mit form_do auf einen Knopfdruck warten. Wenn FERTIG gedrückt wurde, dann beenden wir das Programm. Wurde aber UP oder DOWN betätigt, so erhöhen/erniedrigen wir die aktuelle Zahl um 1, prüfen, ob sie noch im zulässigen Bereich liegt (in unserem Fall sind Zahlen von 0 bis 9 erlaubt) und schreiben die so erhaltene neue Zahl mit write_text in das Text-Objekt AKTUELL. Nur: Damit steht die Zahl noch nicht auf dem Bildschirm; write_text schreibt sie ja nur in den Objektbaum. Um die Änderung auch auf dem Bildschirm erkennen zu lassen, müssen wir den Baum mit objc_draw noch einmal neu zeichnen. Allerdings nicht den ganzen Baum, sondern nur das Text-Objekt AKTUELL. Deshalb können wir beim objc_draw-Aufruf das erste zu zeichnende Objekt angeben. Es wird dann nur dieses Objekt mit seinen Unterobjekten gezeichnet, und auch die maximale Unterobjekttiefe, also die Zahl der untergeordneten Objektebenen, die gezeichnet werden sollen, läßt sich, das habe ich eingangs des Kapitels 5.9.2 schon einmal erwähnt, bei objc_draw angeben. Wir wählen hier eine Null; damit wird das Objekt AKTUELL ohne Unterobjekte gezeichnet - genau das, was wir brauchen. Anschließend gehen

wir (nach einer kurzen Pause durch evnt_timer, um die Zahl nicht allzu schnell springen zu lassen) wieder in die form_do-Routine und warten auf den nächsten Knopfdruck.

Eine kleine Änderung gegenüber den bisherigen Programmen ist noch zu beachten: Da wir objc_draw auch im Hauptprogramm aufrufen, benötigen wir die Koordinaten der Dialogbox, die wir gewöhnlich in den Unterprogrammen als Abfallprodukt von form_center erhalten haben. Dazu gibt das Unterprogramm show_dialog einfach die Koordinaten in Rückgabevariablen (bzw. im BSS-Segment in Assembler) an das Hauptprogramm zurück.

GFA-BASIC

```
'
' Dialogbox anzeigen / Touchexit-Buttons (UP/DOWN)
' GFA-BASIC         MP  16-12-88       DIALOG4.GFA
'
DEFINT "a-z"   ! alle Variablen 4-Byte-Integer
'
dialog=0       ! Konstanten aus DIALOG4.H2
aktuell=5
dn=7           ! (GFA-BASIC würde aus DOWN machen: DO WHILE)
up=6
fertig=8
'
VOID APPL_INIT()
'
IF RSRC_LOAD("DIALOG4.RSC")=0
  VOID FORM_ALERT(1,"[3][Kein Rsc-File!][Ende]")
ELSE
  '
  ' Startadresse des Baumes erfragen:
  '
  VOID RSRC_GADDR(0,dialog,baum_adr)
  '
  ' Ausgabefeld (aktuell) initialisieren:
  '
  write_text(baum_adr,aktuell,"0")
  zahl=0
  '
  ' Dialogbox anzeigen lassen, Koordinaten merken:
  '
  GOSUB show_dialog(baum_adr,x,y,w,h)
  '
  REPEAT
    '
    ' Dialogbox abarbeiten lassen; 'eingabe' erstes Edit-Feld
```

```
    '
    knopf=FORM_DO(baum_adr,eingabe)
    '
    SELECT knopf
    CASE up
      INC zahl
    CASE dn
      DEC zahl
    ENDSELECT
    '
    ' Auf gültigen Bereich überprüfen:
    '
    SELECT zahl
    CASE 10
      zahl=0
    CASE -1
      zahl=9
    ENDSELECT
    '
    ' Zahl in das Formular schreiben:
    '
    write_text(baum_adr,aktuell,STR$(zahl))
    VOID OBJC_DRAW(baum_adr,aktuell,0,x,y,w,h) ! 0=nur aktuell zeichnen
    '
    ' Kurze Pause (0.2 Sekunden):
    '
    VOID EVNT_TIMER(200)
  UNTIL knopf=fertig
  '
  ' Selected-Status löschen:
  '
  deselect(baum_adr,knopf)
  '
  ' Dialogbox wieder verschwinden lassen:
  '
  GOSUB hide_dialog(baum_adr)
  '
  ' Resource-File aus dem Speicher entfernen:
  '
  VOID RSRC_FREE()
  '
ENDIF
'
VOID APPL_EXIT()
'
END
'
'
PROCEDURE select(baum,index)
  ' Bit 0 setzen:
```

```
  OB_STATE(baum,index)=OB_STATE(baum,index) OR 1
RETURN
'
'
PROCEDURE deselect(baum,index)
  ' Bit 0 löschen:
  OB_STATE(baum,index)=OB_STATE(baum,index) AND -2
RETURN
'
'
DEFFN selected(baum,index)=OB_STATE(baum,index) AND 1
'
'
PROCEDURE show_dialog(baum,VAR x,y,w,h)
  LOCAL x,y,w,h
  '
  ' Formular auf dem Bildschirm zentrieren (wird dabei noch
  ' nicht gezeichnet)
  '
  VOID FORM_CENTER(baum,x,y,w,h)
  '
  ' Fensterränder etc. retten lassen:
  '
  VOID FORM_DIAL(0,x,y,w,h,x,y,w,h)
  '
  ' Zeichnen eines 'Zoom'-Rechtecks:
  '
  VOID FORM_DIAL(1,25,25,25,25,x,y,w,h)
  '
  ' Zeichnen des Formulars:
  ' Start bei Objekt Nr. 0 (Wurzelobjekt, äußerer Kasten/Rahmen)
  ' Tiefe: max. 12 Ebenen (willkürlich)
  '
  VOID OBJC_DRAW(baum,0,12,x,y,w,h)
  '
RETURN
'
'
PROCEDURE hide_dialog(baum)
  LOCAL x,y,w,h
  '
  ' Nochmal die Koordinaten erfragen:
  '
  VOID FORM_CENTER(baum,x,y,w,h)
  '
  ' Zeichnen eines kleiner werdenden Rechtecks:
  '
  VOID FORM_DIAL(2,25,25,25,25,x,y,w,h)
  '
  ' Fensterränder wiederherstellen und Redraw-Meldung
```

```
  ' an alle zerstörten Fenster veranlassen:
  '
  VOID FORM_DIAL(3,x,y,w,h,x,y,w,h)
  '
RETURN
'
'
PROCEDURE write_text(baum,index,text$)
  LOCAL adr,i,a$
  '
  ' Startadresse des reinen Textes (te_ptext) ermitteln:
  '
  adr=LPEEK(OB_SPEC(baum,index))
  '
  ' Nullbyte anhängen:
  '
  a$=text$+CHR$(0)
  '
  FOR i=1 TO LEN(a$)
    POKE adr+i-1,ASC(MID$(a$,i,1))
  NEXT i
RETURN
'
'
PROCEDURE read_text(baum,index,VAR text$)
  LOCAL adr,i
  '
  ' te_ptext ermitteln:
  '
  adr=LPEEK(OB_SPEC(baum,index))
  '
  text$=""
  i=0
  '
  WHILE PEEK(adr+i)<>0    ! bis zum Nullbyte lesen
    text$=text$+CHR$(PEEK(adr+i))
    INC i
  WEND
RETURN
'
```

Omikron-BASIC

```
'
' Dialogbox anzeigen / Touchexit-Buttons (UP/DOWN)
' GFA-BASIC          MP  16-12-88        DIALOG4.GFA
'
'
Dialog%L=0'        Konstanten aus DIALOG4.H2
Aktuell%L=5
```

```
Down%L=7
Up%L=6
Fertig%L=8
'
Appl_Init
'
Rsrc_Load("DIALOG4.RSC",Ret%L)
IF Ret%L=0 THEN
 FORM_ALERT (1,"[3][Kein Rsc-File!][Ende]")
ELSE
 '
 ' Startadresse des Baumes erfragen:
 '
 Rsrc_Gaddr(0,Dialog%L,Baum_Adr%L)
 '
 ' Ausgabefeld (aktuell) initialisieren:
 '
 Write_Text(Baum_Adr%L,Aktuell%L,"0")
 Zahl%L=0
 '
 ' Dialogbox anzeigen lassen, Koordinaten merken:
 '
 Show_Dialog(Baum_Adr%L,X%L,Y%L,W%L,H%L)
 '
 REPEAT
   '
   ' Dialogbox abarbeiten lassen
   '
   Form_Do(0,Baum_Adr%L,Knopf%L)
   '
   IF Knopf%L=Up%L THEN Zahl%L=Zahl%L+1
   IF Knopf%L=Down%L THEN Zahl%L=Zahl%L-1
   '
   ' Auf gültigen Bereich überprüfen:
   '
   IF Zahl%L=10 THEN Zahl%L=0
   IF Zahl%L=-1 THEN Zahl%L=9
   '
   ' Zahl in das Formular schreiben:
   '
   Write_Text(Baum_Adr%L,Aktuell%L, RIGHT$( STR$(Zahl%L),1))
   Objc_Draw(Aktuell%L,0,X%L,Y%L,W%L,H%L,Baum_Adr%L)' 0=keine Unterobj.
   '
   ' Kurze Pause (0.2 Sekunden):
   '
   Evnt_Timer(200)
 UNTIL Knopf%L=Fertig%L
 '
 ' Selected-Status löschen:
 '
```

```
 Deselect(Baum_Adr%L,Knopf%L)
 '
 ' Dialogbox wieder verschwinden lassen:
 '
 Hide_Dialog(Baum_Adr%L,X%L,Y%L,W%L,H%L)
 '
 ' Resource-File aus dem Speicher entfernen:
 '
 Rsrc_Free
 '
ENDIF
'
Appl_Exit
'
END
'
'
DEF PROC Select(Baum%L,Index%L)
 ' Bit 0 setzen:
 WPOKE Baum%L+24*Index%L+10, WPEEK(Baum%L+24*Index%L+10) OR 1
RETURN
'
'
DEF PROC Deselect(Baum%L,Index%L)
 ' Bit 0 löschen:
 WPOKE Baum%L+24*Index%L+10, WPEEK(Baum%L+24*Index%L+10) AND -2RETURN
'
'
DEF FN Selected%L(Baum%L,Index%L)= WPEEK(Baum%L+24*Index%L+10) AND 1
'
'
DEF PROC Show_Dialog(Baum%L,R X%L,R Y%L,R W%L,R H%L)
 '
 ' Formular auf dem Bildschirm zentrieren (wird dabei noch
 ' nicht gezeichnet)
 '
 Form_Center(Baum%L,X%L,Y%L,W%L,H%L)
 '
 ' Fensterränder etc. retten lassen:
 '
 Form_Dial(0,X%L,Y%L,W%L,H%L)
 '
 ' Zeichnen eines 'Zoom'-Rechtecks:
 '
 Form_Dial(1,X%L,Y%L,W%L,H%L)
 '
 ' Zeichnen des Formulars:
 ' Start bei Objekt Nr. 0 (Wurzelobjekt, äußerer Kasten/Rahmen)
 ' Tiefe: max. 12 Ebenen (willkürlich)
 '
```

```
 Objc_Draw(0,12,X%L,Y%L,W%L,H%L,Baum%L)
 '
RETURN
'
'
DEF PROC Hide_Dialog(Baum%L,X%L,Y%L,W%L,H%L)
 '
 ' Zeichnen eines kleiner werdenden Rechtecks:
 '
 Form_Dial(2,X%L,Y%L,W%L,H%L)
 '
 ' Fensterränder wiederherstellen und Redraw-Meldung
 ' an alle zerstörten Fenster veranlassen:
 '
 Form_Dial(3,X%L,Y%L,W%L,H%L)
 '
RETURN
'
'
DEF PROC Write_Text(Baum%L,Index%L,Text$)
 LOCAL Adr%L,I%L,A$
 '
 ' Startadresse des reinen Textes (te_ptext) ermitteln:
 '
 Adr%L= LPEEK( LPEEK(Baum%L+24*Index%L+12))
 '
 ' Nullbyte anhängen:
 '
 A$=Text$+ CHR$(0)
 '
 FOR I%L=1 TO LEN(A$)
   POKE Adr%L+I%L-1, ASC( MID$(A$,I%L,1))
 NEXT I%L
RETURN
'
'
DEF PROC Read_Text(Baum%L,Index%L,R Text$)
 LOCAL Adr%L,I%L
 '
 ' te_ptext ermitteln:
 '
 Adr%L= LPEEK( LPEEK(Baum%L+24*Index%L+12))
 '
 Text$=""
 I%L=0
 '
 WHILE PEEK(Adr%L+I%L)<>0'         bis zum Nullbyte lesen
   Text$=Text$+ CHR$( PEEK(Adr%L+I%L))
   I%L=I%L+1
 WEND
```

```
RETURN
'
```

C

```
/*********************************************************/
/*   Dialogbox anzeigen / Touchexit-Buttons (UP/DOWN)    */
/*   Megamax Laser C     MP  14-11-88       DIALOG4.C    */
/*********************************************************/

#include <obdefs.h>     /* GEM-Objekt-Definitionen */

#include "gem_inex.c"
#include "dialog4.h"    /* Header-File der Resource-Datei */

OBJECT *baum_adr;       /* pointer auf das erste Objekt eines Baumes */
int     knopf,
        zahl,
        x, y, w, h;
char    eingabe[80],
        *str = "0";


select (baum, index)          /* schaltet Button 'object' ein */
OBJECT baum[];
int    index;
{
  baum[index].ob_state |= 1;     /* Bit 0 in ob_state setzen */
}


deselect (baum, index)       /* schaltet Button 'object' aus */OBJECT baum[];
int    index;
{
  baum[index].ob_state &= -2;    /* Bit 0 in ob_state löschen */
}


show_dialog (baum, x, y, w, h)  /* Darstellen einer Dialogbox */
OBJECT *baum;
int  *x, *y, *w, *h;
{
  /* Formular auf dem Bildschirm zentrieren. Dabei werden nur */
  /* Koordinaten intern an die Bildschirmauflösung angepaßt,  */
  /* es wird also noch nichts gezeichnet. Außerdem erhalten   */
  /* wir die zukünftigen Koordinaten der Dialogbox.           */

  form_center (baum, x, y, w, h);

  /* Fensterränder etc. retten lassen: */
```

```
  form_dial (0, *x, *y, *w, *h, *x, *y, *w, *h);

  /* Zeichnen eines 'Zoom'-Rechtecks */
  form_dial (1, 25, 25, 25, 25, *x, *y, *w, *h);

  /* Zeichnen des Objektbaumes selbst */
  /* Start bei Objekt Nr. 0 (Wurzel, äußerer Kasten) */
  /* Tiefe: max. 12 Ebenen (willkürlich) */
  objc_draw (baum, 0, 12, *x, *y, *w, *h);
}


hide_dialog (baum)
OBJECT *baum;
{
int  x, y, w, h;

  /* Nochmal die Koordinaten erfragen: */
  form_center (baum, &x, &y, &w, &h);

  /* Zeichnen eines kleiner werdenden Rechtecks */
  form_dial (2, 25, 25, 25, 25, x, y, w, h);

  /* Fensterränder wieder herstellen lassen und Redraw- */
  /* Meldung an alle zerstörten Fenster veranlassen     */
  form_dial (3, x, y, w, h, x, y, w, h);
}


write_text (baum, index, string)        /* Ändert TEXT-Objekt */
OBJECT baum[];
int    index;
char   string[];
{
TEDINFO *ted;

  ted = (TEDINFO *) baum[index].ob_spec;
  strcpy (ted->te_ptext, string);

  /* ted->te_ptext   entspricht:   (*ted).te_ptext */
}


main()
{
  gem_init();

  /* Resource-File (DIALOG4.RSC) laden */

  if (rsrc_load ("DIALOG4.RSC") == 0)
```

```
  form_alert (1, "[3][Kein RSC-File!][Ende]");
else
{
  /* Startadresse des (0 =) Baums DIALOG festellen */
  rsrc_gaddr (0, DIALOG, &baum_adr);

  /* Ausgabefeld (AKTUELL) initialisieren: */
  write_text (baum_adr, AKTUELL, "0");

  /* Dialogbox anzeigen lassen: */
  show_dialog (baum_adr, &x, &y, &w, &h);

  do
  {
    /* Dialog abarbeiten lassen */
    knopf = form_do (baum_adr, 0);

    switch (knopf)    /* Pfeil-Knöpfe abfragen */
    {
      case UP:   zahl++;
                 break;

      case DOWN: zahl--;
                 break;
    }

    switch (zahl)    /* Auf Grenzen prüfen */
    {
      case -1:  zahl = 9;
                break;

      case 10:  zahl = 0;
                break;
    }

    /* Zahl in das Feld schreiben: */
    str[0] = '0' + zahl;
    write_text (baum_adr, AKTUELL, str);

    /* nur das Textfeld neu zeichnen lassen (0 Unter-Ebenen) */
    objc_draw (baum_adr, AKTUELL, 0, x, y, w, h);

    /* Kurze Pause (0.2 Sekunden) */
    evnt_timer (200, 0);

  } while (knopf != FERTIG);

  /* Knopf deselektieren */
  deselect (baum_adr,knopf);
```

```
    /* Dialogbox wieder verschwinden lassen */
    hide_dialog (baum_adr);

    /* Resource-File aus dem Speicher werfen: */
    rsrc_free();
  }

  gem_exit();
}
```

Assembler

```
          ;
          ; Dialogbox anzeigen / Touchexit-Buttons (UP/DOWN)
          ; Assembler          MP  17-11-88        DIALOG4.Q
          ;

gemdos    = 1

DIALOG    = 0       ;Konstanten aus DIALOG4.H2
AKTUELL   = 5
DOWN      = 7
UP        = 6
FERTIG    = 8

          INCLUDE 'GEM_INEX.Q'

          TEXT

main:     jsr       gem_init

          ; Pfeil als Mauszeiger:

          move.w    #78,control      ;graf_mouse
          move.w    #1,control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          clr.w     int_in           ;0 für Pfeil
          jsr       aes

          ; rsrc_load:

          move.w    #110,control
          clr.w     control+2
          move.w    #1,control+4
          move.w    #1,control+6
          clr.w     control+8
          move.l    #rscname,addr_in
```

```
        jsr     aes

        tst.w   int_out         ;Fehler?
        beq     rscerr

        ; rsrc_gaddr ermittelt Startadresse des Dialogs:

        move.w  #112,control
        move.w  #2,control+2
        move.w  #1,control+4
        clr.w   control+6
        move.w  #1,control+8
        clr.w   int_in          ;0 für 'Baum gesucht'
        move.w  #DIALOG,int_in+2 ;Index des Baumes
        jsr     aes
        move.l  addr_out,baum_adr   ;Ergebnis: die Startadresse

        ; Ausgabefeld (AKTUELL) initialisieren:

        lea     ziffstr,a0      ;zeigt anfangs auf '0',0
        move.w  #AKTUELL,d0
        jsr     write_text

        jsr     show_dialog     ;Anzeigen des Baumes

        ; form_do läßt Dialog abarbeiten:

loop:   move.w  #50,control
        move.w  #1,control+2
        move.w  #1,control+4
        move.w  #1,control+6
        clr.w   control+8
        clr.w   int_in          ;kein Edit-Feld
        move.l  baum_adr,addr_in
        jsr     aes
        move.w  int_out,knopf   ;gedrückter Knopf

        cmpi.w  #FERTIG,knopf
        beq     fine

        cmpi.w  #UP,knopf       ;Knop 'UP' gedrückt?
        bne     not_up
        addq.b  #1,ziffstr      ;dann 1 zur Ziffer addieren
        cmpi.b  #'9',ziffstr    ;auf Grenzen abchecken
        ble.s   go_on           ;>9?
        move.b  #'0',ziffstr    ;dann eine 0 draus machen
        bra     go_on

not_up: cmpi.w  #DOWN,knopf     ;oder 'DOWN'?
        bne     loop            ;nein, dann ignorieren
```

```
        subq.b    #1,ziffstr        ;minus 1
        cmpi.b    #'0',ziffstr      ;<0?
        bge.s     go_on
        move.b    #'9',ziffstr      ;dann eine 9 reinschreibengo_on:
        lea       ziffstr,a0        ;Ziffer in Formular 'eintragen'
        move.w    #AKTUELL,d0
        jsr       write_text

        ; objc_draw zeichnet AKTUELL neu/keine Unterobjekte (0)

        move.w    #42,control
        move.w    #6,control+2
        move.w    #1,control+4
        move.w    #1,control+6
        clr.w     control+8
        move.w    #AKTUELL,int_in   ;erstes Objekt ist Ziffer
        clr.w     int_in+2          ;keine Unterobjekte
        move.w    x,int_in+4
        move.w    y,int_in+6
        move.w    w,int_in+8
        move.w    h,int_in+10
        move.l    baum_adr,addr_in
        jsr       aes

        ; evnt_timer wartet 200 ms

        move.w    #24,control
        move.w    #2,control+2
        move.w    #1,control+4
        clr.w     control+6
        clr.w     control+8
        move.w    #200,int_in       ;Low-Word
        clr.w     int_in+2          ;und High-Word von 200
        jsr       aes

        bra       loop


fine:   jsr       deselect          ;Selected-Status löschen

        jsr       hide_dialog       ;Dialogbox vom Bildschirm entfernen

        ; rsrc_free:

        move.w    #111,control
        clr.w     control+2
        move.w    #1,control+4
        clr.w     control+6
        clr.w     control+8
```

```
        jsr     aes

quit:   jsr     gem_exit

        clr.w   -(sp)
        trap    #gemdos


rscerr: lea     errtxt,a0
        jsr     form_alert      ;Warnhinweis ausgeben

        bra     quit


show_dialog:
        ; Dieses Unterprogramm malt einen Objektbaum auf den
        ; Bildschirm. Dazu muß sich dessen Startadresse unter
        ; 'baum_adr' befinden.

        ; Neu: Die Koordinaten werden später noch im Hauptprogramm
        ; benötigt und daher im bss-Segment gespeichert
        ; (nicht mehr in Registern wie bisher)

        ; form_center

        move.w  #54,control
        clr.w   control+2
        move.w  #5,control+4
        move.w  #1,control+6
        clr.w   control+8
        move.l  baum_adr,addr_in
        jsr     aes
        move.w  int_out+2,x     ;Koordinaten sichern
        move.w  int_out+4,y
        move.w  int_out+6,w
        move.w  int_out+8,h

        ; form_dial rettet Fensterränder etc. (0)

        move.w  #51,control
        move.w  #9,control+2
        move.w  #1,control+4
        clr.w   control+6
        clr.w   control+8
        clr.w   int_in          ;Unterfunktion 0
        ; keine Werte für das kleine Rechteck fo_dilittlx/y/w/h
        move.w  x,int_in+10     ;großes Rechteck do_dibigx/y/w/h
        move.w  y,int_in+12
        move.w  w,int_in+14
        move.w  h,int_in+16
```

```
        jsr     aes

        ; form_dial zeichnet 'Zoom'-Rechteck (1):

        move.w  #51,control
        move.w  #9,control+2
        move.w  #1,control+4
        clr.w   control+6
        clr.w   control+8
        move.w  #1,int_in       ;Unterfunktion 1
        move.w  x,int_in+2      ;Größe des kleinen Rechtecks
        move.w  y,int_in+4
        move.w  #1,int_in+6
        move.w  #1,int_in+8
        move.w  x,int_in+10     ;großes Rechteck do_dibigx/y/w/h
        move.w  y,int_in+12
        move.w  w,int_in+14
        move.w  h,int_in+16
        jsr     aes

        ; Dialog zeichnen mit objc_draw:

        move.w  #42,control
        move.w  #6,control+2
        move.w  #1,control+4
        move.w  #1,control+6
        clr.w   control+8
        clr.w   int_in          ;0=Wurzelobjekt zuerst zeichnen
        move.w  #12,int_in+2    ;max. 12 Ebenen (willkürlich)
        move.w  x,int_in+4
        move.w  y,int_in+6
        move.w  w,int_in+8
        move.w  h,int_in+10
        move.l  baum_adr,addr_in
        jmp     aes


hide_dialog:
        ; Entfernt das Formular vom Bildschirm.
        ; Adresse des Objektbaums wieder in baum_adr

        ; form_dial zeichnet schrumpfendes-Rechteck (2):

        move.w  #51,control
        move.w  #9,control+2
        move.w  #1,control+4
        clr.w   control+6
        clr.w   control+8
        move.w  #2,int_in       ;Unterfunktion 2
        move.w  x,int_in+2      ;Größe des kleinen Rechtecks
```

```
        move.w    y,int_in+4
        move.w    #1,int_in+6
        move.w    #1,int_in+8
        move.w    x,int_in+10       ;großes Rechteck do_dibigx/y/w/h
        move.w    y,int_in+12
        move.w    w,int_in+14
        move.w    h,int_in+16
        jsr       aes

        ; form_dial sendet Redraw-Meldungen an Fenster (3)

        move.w    #51,control
        move.w    #9,control+2
        move.w    #1,control+4
        clr.w     control+6
        clr.w     control+8
        move.w    #3,int_in         ;Unterfunktion 3
        ; keine Werte für das kleine Rechteck fo_dilittlx/y/w/h
        move.w    x,int_in+10       ;großes Rechteck do_dibigx/y/w/h
        move.w    y,int_in+12
        move.w    w,int_in+14
        move.w    h,int_in+16
        jmp       aes


select:
        ; Schaltet den Button 'knopf' auf 'selected'-Status
        ; Objektbaum muß in baum_adr stehen

        movea.l   baum_adr,a0
        move.w    knopf,d0          ;Index des Objekts
        mulu      #24,d0            ;* 24 (jedes Objekt hat 24 Bytes)
        ori.w     #1,10(a0,d0.w)    ;Bit 0 (selected) setzen
        rts


deselect:
        ; Schaltet den Button 'knopf' auf 'nicht-selected'-Status
        ; Objektbaum muß in baum_adr stehen

        movea.l   baum_adr,a0
        move.w    knopf,d0          ;Index des Objekts
        mulu      #24,d0            ;* 24 (jedes Objekt hat 24 Bytes)
        andi.w    #-2,10(a0,d0.w)   ;Bit 0 (selected) löschen
        rts


selected:
        ; Zero-Flag wird gesetzt, wenn Objekt (Index in d0) nicht
```

```
        ; selektiert ist. Aadresse des Objektbaums in baum_adr.

        movea.l   baum_adr,a0
        mulu      #24,d0            ;Index * 24 = Offset
        btst      #0,11(a0,d0.w)    ;btst testet Byte!!!, daher 11
        rts


form_alert:
        ; Zeigt Alert-Box. Beschreibender String steht ab a0.
        ; Knopf Nr. 1 ist Default-Button.

        move.w    #52,control
        move.w    #1,control+2
        move.w    #1,control+4
        move.w    #1,control+6
        clr.w     control+8
        move.w    #1,int_in         ;Default-Button
        move.l    a0,addr_in
        jmp       aes


write_text:
        ; Schreibt String in Text- oder Edit-Objekt (Index in d0)
        ; String in a0, Objektbaum in baum_adr

        movea.l   baum_adr,a1
        mulu      #24,d0
        movea.l   12(a1,d0.w),a1    ;Adresse des TEDINFO-Blocks
        movea.l   (a1),a1           ;te_ptext enthält den Textwrt_lp:
        move.b    (a0)+,(a1)+
        bne.s     wrt_lp
        rts


read_text:
        ; Gegenstück zu write_text. Zieladresse ist in a0
        ; zu übergeben.

        movea.l   baum_adr,a1
        mulu      #24,d0
        movea.l   12(a1,d0.w),a1
        movea.l   (a1),a1

rd_lp:  move.b    (a1)+,(a0)+
        bne.s     rd_lp
        rts
```

```
          DATA

ziffstr:  DC.b '0',0
rscname:  DC.b 'DIALOG4.RSC',0
errtxt:   DC.b '[3][Kein RSC-File!][Ende]',0

          BSS

x:        DS.w 1   ;Koordinaten der Dialogbox
y:        DS.w 1
w:        DS.w 1
h:        DS.w 1

baum_adr: DS.l 1
knopf:    DS.w 1

          END
```

5.10 Accessories

Wenn der ST-Besitzer das Wort Accessory hört, dann denkt er sofort an die kleinen nützlichen Hilfsprogramme wie Kontrollfeld, Drucker-Anpassung oder Taschenrechner. Ich will Ihnen nun zeigen, wie Sie solche Programme selbst schreiben können.

Accessories sind Programme, die sich beim Booten auf der Diskette (im Hauptverzeichnis) befinden und die Extension .ACC tragen. Die Programme kann man dann aus jeder GEM-Anwendung heraus aufrufen, sobald eine Menüleiste vorhanden ist. Accessories müssen übrigens keine kleinen Programme sein: Wenn Sie wollen, können Sie durchaus ein Grafik-Programm oder eine Textverarbeitung als Accessory programmieren. Allerdings sollten Sie beachten, daß diese Programme ständig Speicherplatz belegen, ob sie nun gebraucht werden oder nicht, was freilich den MEGA-ST4-Besitzer nicht hindern sollte, ruhig etwas umfangreiche Accessories zu schreiben.Accessories müssen grundsätzlich Stand-Alone-Programme sein, das heißt die BASIC-Fans müssen ihre Programme compilieren. Leider lag mir vor dem Erscheinen dieses Buches der GFA-BASIC-Compiler 3.0 noch nicht vor, so daß ich Ihnen für diese Sprache leider kein Beispiel anbieten kann. Die allgemeinen Erläuterungen sind jedoch mit einiger Sicherheit auch für GFA-BASIC gültig, sobald der Compiler erscheint.

Ein Accessory besteht immer aus zwei Teilen: Einer Initialisierung, in der sich das Programm einen Platz in der Menüleiste reservieren lassen muß und auch schon den Namen angibt, unter dem es dort später erscheinen

will, und einer Endlosschleife, die immer nur darauf wartet, daß das Accessory aus seinem Schlaf geweckt wird. Wenn der Benutzer also das Accessory aufruft, so wird der Teil des Programms innerhalb dieser Endlosschleife ausgeführt und anschließend weiter gewartet. Accessories enden aufgrund dieser Endlosschleife nie. Deshalb können Sie auch nur beim Booten vom GEM in den Speicher geladen werden; diese Laderoutine unterbricht das Programm in der Endlosschleife, lädt das nächste Accessory usw. Spaßeshalber können Sie ja einmal ein Accessory umbenennen, so daß die Extension nicht mehr .ACC sondern .PRG lautet, und es dann starten - der Rechner bleibt in der Endlos-Schleife hängen.

Betrachten wir den ersten Teil eines Accessories: die Initialisierung. Jedes Accessory muß, genau wie eine Applikation, mit appl_init eröffnet werden (in C und Assembler macht das wieder die Include-Datei, die wir ja auch bisher immer benutzt haben). Allerdings benötigen wir diesmal die Identifikations-Nummer des Programms, die als Funktionswert von appl_init zurückgegeben wird. In GEM_INEX ist dazu schon die globale Variable ap_id (Applikations-ID) deklariert; wir brauchen uns darum also nicht zu kümmern.

Anschließend müssen wir uns um einen Platz in der Menüleiste bewerben. Das geht mit der AES-Funktion menu_register, die als Parameter die Programm-ID (ap_id) erhält. Zusätzlich geben wir den String an, der als Menü-Eintrag im Desk-Menü erscheinen soll. Dabei ist es üblich, diesem Namen zwei Leerzeichen voranzustellen.

Diese Funktion menu_register gibt eine weitere Identifikationsnummer zurück: ac_id, das heißt: Accessory-ID. Mit diesem Wert unterscheidet GEM zwischen den maximal sechs möglichen Accessories. Wenn dieser Funktionswert gleich -1 wird, dann bedeutet das übrigens, daß die Menüleiste schon voll war, unser Antrag also abgelehnt wurde. In diesem Fall sollten wir natürlich nicht in die Endlosschleife einsteigen, sondern das Programm wie eine Applikation verlassen.

Bevor wir zum Hauptteil eines Accessories kommen, möchte ich noch kurz erläutern, was es in den einzelnen Sprachen Besonderes zu beachten gibt:

Omrikon-BASIC

Sie erhalten bei appl_init keine ap_id. Dafür müssen Sie diese ID auch bei menu_register nicht angeben; der Wert wird sozusagen intern automatisch weitergeleitet.

Wichtiger ist jedoch, wie Sie aus einem Quelltext ein lauffähiges Accessory machen. Dazu müssen Sie, wie schon gesagt, im Besitz des Omikron-Compilers sein. Schreiben Sie Ihr Programm wie gewohnt, allerdings ohne vorher die Datei GEMLIB.BAS geladen zu haben. Testläufe können Sie selbstverständlich im Interpreter nicht durchführen. Deshalb bietet es sich an, die eigentliche Routine, die Sie als Accessory schreiben wollen, zunächst als Applikation zu entwickeln und erst später, wenn alles wie gewünscht funktioniert, daraus ein Accessory zu machen.

So, Sie haben also jetzt einen Accessory-Quelltext. Den speichern Sie ohne GEMLIB als ASCII-Text ab, also mit dem Befehl MERGE. Die Extension dieser Datei sollte .BAS sein. Als nächstes laden Sie das Programm GEMSEL.BAS (gehört zu Omikron-BASIC) in den Speicher und starten es. Sie werden nach dem Dateinamen Ihres Quelltextes und nach dem Pfadnamen der GEMLIB.BAS-Datei gefragt. Das Programm hängt nun alle benötigten Routinen aus der GEMLIB-Datei an Ihren Quelltext an und speichert diesen unter dem alten Namen (wieder als ASCII-Text).

Bevor Sie das Programm compilieren können, müssen Sie diesen ASCII-Text wieder in das Standard-BASIC-Format übertragen. Laden Sie dazu den ASCII-Text ein, und speichern Sie ihn ganz normal mit SAVE unter gleichem Namen wieder ab. Anschließend starten Sie den Compiler und übersetzen das Programm. Auf der Compiler-Diskette gibt es nun noch ein Programm namens CUTLIB.PRG, das die Library BASLIB teilweise an das compilierte Programm anhängt. Schließlich benennen Sie das fertige Programm noch in NAME.ACC um, kopieren das Programm auf eine Test-Diskette und drücken den Resetknopf. Wenn alles richtig ist, können Sie das Accessory gleich ausprobieren.

C

Beim Megamax-Laser-C-System sind Sie es gewohnt, ein Programm aus dem Editor heraus mit Control und R compilieren, linken und starten zu lassen. Das geht bei Accessories natürlich nicht; der Rechner würde ja in der Endlosschleife steckenbleiben. Speichern Sie statt dessen mit Control und S Ihren Quelltext ab und lassen Sie den Compiler im Menü Execute Ihr Programm bearbeiten.

Als nächstes starten Sie den Linker im gleichen Menü. Sie müssen die Dateien INIT.O und Ihr Accessory-.O-File linken. Unten in der Dialogbox tragen Sie dann noch ein, wie Ihr Programm heißen soll. Der Name muß hier schon auf .ACC enden, damit der Linker weiß, daß er ein Ac-

cessory linken soll. Wenn Sie keine Fehlermeldung erhalten, dann kopieren Sie die ACC-Datei auf eine Test-Diskette, und booten Sie den Rechner.

Assembler

In Assembler schreiben Sie Ihr Programm wie gewohnt. Allerdings benötigen Sie eine veränderte Initialisierungs-Datei. Das kommt daher, daß ein Accessory niemals den nicht benötigten Speicherplatz mit Setblock bzw. Mshrink wieder freigeben darf. Einen Stack richten wir uns einfach im BSS-Segment ein. Auch mit dieser neuen Include-Datei (ACC_INIT.Q) sollte der erste Assembler-Befehl im Hauptprogramm jsr gem_init sein. Sie können Ihre Programme wie üblich assemblieren oder auch linken lassen, müssen dann aber die Extension ändern (aus .PRG wird .ACC).

Die neue Header-Datei sieht so aus:

```
;
; Include-Datei für AES/VDI Anmeldung bei Accessory
; Assembler          MP  13-10-88        ACC_INIT.Q
;

;Diese Datei kann in eigenen Applikationen mit Include
;verwendet werden. Die Include-Anweisung sollte der
;erste Befehl im Assembler-Quelltext sein.

;Unterschied zu GEM_INIT und GEM_INEX:
;Ein Accessory darf nicht mit Setblock/Mshrink Speicher
;freigeben. Nur ein Stack von ausreichender Größe (4 KB)
;wird eingerichtet.


              TEXT

              lea     stackend,sp  ;Stack einrichten
              jmp     main         ;Sprung in die Applikation

; Unterprogramme aes und vdi

aes:          move.l  #aespb,d1    ;AES-Parameterblock
              move.w  #$c8,d0      ;Magic-Number für AES
              trap    #2           ;GEM-Aufruf
              rts

vdi:          move.l  #vdipb,d1    ;VDI-Parameterblock
              move.w  #$73,d0      ;Code für VDI
              trap    #2           ;GEM-Aufruf
```

```
            rts

gem_init:           ;vor dem ersten GEM-Call aufrufen

;Anmeldung beim AES (appl_init):

            move.w  #10,control  ;appl_init (AES)
            clr.w   control+2
            move.w  #1,control+4
            clr.w   control+6
            clr.w   control+8
            jsr     aes
            move.w  int_out,ap_id      ;Identifikation merken

;Bildschirm als Arbeitsstation anmelden (VDI):

            moveq.l #18,d0       ;intin vorbereiten
            lea     intin,a0
gi_lp:      move.w  #1,0(a0,d0.w)      ;Element auf 1 setzen
            subq.w  #2,d0        ;voriges Element
            bpl.s   gi_lp        ;Ende?
            move.w  #2,20(a0)    ;Koordinatenflag (immer 2)

            move.w  #100,contrl  ;v_opnvwk (VDI)
            clr.w   contrl+2
            move.w  #12,contrl+4
            move.w  #11,contrl+6
            move.w  #45,contrl+8
            jsr     vdi
            move.w  contrl+12,handle   ;VDI-Grafik-Handle

            move.w  intout,x_max ;Auflösung speichern
            move.w  intout+2,y_max
            rts


gem_exit:           ;vor Verlassen des Programms aufrufen

            move.w  #101,contrl  ;v_clsvwk (VDI)
            clr.w   contrl+2
            clr.w   contrl+4
            clr.w   contrl+6
            clr.w   contrl+8
            move.w  handle,contrl+12
            jsr     vdi

            move.w  #19,control  ;appl_exit (AES)
            clr.w   control+2
            move.w  #1,control+4
            clr.w   control+6
```

```
            clr.w   control+8
            jsr     aes
            rts

; Es folgen die Parameterblöcke:

            DATA

aespb:      DC.l control
            DC.l global
            DC.l int_in         ;Unterschied zwischen AES- und
            DC.l int_out        ;VDI-Integer-Arrays: _
            DC.l addr_in
            DC.l addr_out

vdipb:      DC.l contrl
            DC.l intin
            DC.l ptsin
            DC.l intout
            DC.l ptsout

; Jetzt kommen die eigentlichen Arrays:

            BSS

global:     DS.w 16             ;AES
control:    DS.w 10
int_in:     DS.w 128
int_out:    DS.w 128
addr_in:    DS.l 128
addr_out:   DS.l 128

contrl:     DS.w 12             ;VDI
intin:      DS.w 128
ptsin:      DS.w 128
intout:     DS.w 128
ptsout:     DS.w 128

stack:      DS.w 2000
stackend:

ap_id:      DS.w 1 ;ap_id, wird von appl_init geliefert

handle:     DS.w 1 ;VDI-Grafik-Handle

x_max:      DS.w 1 ;Bildschirmauflösung, erfährt man
y_max:      DS.w 1 ;bei v_opnvwk

            END
```

Das Folgende gilt wieder für alle Programmiersprachen gleichermaßen. Wenn Sie so ein Accessory in der Initialisierung angemeldet haben, dann muß es in eine Endlos-Warteschleife gehen. Aber worauf warten wir eigentlich? Auf den Aufruf unseres Accessories durch den Anwender, und genau das teilt uns GEM über eine Nachricht mit. Die Nachrichten kennen Sie schon von der Fensterprogrammierung: Wir brauchen einen Nachrichten-Puffer, der acht Worte aufnehmen kann (in Omrikon-BASIC reicht ein String); die Nachricht selbst bekommen wir durch einen evnt_mesag-Aufruf.

Die Nachricht, auf die wir warten müssen, heißt AC_OPEN. Wir erkennen Sie daran, daß im Wort 0 des Puffers die Zahl 40 steht. Außerdem müssen wir überprüfen, ob im Wort 4 die Accessory-Identifikationsnummer unseres Programms steht (ac_id); diese Zahl haben wir ja durch menu_register erhalten. Wenn das so ist, dann starten wir das eigentliche Accessory. Anschließend gehen wir zurück zum evnt_mesag-Aufruf und warten auf die nächste Nachricht.

Dazu gleich einmal ein Beispielprogramm: Ein Accessory, das lediglich eine Alert-Box auf den Bildschirm zaubert. Sie können es gut als Grundlage für eigene Accessories benutzen; denn die eigentliche Aufgabe des Programms, also das Anzeigen der Alert-Box, wird in einem Unterprogramm (go_accessory) vorgenommen. Aber Achtung: Dieses Programm prüft nicht, ob menu_register eine Fehlermeldung zurückgegeben hat, also ob überhaupt noch Platz für ein weiteres Accessory war.

Omikron-BASIC

```
'
' Accessory-Demo       DEMOACC.BAS
' Omikron-BASIC       MP  23-12-88
'
Appl_Init
'
' Eintragen als Accessory:
'
Menu_Register("  Demo-Accessory",Ac_Id%L)
'
REPEAT
  Evnt_Mesag(Puffer$)'  Auf Nachricht warten
  '
  ' Unser Accessory gewünscht?
  '
  IF (FN P%L(0)=40) AND (FN P%L(4)=Ac_Id%L)
    THEN Go_Accessory
  ENDIF
```

```
  '
UNTIL 0'  Endlosschleife
'
'
DEF FN P%L(N%L)= CVI( MID$(Puffer$,N%L*2+1,2))
'
'
DEF PROC Go_Accessory
  '
  ' Eigentliches Accessory:
  '
  FORM_ALERT (1,"[1][S U P E R - A C C E S S O R Y| |Bitte wählen ...
             ... Sie:][Ende|Schluß|Abbruch]")
RETURN
```

C

```
/*************************************************************/
/*   Accessory - Header (muß als Accessory gelinkt werden)   */
/*   Megamax Laser C         MP  21-12-88        DEMOACC.C   */
/*************************************************************/

#include "gem_inex.c"

int  ac_id,
     puffer[8];


void go_accessory()
{
  form_alert (1,"[1][S U P E R - A C C E S S O R Y| |Bitte wählen ...
                   ... Sie:][Ende|Schluß|Abbruch]");
}


main()
{
  gem_init();

  /* gem_init liefert die Applikations-Nummer in der globalen */  /* Variablen
ap_id zurück, die im Include-File deklariert ist */

  ac_id = menu_register (ap_id, "  Demo-Accessory");

  /* Endlos-Warteschleife */

  while (1)
  {
    evnt_mesag (puffer);
```

```
    if (puffer[0] == 40)          /* Accessory gewünscht? */
      if (puffer[4] == ac_id)     /* unser Accessory? */
        go_accessory();           /* dann das Unterprogramm aufrufen */
  }
}
```

Assembler

```
;
; Demo-Accessory       DEMOACC.Q
; Assembler           MP 22-12-88
;

              INCLUDE 'ACC_INIT.Q'      ;neue Include-Datei

              TEXT

main:         jsr     gem_init     ;ap_id wird zurückgeliefert; menu_register:

              move.w  #35,control
              move.w  #1,control+2
              move.w  #1,control+4
              move.w  #1,control+6
              clr.w   control+8
              move.w  ap_id,int_in
              move.l  #acc_name,addr_in
              jsr     aes
              move.w  int_out,ac_id      ;ID des Menü-Eintrags

;evnt_mesag in einer Endlosschleife:

loop:         move.w  #23,control  ;Opcode evnt_mesag
              clr.w   control+2
              move.w  #1,control+4
              move.w  #1,control+6
              clr.w   control+8
              move.l  #puffer,addr_in
              jsr     aes

; erhaltene Nachricht überprüfen:

              cmpi.w  #40,puffer   ;Accessory aufgerufen?
              bne     loop

              move.w  puffer+8,d0  ;ID unseres Accessory
              cmp.w   ac_id,d0     ;sind wirklich wir dran?
              bne     loop

              jsr     go_accessory ;Programm aufrufen
```

```
            bra     loop        ;und Endlosschleife fortsetzen...

; Das folgende kleine Unterprogramm soll das eigentliche
; Accessory sein. Es gibt mit form_alert eine kleine
; Alert-Box auf dem Bildschirm aus.

go_accessory:

            move.w  #52,control  ;Opcode form_alert
            move.w  #1,control+2
            move.w  #1,control+4
            move.w  #1,control+6
            clr.w   control+8
            move.w  #1,int_in    ;Default-Button
            move.l  #form_str,addr_in
            jsr     aes

            rts                  ;Accessory - Ende


            DATA

form_str:   DC.b '[1][S U P E R - A C C E S S O R Y|'
            DC.b 'Bitte wählen Sie:]'
            DC.b '[Ende|Schluß|Abbruch]',0

acc_name:   DC.b '  Demo-Accessory',0

            BSS

puffer:     DS.w 8 ;Platz für Nachrichten

ac_id:      DS.w 1 ;Accessory-ID

            END
```

5.11 Beispielprogramme

Trotz der vielen Demoprogramme, die Sie bisher gesehen haben, ist es durchaus möglich, daß die ein oder andere Frage offen geblieben ist. Das liegt sicherlich nicht zuletzt daran, daß es in all den Programmen nicht um eine Anwendung, sondern um die jeweiligen Systemfunktionen an sich ging. Es ist also nicht verwunderlich, daß der sinnvolle Einsatz von Systemfunktionen zur Lösung eines Problems manchmal mehr Probleme bereitet als das Verständnis der einzelnen Betriebssystemfunktion.

Deshalb habe ich ein paar einfache Programme unter GEM erstellt, die nicht zur Demonstration der GEM-Funktionen geschrieben wurden, sondern wirklich zur Anwendung zu gebrauchen sind. Wenn Sie die Programme gründlich durcharbeiten (BASIC-Fans: Sehen Sie sich ruhig mal ein C-Listing an), dann werden Sie bestimmt noch eine ganze Menge von Tricks kennenlernen, die ich bis jetzt vielleicht nur deshalb nicht erwähnt habe, weil ich Sie für selbstverständlich halte.

Die Listings sind gut kommentiert, und auf eine umfangreiche Dokumentation möchte ich ganz bewußt verzichten; Sie sind mittlerweile so weit fortgeschritten, daß Sie alles Wichtige selbst herausfinden können, und das ist wesentlich sinnvoller als das Durchlesen der Programmdokumentation.

Ich hielt es jedoch nicht mehr für erforderlich, jedes Programm in allen vier Sprachen abzudrucken, so wie ich es bisher gemacht habe. Es ist aber für jede Sprache etwas dabei!

5.11.1 Accessory: Anzeige einer ASCII-Tabelle

Jedem der 256 Zeichen, die unser Atari ausgeben kann, ist eine Nummer (0 bis 255) zugeordnet. Ein Teil dieser Zeichen hat bei den meisten Computern die gleiche Nummer, weil die Zuordnung dieser Zeichen durch ASCII genormt ist. ASCII bedeutet American standard code for information interchange (amerikanischer Standard-Code für Informationsaustausch). Man sagt auch, daß jedem Zeichen ein ASCII-Code zugeordnet ist. Dieser ASCII-Code ist dann die Nummer, die das jeweilige Zeichen repräsentiert.Nun benötigt der Programmierer recht oft eine Tabelle mit den ASCII-Zeichen und ASCII-Codes; es gibt sie in unzähligen Varianten in Computer-Fachbüchern oder Fachzeitschriften. Solche Tabellen haben jedoch zwei Nachteile: Sie stören auf dem ohnehin meist überfüllten Schreibtisch, und nicht alle Tabellen enthalten auch die Erweiterungen des Atari-Zeichensatzes (mehr als die Hälfte der 256 Zeichen des Atari gehört nicht zum ASCII-Zeichensatz).

Da wir die Tabelle in der Regel direkt am Rechner brauchen werden, können wir jedoch ein kleines Accessory schreiben, das per Mausklick alle Zeichen unseres Computers zusammen mit ihren ASCII-Codes anzeigt. Die Codes sollten zweckmäßigerweise sowohl dezimal als auch hexadezimal ausgegeben werden; C-Programmierer können ja die Hex-Ausgabe durch eine oktale ersetzen.

Das Assembler-Programm benötigt beim Assemblieren die Include-Datei ACC_INIT.Q, die Sie im Ordner GEM der Diskette im Buch finden. Das Programm läuft nur auf dem Monochrom-Monitor, weil die große Informationsmenge selbst bei mittlerer Auflösung nicht mehr dargestellt werden könnte.

Die Tabelle sieht zwar, wenn sie fertig ist, wie eine Dialogbox aus, doch kommen wir ohne dieses Hilfsmittel (Objekte, ...) aus. Ein Resource-File würde die Ausgabe der Codes auch stark bremsen (es dauert auch so schon etwa zwei Sekunden, trotz Assembler-Programmierung). Statt dessen malen wir alles selbst, auch den Rahmen; auch das Löschen unseres Ausgaberechtecks nehmen wir höchstpersönlich vor. Nur zwei Funktionsaufrufe erinnern an normale Objekte: die form_dial-Unterfunktionen 0 und 3. Diese werden benötigt, damit das AES später den Bildschirm wiederherstellen kann. Wir müssen also nicht den gesamten Bildschirm retten, bevor wir uns austoben!

So arbeitet übrigens auch das VT-52-Emulator-Accessory: Wenn Sie dieses Hilfsprogramm beenden, dann wird das AES mit einem form_dial-Aufruf veranlaßt, den gesamten Arbeitsbereich wiederherzustellen. Das AES sendet dann auch Redraw-Meldungen an alle sichtbaren Fenster. Nur die Menüzeile kann damit nicht gerettet werden, doch die können Sie ohne Probleme mit den Raster-Funktionen sichern lassen.

```
;
; Accessory zum Anzeigen des Zeichensatzes (ASCII-Tabelle)
; Assembler              MP  05-01-89            ASCII.Q
;

gemdos      = 1
crawcin     = 7


            TEXT

            INCLUDE 'ACC_INIT.Q'

main:       jsr     gem_init

; menu_register:

            move.w  #35,control
            move.w  #1,control+2
            move.w  #1,control+4
            move.w  #1,control+6
            clr.w   control+8
            move.w  ap_id,int_in
```

```
                move.l  #acc_name,addr_in
                jsr     aes
                move.w  int_out,ac_id     ;Accessory-ID merken

; evnt_mesag-Endlos-Schleife:

mainloop:       move.w  #23,control  ;Opcode evnt_mesag
                clr.w   contrl+2
                move.w  #1,control+4
                move.w  #1,control+6
                clr.w   control+8
                move.l  #puffer,addr_in
                jsr     aes

; Prüfen der Nachricht: Waren wir gemeint?

                cmpi.w  #40,puffer   ;Erkennungszeichen: Accessory gerufen
                bne     mainloop

                move.w  puffer+8,d0  ;ac_id des gewünschten Programms
                cmp.w   ac_id,d0     ;Sind wir das?
                bne     mainloop

                jsr     accessory    ;Dann los!
                bra     mainloop


; ***** Jetzt folgt die eigentliche Routine: *****


accessory:

                move.w  #51,control  ;form_dial (AES)
                move.w  #9,control+2 ;AES soll Fensterränder retten
                move.w  #1,control+4
                clr.w   control+6
                clr.w   control+8
                clr.w   int_in       ;Unterfunktionsnummer 0
                move.w  #11,int_in+10     ;x,
                move.w  #31,int_in+12     ;y,
                move.w  #617,int_in+14    ;Breite (x2-x1+1) und
                move.w  #345,int_in+16    ;Höhe (y2-y1+1)
                jsr     aes


; Mauszeiger verstecken:

                move.w  #123,contrl  ;v_hide_c
                clr.w   contrl+2
                clr.w   contrl+4
```

```
            clr.w   contrl+6
            clr.w   contrl+8
            move.w  handle,contrl+12
            jsr     vdi

; Löschen des Ausgabebereichs:

            move.w  #23,contrl   ;vsf_interior
            clr.w   contrl+2     ;(setzt Fülltyp)
            clr.w   contrl+4
            move.w  #1,contrl+6
            move.w  #1,contrl+8
            clr.w   intin        ;Füllen mit Hintergrundfarbe
            jsr     vdi

            move.w  #104,contrl  ;vsf_perimeter
            clr.w   contrl+2
            clr.w   contrl+4
            move.w  #1,contrl+6
            move.w  #1,contrl+8
            move.w  handle,contrl+12
            clr.w   intin        ;Umrahmung beim Füllen aus
            jsr     vdi

            move.w  #11,contrl   ;v_bar
            move.w  #2,contrl+2
            clr.w   contrl+4
            clr.w   contrl+6
            clr.w   contrl+8
            move.w  #1,contrl+10 ;Unterfunktionsnummer
            move.w  handle,contrl+12
            move.w  #11,ptsin    ;Koordinaten
            move.w  #31,ptsin+2
            move.w  #627,ptsin+4
            move.w  #375,ptsin+6
            jsr     vdi


; Kästchen zeichnen:

            clr.w   d6
            clr.w   d7

r_loop:     move.w  #6,contrl    ;v_pline
            move.w  #2,contrl+2  ;2 Punkte verbinden
            clr.w   contrl+4
            clr.w   contrl+6
            clr.w   contrl+8

            move.w  #15,ptsin    ;x-Koordinate Punkt1
```

```
                move.w  #35,ptsin+2  ;y-Koordinate Punkt1
                add.w   d6,ptsin+2
                move.w  #623,ptsin+4 ;x-Koordinate Punkt2
                move.w  ptsin+2,ptsin+6   ;y-Koordinate Punkt2

                jsr     vdi

                move.w  #6,contrl    ;v_pline
                move.w  #2,contrl+2  ;2 Punkte verbinden
                clr.w   contrl+4
                clr.w   contrl+6
                clr.w   contrl+8

                move.w  #15,ptsin    ;x-Koordinate Punkt1
                add.w   d7,ptsin
                move.w  #35,ptsin+2  ;y-Koordinate Punkt1
                move.w  ptsin,ptsin+4     ;x-Koordinate Punkt2
                move.w  #371,ptsin+6

                jsr     vdi

                addi.w  #21,d6
                addi.w  #38,d7

                cmpi.w  #357,d6      ;Endwert
                bne     r_loop

; Rahmen wie bei Dialogbox zeichnen:

                move.w  #6,contrl    ;v_pline
                move.w  #5,contrl+2  ;5 Punkte verbinden
                clr.w   contrl+4
                clr.w   contrl+6
                clr.w   contrl+8

                move.w  #14,ptsin
                move.w  #34,ptsin+2
                move.w  #624,ptsin+4
                move.w  #34,ptsin+6
                move.w  #624,ptsin+8
                move.w  #372,ptsin+10
                move.w  #14,ptsin+12
                move.w  #372,ptsin+14
                move.w  #14,ptsin+16
                move.w  #34,ptsin+18

                jsr     vdi

                move.w  #6,contrl    ;v_pline
                move.w  #5,contrl+2  ;5 Punkte verbinden
```

```
            clr.w   contrl+4
            clr.w   contrl+6
            clr.w   contrl+8

            move.w  #11,ptsin
            move.w  #31,ptsin+2
            move.w  #627,ptsin+4
            move.w  #31,ptsin+6
            move.w  #627,ptsin+8
            move.w  #375,ptsin+10
            move.w  #11,ptsin+12
            move.w  #375,ptsin+14
            move.w  #11,ptsin+16
            move.w  #31,ptsin+18

            jsr     vdi


; Ausgabe der Tabelle: Zeichen + ASCII-Code (dezimal und hexadezimal)

            move.b  #'$',hex      ;Dollarzeichen in HEX-String

            clr.w   d5            ;nächstes auszugebende Zeichen
            clr.w   d7            ;Zeile
loop1:      clr.w   d6            ;Spalte

; Ausgabe des Zeichens:

loop2:      lea     digits,a0     ;ASCII-Codes in Ziffern schreiben

            move.w  d5,d4         ;Hexzahl erzeugen
            andi.w  #%1111,d4
            move.b  0(a0,d4.w),hex+2  ;Einerziffer schreiben
            move.w  d5,d4
            asr.w   #4,d4
            move.b  0(a0,d4.w),hex+1  ;16er-Ziffer

            move.w  d5,d4         ;Dezimalzahl erzeugen
            andi.l  #$ffff,d4     ;oberes Wort löschen
            divu    #10,d4        ;/ Basis des Zahlensystems
            swap    d4            ;Rest in unteres Wort
            move.b  0(a0,d4.w),dez+2  ;Einerziffer
            swap    d4            ;Ergebnis weiterverarbeiten
            andi.l  #$ffff,d4     ;oberes Wort löschen
            divu    #10,d4
            move.b  0(a0,d4.w),dez    ;Ergebnis ist 100er-Ziffer
            swap    d4
            move.b  0(a0,d4.w),dez+1  ;Rest ist Zehnerziffer
            cmpi.b  #'0',dez      ;führende Nullen löschen
            bne.s   l
```

```
            move.b  #' ',dez      ;Leerzeichen
            cmpi.b  #'0',dez+1
            bne.s   l
            move.b  #' ',dez+1


l:          move.w  #107,contrl   ;8x16 Schriftgröße wählen
            clr.w   contrl+2      ;vst_point
            move.w  #2,contrl+4
            move.w  #1,contrl+6
            move.w  #1,contrl+8
            move.w  handle,contrl+12
            move.w  #13,intin     ;Größe 13
            jsr     vdi

            move.w  #8,contrl     ;v_gtext
            move.w  #1,contrl+2
            clr.w   contrl+4
            move.w  #1,contrl+6   ;1 Zeichen ausgeben
            clr.w   contrl+8
            move.w  handle,contrl+12
            move.w  d5,intin      ;Zeichen
            move.w  #43,ptsin     ;x-Koordinate
            add.w   d6,ptsin
            move.w  #52,ptsin+2   ;y-Koordinate
            add.w   d7,ptsin+2
            jsr     vdi

            move.w  #107,contrl   ;6x6 Schriftgröße wählen
            clr.w   contrl+2      ;vst_point
            move.w  #2,contrl+4
            move.w  #1,contrl+6
            move.w  #1,contrl+8
            move.w  handle,contrl+12
            move.w  #4,intin      ;Größe 13
            jsr     vdi

            move.w  #8,contrl     ;v_gtext (Ausgabe Dezimal)
            move.w  #1,contrl+2
            clr.w   contrl+4
            move.w  #3,contrl+6   ;3 Zeichen ausgeben
            clr.w   contrl+8
            move.w  handle,contrl+12

            clr.b   intin         ;Zeichen
            move.b  dez,intin+1
            clr.b   intin+2
            move.b  dez+1,intin+3
            clr.b   intin+4
```

```
                move.b  dez+2,intin+5

                move.w  #19,ptsin       ;x-Koordinate
                add.w   d6,ptsin
                move.w  #42,ptsin+2     ;y-Koordinate
                add.w   d7,ptsin+2
                jsr     vdi

                move.w  #8,contrl       ;v_gtext (Ausgabe Hexadezimal)
                move.w  #1,contrl+2
                clr.w   contrl+4
                move.w  #3,contrl+6     ;3 Zeichen ausgeben
                clr.w   contrl+8
                move.w  handle,contrl+12

                clr.b   intin           ;Zeichen
                move.b  hex,intin+1
                clr.b   intin+2
                move.b  hex+1,intin+3
                clr.b   intin+4
                move.b  hex+2,intin+5

                move.w  #19,ptsin       ;x-Koordinate
                add.w   d6,ptsin
                move.w  #53,ptsin+2     ;y-Koordinate
                add.w   d7,ptsin+2
                jsr     vdi

                addq.w  #1,d5           ;nächstes Zeichen

                addi.w  #38,d6
                cmpi.w  #608,d6         ;Zeile fertig?
                bne     loop2

                addi.w  #21,d7          ;ja, dann nächste Zeile
                cmpi.w  #336,d7
                bne     loop1


; Mauszeiger wieder einschalten:

                move.w  #122,contrl
                clr.w   contrl+2
                clr.w   contrl+4
                move.w  #1,contrl+6
                clr.w   contrl+8
                move.w  handle,contrl+12
                move.w  #1,intin
                jsr     vdi
```

```
; Warten auf Tastendruck:

; Bemerkung:
;   Eigentlich wollte ich evnt_multi verwenden und auf Tastendruck
;   oder Mausklick warten. Das geht jedoch nicht, weil uns GEM
;   leider nur dann solche Ereignisse mitteilt, wenn wir das
;   aktive Bildschirmfenster haben. Da unser Programm aber über-
;   haupt keine Fenster hat, haben wir natürlich auch nicht das
;   aktive Fenster. Deshalb folgt nun ein einfacher GEMDOS-Aufruf:

            move.w  #crawcin,-(sp)
            trap    #gemdos
            addq.l  #2,sp

; Das Wiederherstellen des Bildschirms überlassen wir GEM:

            move.w  #51,control  ;form_dial
            move.w  #9,control+2 ;AES soll Redraw-Meldungen geben
            move.w  #1,control+4
            clr.w   control+6
            clr.w   control+8
            move.w  #3,int_in    ;Unterfunktionsnummer
            move.w  #11,int_in+10     ;x,
            move.w  #31,int_in+12     ;y,
            move.w  #617,int_in+14    ;Breite (x2-x1+1) und
            move.w  #345,int_in+16    ;Höhe (y2-y1+1)
            jsr     aes

            rts                  ;Ende Accessory


            DATA

digits:     DC.b '0123456789ABCDEF'
acc_name:   DC.b '  ASCII-Codes',0

            BSS

dez:        DS.b 3 ;Hier werden die Zahlen für die Ausgabe
hex:        DS.b 3 ;aufbereitet

            ALIGN

puffer:     DS.w 8
ac_id:      DS.w 1 ;Accessory-ID

            END
```

5.11.2 Applikation: Quelltext-Lister

Dieses Omrikon-BASIC-Programm wird schnell seine Freunde finden. Es bietet Ihnen die Möglichkeit, Programmlistings oder andere ASCII-Texte sehr komfortabel auszudrucken. Mit verschiedenen Dialogboxen können Sie folgende Parameter bestimmen:

- Ausdruck seitenweise (mit beliebigem Titel im Kopf) oder durchgehend. Entscheiden Sie sich für den seitenweisen Ausdruck, so druckt der Rechner im Kopf jeder Seite auch die Seitennummer, den Dateinamen sowie Datum und Uhrzeit aus.

- Druck mit oder ohne Zeilennummern.

- Steuercodes Ihres Druckers für:

  a) Seitenvorschub (Form Feed)
  b) Wagenrücklauf+Zeilenvorschub
  c) Init-String (z.B.: Umschalten auf Schmalschrift)
  d) Exit-String (z.B.: Drucker-Reset, Glocke etc.)

- Seitenformat:

  a) Zeilen pro Seite (einschließlich Kopf)
  b) Zeichen pro Zeile (Zeilenbreite) einschließlich Rand
  c) Breite des linken Randes
  d) Ausgabe ein- oder zweispaltig (nur bei seitenweisem Ausdruck).Die letzte Option (Ausdruck in zwei Spalten) ist eine Rarität; Sie können damit bei nicht zu breiten Texten oder auf einem A3-Drucker die Hälfte des Papiers sparen!

Die Dialogboxen und Ihren Aufbau möchte ich nicht detailliert erklären; statt dessen habe ich zusätzlich zum RSC-File auch die Definitionsdateien für das Atari-RCS2 und für das Megamax RCP auf die Diskette kopiert. Wenn es also zu Unklarheiten im Zusammenhang mit den Objekten oder deren Namen kommt, dann schauen Sie sich bitte das RSC-File mit einem RCS an.

Sie werden feststellen, daß die eigentliche Druckroutine gar nicht so lang ist. Das Drumherum nimmt jedoch ziemlich viel Platz in Anspruch, was typisches Erkennungsmerkmal einer GEM-Applikation ist. Lassen Sie sich nicht davon abschrecken; es werden keine GEM-Routinen benutzt, die ich nicht schon in kleineren Beispielprogrammen vorgestellt habe.

Dies ist übrigens das einzige Omrikon-Programm in diesem Buch, das Zeilennummern besitzt. Diese wurden durch die ON-ERROR-GOTO-Anweisung beim Laden eines Textes nötig, weil die EOF-Funktion des BASIC bei nicht-Omrikon-Dateien nur unzuverlässig arbeitete. In diesem Zusammenhang muß auch erwähnt werden, daß das Programm (wegen des ON-ERROR-GOTO-Befehls) compiliert nur dann richtig arbeitet, wenn in der Laderoutine die Compiler-Anweisung Trace_On steht (siehe Programm). Letzter Hinweis: In der Druck-Routine wird die Ausgabe des LPRINT-Befehls durch die Anweisung MODE LPRINT "D" dahingehend beeinflußt, daß auf einem EPSON-kompatiblen Drucker auch die deutschen Umlaute korrekt ausgedruckt werden. Wenn Sie einen Drucker mit IBM-Zeichensatz besitzen, dann sollten Sie diesen MODE-Befehl streichen. Einen Hinweis dazu finden Sie im Listing.

```
100 '
101 ' GEM-Applikation: Datei-Lister (Drucker-Ausgabe)
102 ' Omikron-BASIC      MP  05-01-89     LISTER.BAS
103 '
104 CLEAR 33000'   Speicher muß für RSRC-LOAD zurückgegeben werden
105 '
106 ' Konstanten aus dem Resource-Header-File:
107 '
108 Starten=18
109 Fine=20
110 Format=23
111 Steuer=22
112 Speicher=25
113 B_Steuer=4
114 Sv=1
115 Init=4
116 Crlf=3
117 Ex=5
118 Okk=6
119 Abbr=7
120 B_Form=3
121 B_Info=2
122 Eins=2
123 Zwei=3
124 F_Zeilen=6
125 Rand=8
126 F_Zeich=7
127 Dname=1
128 B_Druck=1
129 B_Menue=0
130 Kopf=2
131 Seiten=5
132 Znr=4
133 Abb=7
```

```
134 Ok=8
135 Info=8
136 Laden=17
137 '
138 DIM Zeilen$(3000)'  reicht wohl für längere Listings...
139 '
140 Datei_Geladen=0'    Flag initialisieren
141 '
142 Puffer$= SPACE$(16)' AES-Ereignis-Puffer (8 Worte)
143 '
144 ' Funktion zum Zugriff auf den Puffer:
145 '
146 DEF FN P(X)= CVI( MID$(Puffer$,X*2+1,2))
147 '
148 Appl_Init
149 V_Opnvwk
150 '
151 Lade_Parameterdatei
152 '
153 Rsrc_Load("LISTER.RSC",Back)'  RSC-File laden
154 IF Back=0 THEN
155   FORM_ALERT (1,"[3][Kein RSC-File!][Abbruch]")
156   Appl_Exit
157   END
158 ENDIF
159 '
160 ' Adressen der Objekt-Bäume holen:
161 '
162 Rsrc_Gaddr(0,B_Menue,Menue_Ptr)
163 Rsrc_Gaddr(0,B_Info,Info_Ptr)
164 Rsrc_Gaddr(0,B_Druck,Druck_Ptr)
165 Rsrc_Gaddr(0,B_Form,Form_Ptr)
166 Rsrc_Gaddr(0,B_Steuer,Steuer_Ptr)
167 '
168 ' Menüleiste anzeigen:
169 '
170 Menu_Bar(Menue_Ptr)
171 '
172 Abbruch=0
173 '
174 ' Ereignis-Warteschleife:
175 '
176 REPEAT
177   Evnt_Mesag(Puffer$)'  Warten auf AES-Nachricht
178   '
179   IF FN P(0)=10 THEN '   Wort 10=0  --> Menüpunkt angeklickt
180     Title=FN P(3)'       Titel-Index merken
181     '
182     IF FN P(4)=Info THEN Sub_Info
183     IF FN P(4)=Laden THEN Sub_Laden
```

```
184     IF FN P(4)=Starten THEN Sub_Starten
185     IF FN P(4)=Fine THEN Sub_Fine
186     IF FN P(4)=Steuer THEN Sub_Steuer
187     IF FN P(4)=Format THEN Sub_Format
188     IF FN P(4)=Speicher THEN Sub_Speicher
189     '
190     Menu_Tnormal(Title,1)'   Menütitel normal zeichnen
191   ENDIF
192   '
193 UNTIL Abbruch=1'   Zurück, wenn nicht Ende gewünscht
194 '
195 Rsrc_Free
196 V_Clsvwk
197 Appl_Exit
198 END
199 '
200 '
201 DEF PROC Sub_Fine
202   FORM_ALERT (1,"[2][Programm bestimmt verlassen?][Ja|Nein]",Back)
203   IF Back=1 THEN Abbruch=1'  Flag für Hauptprogramm setzen204 RETURN
205 '
206 '*****************************************************************
207 '
208 DEF PROC Sub_Laden
209   Trace_On'    Ermöglicht im Compiler ON ERROR GOTO...
210   '
211   ' Anzeigen einer File-Selektor-Box:
212   '
213   Path$= STRING$(64,0)'    Platz machen
214   Adr= LPEEK( VARPTR(Path$))+ LPEEK( SEGPTR +28)
215   GEMDOS (,$47, HIGH(Adr), LOW(Adr),0)'  Pfadname holen
216   Path$= LEFT$(Path$, INSTR(Path$, CHR$(0))-1)' Nullbytes weg
217   '
218   GEMDOS (Akt_Drive,$19)'  aktuelles Laufwerk ermitteln
219   Path$= CHR$( ASC("A")+Akt_Drive)+":"+Path$+"\*.*"
220   '
221   Default$=""
222   FILESELECT (Path$,Default$,Flag)
223   '
224   Dateiname$= LEFT$(Path$, LEN(Path$)- INSTR( MIRROR$(Path$)+...
                        ..."\","\"))+"\"+Default$
225   '
226   IF NOT (Flag=0 OR Default$="") THEN
227     ON ERROR GOTO 243'  EOF bereitet gelegentlich Schwierigkeiten...
228     OPEN "I",1,Dateiname$'   Open for Input
229     I=-1
230     WHILE NOT EOF(1)
231       I=I+1
232       LINE INPUT #1,Zeilen$(I)
233     WEND
```

```
234     Letzte_Zeile=I
235     FOR I=Letzte_Zeile+1 TO 3000:Zeilen$(I)="": NEXT I
236     IF I>0 THEN Datei_Geladen=1:Write_Text(Druck_Ptr,Kopf,"")
237     CLOSE 1
238   ENDIF
239   '
240   Trace_Off'      Brauchen wir jetzt nicht mehr
241 RETURN
242 '
243 I=I-1: RESUME NEXT '   Beim Fehler wird EOF(1) true.
244 '
245 '
246 DEF PROC Trace_On: RETURN '    Dummy-Prozeduren, damit das
247 DEF PROC Trace_Off: RETURN '   Programm im Interpreter läuft
248 '
249 '
250 '*****************************************************************
251 '
252 DEF PROC Sub_Starten
253   IF Datei_Geladen=0 THEN
254     FORM_ALERT (1,"[1][Sie haben noch keine|Datei geladen!][Ach?]")
255     RETURN
256   ENDIF
257   '
258   Write_Text(Druck_Ptr,Dname,Dateiname$)
259   IF Seitenweise=1 THEN Select(Druck_Ptr,Seiten) ELSE...
                    ... Deselect(Druck_Ptr,Seiten)
260   IF Zeilennummern=1 THEN Select(Druck_Ptr,Znr) ELSE...
                    ... Deselect(Druck_Ptr,Znr)
261   Do_Dialog(Druck_Ptr,Kopf,Back)
262   '
263   Read_Text(Druck_Ptr,Kopf,Kopf$)
264   '
265   IF Back=Ok THEN
266     Seitenweise=0:Zeilennummern=0
267     IF FN Selected(Druck_Ptr,Seiten) THEN Seitenweise=1
268     IF FN Selected(Druck_Ptr,Znr) THEN Zeilennummern=1
269     Drucken
270   ENDIF
271   '
272 RETURN
273 '
274 '*****************************************************************
275 '
276 DEF PROC Sub_Info
277   Do_Dialog(Info_Ptr,0,Dummy)
278 RETURN
279 '
280 '*****************************************************************
281 '
```

```
282 DEF PROC Sub_Steuer
283   USING "####"'   Gilt auch für STR$(..)
284   A$=FN St_Str$(Sv1,Sv2,Sv3,Sv4)
285   Write_Text(Steuer_Ptr,Sv,A$)
286   A$=FN St_Str$(Crlf1,Crlf2,Crlf3,Crlf4)
287   Write_Text(Steuer_Ptr,Crlf,A$)
288   A$=FN St_Str$(Init1,Init2,Init3,Init4)
289   Write_Text(Steuer_Ptr,Init,A$)
290   A$=FN St_Str$(Ex1,Ex2,Ex3,Ex4)
291   Write_Text(Steuer_Ptr,Ex,A$)
292   '
293   Do_Dialog(Steuer_Ptr,Sv,Back)
294   '
295   IF Back=Okk THEN
296     Read_Text(Steuer_Ptr,Sv,A$)
297     Lies(A$,Sv1,Sv2,Sv3,Sv4)
298     Read_Text(Steuer_Ptr,Crlf,A$)
299     Lies(A$,Crlf1,Crlf2,Crlf3,Crlf4)
300     Read_Text(Steuer_Ptr,Init,A$)
301     Lies(A$,Init1,Init2,Init3,Init4)
302     Read_Text(Steuer_Ptr,Ex,A$)
303     Lies(A$,Ex1,Ex2,Ex3,Ex4)
304   ENDIF
305 RETURN
306 '
307 DEF FN St_Str$(X1,X2,X3,X4)
308   LOCAL A$:A$=""
309   IF X1>=0 THEN A$=A$+ RIGHT$( STR$(X1),3) ELSE RETURN A$
310   IF X2>=0 THEN A$=A$+ RIGHT$( STR$(X2),3) ELSE RETURN A$
311   IF X3>=0 THEN A$=A$+ RIGHT$( STR$(X3),3) ELSE RETURN A$
312   IF X4>=0 THEN A$=A$+ RIGHT$( STR$(X4),3)
313 RETURN A$
314 '
315 DEF PROC Lies(A$,R A,R B,R C,R D)
316   A=-1:B=-1:C=-1:D=-1
317   IF MID$(A$,1,3)<>"" THEN A= VAL( MID$(A$,1,3))
318   IF MID$(A$,4,3)<>"" THEN B= VAL( MID$(A$,4,3))
319   IF MID$(A$,7,3)<>"" THEN C= VAL( MID$(A$,7,3))
320   IF MID$(A$,10,3)<>"" THEN D= VAL( MID$(A$,10,3))
321 RETURN
322 '
323 '*****************************************************************
324 '
325 DEF PROC Sub_Format
326   USING "####"
327   Write_Text(Form_Ptr,F_Zeilen, RIGHT$( STR$(Zeilen_Seite),2))
328   Write_Text(Form_Ptr,F_Zeich, RIGHT$( STR$(Zeichen_Zeile),3))
329   Write_Text(Form_Ptr,Rand, RIGHT$( STR$(Randbreite),2))
330   '
331   Deselect(Form_Ptr,Eins)'  Spaltenzahl
```

```
332   Deselect(Form_Ptr,Zwei)'  (Radio-Buttons)
333   IF Spalten=1 THEN Select(Form_Ptr,Eins) ELSE Select(Form_Ptr,Zwei)
334   '
335   Do_Dialog(Form_Ptr,F_Zeilen,Back)
336   '
337   IF FN Selected(Form_Ptr,Eins) THEN Spalten=1 ELSE Spalten=2338   '
339   Read_Text(Form_Ptr,F_Zeilen,A$)
340   Zeilen_Seite= VAL(A$)
341   Read_Text(Form_Ptr,F_Zeich,A$)
342   Zeichen_Zeile= VAL(A$)
343   Read_Text(Form_Ptr,Rand,A$)
344   Randbreite= VAL(A$)
345 RETURN
346 '
347 '*****************************************************************
348 '
349 DEF PROC Lade_Parameterdatei
350   OPEN "I",1,"LISTER.PAR"
351   INPUT #1,Sv1,Sv2,Sv3,Sv4
352   INPUT #1,Crlf1,Crlf2,Crlf3,Crlf4
353   INPUT #1,Init1,Init2,Init3,Init4
354   INPUT #1,Ex1,Ex2,Ex3,Ex4
355   INPUT #1,Zeilen_Seite
356   INPUT #1,Zeichen_Zeile
357   INPUT #1,Randbreite
358   INPUT #1,Spalten
359   INPUT #1,Seitenweise
360   INPUT #1,Zeilennummern
361   CLOSE 1
362 RETURN
363 '
364 '
365 DEF PROC Sub_Speicher
366   OPEN "O",1,"LISTER.PAR"
367   WRITE #1,Sv1,Sv2,Sv3,Sv4
368   WRITE #1,Crlf1,Crlf2,Crlf3,Crlf4
369   WRITE #1,Init1,Init2,Init3,Init4
370   WRITE #1,Ex1,Ex2,Ex3,Ex4
371   WRITE #1,Zeilen_Seite
372   WRITE #1,Zeichen_Zeile
373   WRITE #1,Randbreite
374   WRITE #1,Spalten
375   WRITE #1,Seitenweise
376   WRITE #1,Zeilennummern
377   CLOSE 1
378 RETURN
379 '
380 '*****************************************************************
381 '
382 DEF PROC Drucken
```

```
383  WHILE 1=1'       Endlosschleife
384    BIOS (Back,8,0)'   Bcostat (Nr. 8):  Ist Drucker (0) bereit?
385    IF Back<0 THEN EXIT '   Doch nicht endlos...
386    FORM_ALERT (1,"[3][Bitte Drucker anschalten|und auf ONLINE...
                 ... stellen!][Wird gemacht, Chef!]")
387  WEND
388  '
389  ' Der folgende Befehl dient dazu, die deutschen Umlaute auf
390  ' EPSON-kompatiblen Druckern korrekt auszugeben. Wenn Sie einen
391  ' Atari oder IBM Drucker verwenden, so müssen Sie diesen Befehl
392  ' löschen:
393  '
394  MODE LPRINT "D"'    Man druckt deutsch
395  '
396  ' Weil diese automatische Umwandlung die Übertragung von Steuer-
397  ' codes gefährden könnte, öffnen wir zusätzlich einen direkten
398  ' Printer-Kanal:
399  '
400  OPEN "P",2
401  '
402  ' Init-String senden:
403  '
404  PRINT #2,FN D_Code$(Init1,Init2,Init3,Init4);
405  '
406  Rand$= SPACE$(Randbreite)
407  Zs=Zeilen_Seite-4'   4 Zeilen für Kopfzeile
408  Akt_Breite=Zeichen_Zeile-Rand
409  Spalten_Breite=Akt_Breite
410  IF Spalten=2 THEN Spalten_Breite=(Akt_Breite)\2
411  Crlf$=FN D_Code$(Crlf1,Crlf2,Crlf3,Crlf4)'  neue Zeile
412  Sv$=FN D_Code$(Sv1,Sv2,Sv3,Sv4)'      Seitenvorschub
413  '
414  FOR I=0 TO Letzte_Zeile'    Hauptschleife für Ausdruck
415    '
416    IF (I MOD Zs=0) AND (Spalten=2) AND (I>0) THEN I=I+Zs
417    IF I>Letzte_Zeile THEN EXIT
418    '
419    IF (I MOD (Zs*Spalten)=0) AND (Seitenweise=1) THEN '  Kopfzeile
420      IF I>0 THEN PRINT #2,Sv$'   Seitenvorschub ab Seite 2
421      USING "":Sn$="-"+ STR$(I\(Zs*Spalten)+1)+" -"
422      MODE "D"'   für Datum
423      A$= SPACE$((Akt_Breite- LEN(Kopf$))\2- LEN(Sn$))+Kopf$
424      LPRINT Rand$;Sn$;A$;: PRINT #2,Crlf$;
425      A$= DATE$ +", "+ TIME$ + SPACE$(Akt_Breite-18-...
                        ... LEN(Dateiname$))
426      A$=A$+Dateiname$: LPRINT Rand$;A$;: PRINT #2,Crlf$;
427      A$= STRING$(Akt_Breite,"-"): LPRINT Rand$;A$;: PRINT #2,Crlf$;
428      PRINT #2,Crlf$;'    Leerzeile
429    ENDIF
430    '
```

```
431     A$="": USING "#####"
432     IF Zeilennummern=1 THEN A$=A$+ STR$(I+1)+": "
433     A$=A$+Zeilen$(I)
434     '
435     IF (Spalten=2) AND (Seitenweise=1) AND (I+Zs)<=Letzte_Zeile THEN
436       IF LEN(A$)<Spalten_Breite THEN A$=A$+ SPACE$...
                    ...(Spalten_Breite- LEN(A$))
437       IF Zeilennummern=1 THEN A$=A$+ STR$(I+Zs+1)+": "
438       A$=A$+Zeilen$(I+Zs)
439     ENDIF
440     '
441     LPRINT Rand$;A$;: PRINT #2,Crlf$;
442     '
443   NEXT I
444   '
445   ' Exit-String senden:
446   '
447   PRINT #2,FN D_Code$(Ex1,Ex2,Ex3,Ex4);
448   '
449   CLOSE 2'     Druckerkanal schließen
450 RETURN
451 '
452 '
453 DEF FN D_Code$(A,B,C,D)
454   LOCAL A$
455   IF A>=0 THEN A$= CHR$(A) ELSE RETURN ""
456   IF B>=0 THEN A$=A$+ CHR$(B) ELSE RETURN A$
457   IF C>=0 THEN A$=A$+ CHR$(C) ELSE RETURN A$
458   IF D>=0 THEN A$=A$+ CHR$(D)
459 RETURN A$
460 '
461 '*****************************************************************
462 '
463 DEF PROC Select(Baum,Index)
464   WPOKE Baum+24*Index+10, WPEEK(Baum+24*Index+10) OR 1
465 RETURN
466 '
467 DEF PROC Deselect(Baum,Index)
468   WPOKE Baum+24*Index+10, WPEEK(Baum+24*Index+10) AND -2
469 RETURN
470 '
471 DEF FN Selected(Baum,Index)= WPEEK(Baum+24*Index+10) AND 1
472 '
473 DEF PROC Write_Text(Baum,Index,Text$)
474   LOCAL Adr,I,A$
475   Adr= LPEEK( LPEEK(Baum+24*Index+12))
476   A$=Text$+ CHR$(0)
477   FOR I=1 TO LEN(A$)
478     POKE Adr+I-1, ASC( MID$(A$,I,1))
479   NEXT I
```

```
480 RETURN
481 '
482 DEF PROC Read_Text(Baum,Index,R Text$)
483   LOCAL Adr,I,A$
484   Adr= LPEEK( LPEEK(Baum+24*Index+12))
485   Text$="":I=0
486   WHILE PEEK(Adr+I)<>0
487     Text$=Text$+ CHR$( PEEK(Adr+I))
488     I=I+1
489   WEND
490 RETURN
491 '
492 DEF PROC Do_Dialog(Baum,Start_Ed,R Ex_Button)
493   LOCAL X,Y,W,H
494   Form_Center(Baum,X,Y,W,H)
495   Form_Dial(0,X,Y,W,H)
496   Form_Dial(1,X,Y,W,H)
497   Objc_Draw(0,12,X,Y,W,H,Baum)
498   Form_Do(Start_Ed,Baum,Ex_Button)
499   Deselect(Baum,Ex_Button)
500   Form_Dial(2,X,Y,W,H)
501   Form_Dial(3,X,Y,W,H)
502 RETURN
503 '
```

5.11.3 Applikation: Balken-/Tortendiagramme

Im Gegensatz zum Computer kann der Mensch Zahlen viel leichter erfassen und miteinander vergleichen, wenn sie grafisch in Diagrammen dargestellt werden. Je nach Art der Zahlen bieten sich dafür verschiedene Diagramm-Typen an: Balkendiagramme, Liniendiagramme oder Tortendiagramme, um nur einige zu nennen.

Mein Beispielprogramm kann als Grundlage für ein solches Diagramm-Programm dienen. Es kann aus maximal acht Zahlenwerten Balken- und Tortendiagramme zeichnen. Zusätzlich kann der Benutzer nachträglich Text zur Beschriftung und Linien (z.B. zur Skalierung) anbringen. Die Bedienung erfolgt über Menüs und Dialogboxen, während die Ausgabe in einem Window stattfindet. Größe und Position des Fensters sind konstant, und es gibt nur dieses eine Fenster. Da allerdings der Aufruf von Accessories gestattet ist, wurde ein vollständiges Redraw-Verfahren eingebaut.

Wieder einmal ist es so, daß das Programm durch die GEM-Umgebung "etwas" länger geworden ist. Sie werden allerdings bemerken, daß die Arbeit mit Dialogboxen zur Daten-Eingabe teilweise zu langweiligen Passagen im Listing führt. Dadurch, daß man Konstanten wie s1, s2, s3... nicht über einen Index anspechen kann, entstehen im Listing oft Folgen von fast identischen Anweisungen. Das ist der Preis, mit dem der Programmierer dem Anwender eine komfortable Bedienung erkauft.

Auf ein paar Punkte soll jetzt noch genauer eingegangen werden: Da ist zunächst die globale Variable kein_redraw, die vielleicht Verwirrung stiften könnte. Wie Sie der Prozedur redraw entnehmen können, können Sie durch Setzen der Variablen kein_redraw einmal eine Redraw-Nachricht vom AES ignorieren lassen. Erst die nächste Redraw-Meldung wird dann wieder wie gewohnt ausgeführt.

Nehmen wir doch ein Beispiel, wie es in diesem Programm oft vorkommt: Ein Diagramm befindet sich auf dem Bildschirm, während der Anwender in einer Dialogbox die Daten für dieses Diagramm editiert. Bestätigt er die Änderung durch den OK-Button, dann zeichnet das Programm grundsätzlich das ganze Diagramm neu (also nicht nur den Teil, der durch die Box verdeckt war). Allgemein ausgedrückt: Eine Änderung irgendwelcher Daten hat ein Neuzeichnen des gesamten Fensters erforderlich gemacht.

Allerdings weiß GEM natürlich nicht, daß wir das Fenster komplett erneuert haben. Folglich erhalten wir beim nächsten evnt_mesag-Aufruf eine Redraw-Nachricht: Die Dialogbox wurde ja gelöscht, und wir möchten doch bitte den Hintergrund wiederherstellen. Da aber ohnehin das gesamte Fenster gerade erst restauriert wurde, ist natürlich auch der Bereich der Dialogbox in ordentlichem Zustand. Also ist die Redraw-Meldung überflüssig und wird von uns nicht beachtet. Mit dem gleichen Trick arbeitet das Programm auch an einigen anderen Stellen.

Punkt 2: Grafik abspeichern. Dieser Menüpunkt erlaubt es, ein Diagramm zur Weiterverarbeitung mit einem Grafikprogramm im Doodle-(Screen-)Format abzuspeichern. Dazu ist es nur erforderlich, mit BSAVE den aktuellen Bildschirmspeicher (Adresse erfahren wir durch XBIOS(3), Logbase; Länge: stets 32000 Bytes) zu sichern. Allerdings hätten wir dann die Menüzeile und den Fensterrand mit auf der Datei. Um das zu unterbinden, legen wir mit MALLOC einen Speicherbereich im RAM an, der komplett gelöscht wird. Dann kopieren wir mit der Anweisung RC_COPY nur den Inhalt des Fensters vom Original-Bildschirm in den

zweiten logischen Bildschirm (siehe auch Kapitel 2.1). Diesen Bereich (seine Startadresse heißt im Listing screen2) speichern wir mit BSAVE ab.

Der dritte Punkt betrifft noch einmal das Redraw-Verfahren. Da das Programm den Bildschirm nicht rettet (würde viel Speicherplatz kosten), sondern das Diagramm bei jedem Redraw neu berechnet und gezeichnet wird, fehlen nach jedem Neuzeichnen alle bisherigen Text-Ausgaben und Linien, die der Benutzer nachträglich in das Bild geschrieben hat. Um das zu verhinden, habe ich den ersten Schritt zum objekt-orientierten Grafik-Programm getan (aber auch wirklich nur den ersten): Jedesmal, wenn Text ausgegeben oder eine Linie gezogen wird, dann wird diese Aktion in einer Liste in Strings codiert festgehalten. Diese Liste wird über die Variablen opts$() und next_opt angesprochen. Bei jedem Neuzeichnen des Bildschirms oder eines Teils davon wird also nicht nur das Diagramm gezeichnet; es werden auch alle Grafik-Befehle, die in der Liste stehen, ausgeführt. Das erste Byte eines Befehls bestimmt übrigens, ob die Ausgabe im Modus Balkengrafik oder Tortengrafik erfolgen soll. Die Liste wird nur dann gelöscht, wenn Sie eine neue Datei zur Bearbeitung laden.

```
'
' Zahlen grafisch in Balken/Tortendiagramm darstellen
' GFA-BASIC          MP  06-01-88        DIAGRAMM.GFA
'
RESERVE -33000     ! etwas Speicher zurückgeben an GEMDOS
'
DEFINT "a-z"       ! Ab sofort: Alle Variablen sind Integer
'
' Konstanten des Resource-Files:
'
b_daten=2
s1=4
w1=5
w2=8
s2=7
s3=10
w3=11
s4=13
w4=14
s5=16
w5=17
w6=20
s6=19
s7=22
w7=23
s8=25
```

```
w8=26
ok=28
abbruch=27
b_menue=0
info=8
laden=17
sich=18
gsich=20
ende=22
dedit=24
balken=26
torten=27
schrift=29
linien=30
b_info=1
'
' Eigene Konstanten:
'
akt_form=balken  ! 'balken' und 'torten' sind oben definiert
'
DIM bez$(8),wert$(8),wert#(8)   ! Daten einer Grafik
FOR i=1 TO 8
  bez$(i)=""
  wert$(i)=""
  wert#(i)=0
NEXT i
'
DIM opts$(100)      ! Zusätzliche Grafik wird hier für Redraw gesichert
next_opt=0
'
' Der File-Selector braucht nacher einen Pfadnamen:
'
path$=CHR$(ASC("A")+GEMDOS(&H19))+":"+DIR$(0)+"\*.DIF"
path2$=CHR$(ASC("A")+GEMDOS(&H19))+":"+DIR$(0)+"\*.DOO"
default$=""
default2$=""
'
VOID APPL_INIT()
'
IF RSRC_LOAD("DIAGRAMM.RSC")=0
  VOID FORM_ALERT(1,"[3][Kein RSC-File!][Ende]")
  VOID APPL_EXIT()
  END
ENDIF
'
' Startadressen der Bäume holen:
'
VOID RSRC_GADDR(0,b_menue,menue_ptr)
VOID RSRC_GADDR(0,b_info,info_ptr)
VOID RSRC_GADDR(0,b_daten,daten_ptr)
```

```
'
' Arbeitsbereich ermitteln, graue Fläche zeichnen:
'
VOID WIND_GET(0,4,xdesk,ydesk,wdesk,hdesk)
'
DEFFILL 1,2,4          ! Desktop-Grau
BOUNDARY 0             ! Umrahmung aus
PBOX xdesk,ydesk,xdesk+wdesk-1,ydesk+hdesk-1
'
' Menüpunkt Linien und Beschriften sind noch nicht erlaubt:
'
VOID MENU_IENABLE(menue_ptr,schrift,0)
VOID MENU_IENABLE(menue_ptr,linien,0)
VOID MENU_IENABLE(menue_ptr,gsich,0)  ! gilt auch für 'Grafik sichern'
'
' Menüleiste zeigen:
'
VOID MENU_BAR(menue_ptr,1)
'
fenster_offen=0        ! Flag: Fenster noch nicht geöffnet
abbruch=0              ! Flag: noch kein Abbruch
puffer$=SPACE$(16)     ! Platz für GEM-Ereignispuffer
'
REPEAT
  neu_zeichnen=0       ! Flag: Grafik muß nicht neu gezeichnet werden
  '
  VOID EVNT_MESAG(VARPTR(puffer$))   ! Warten auf Nachricht...
  '  IF FN p(0)=10     ! Menüpunkt?
    title=FN p(3) ! Titel-Index merken
    '
    SELECT FN p(4)
    CASE info
      sub_info
    CASE laden
      sub_laden
    CASE sich
      sub_sich
    CASE gsich
      sub_gsich
    CASE ende
      sub_ende
    CASE dedit
      sub_dedit
    CASE balken
      sub_balken
    CASE torten
      sub_torten
    CASE schrift
      sub_schrift
    CASE linien
```

```
      sub_linien
    ENDSELECT
    '
    VOID MENU_TNORMAL(menue_ptr,title,1)  ! Titel normal darstellen
  ENDIF
  '
  IF FN p(0)=20 AND FN p(3)=whandle     ! Redraw
    redraw
  ENDIF
  '
  IF FN p(0)=21 AND FN p(3)=whandle     ! Topped
    neu_zeichnen=1
  ENDIF
  '
  IF FN p(0)=22 AND FN p(3)=whandle     ! Closed
    sub_ende
  ENDIF
  '
  ' Prüfen: Erfolgte eine Veränderung, die das Neuzeichnen der
  ' Grafik erforderlich macht?
  '
  IF neu_zeichnen=1
    IF fenster_offen=1
      '
      ' Prüfen, ob unser Fenster oben (aktiv) ist:
      '
      VOID WIND_GET(whandle,10,oben,dummy,dummy,dummy)
      '
      IF whandle<>oben
        VOID WIND_SET(whandle,10,0,0,0,0)
        kein_redraw=1    ! Redraw kommt automatisch nach WIND_SET
      ENDIF
    ENDIF
    zeichne_grafik
  ENDIF
  '
UNTIL abbruch=1
'
IF fenster_offen=1
  VOID WIND_CLOSE(whandle)
  VOID WIND_DELETE(whandle)
ENDIF
'
VOID MFREE(s_ptr)   ! Hier stand die Titelzeile
'
VOID MENU_BAR(menue_ptr,0)     ! Menüleiste löschen
'
VOID RSRC_FREE()               ! Resource-Daten löschen
VOID APPL_EXIT()
'
```

```
RESERVE
'
END
'
'
' Funktion zum Zugriff auf einzelne Worte des AES-Nachrichtenpuffers:
'
DEFFN p(x)=CVI(MID$(puffer$,x*2+1,2))
'
'
PROCEDURE sub_gsich
  FILESELECT path2$,default2$,ausgabe$
  IF ausgabe$<>"" AND RIGHT$(ausgabe$,1)<>"\"
    screen2=MALLOC(32000)
    IF screen2>0
      FOR i=0 TO 31996 STEP 4
        LPOKE screen2+i,0
      NEXT i
      '
      ' Grafik nach der File-Selektor-Box noch einmal ausgeben:
      ' Prüfen, ob unser Fenster oben (aktiv) ist:
      '
      VOID WIND_GET(whandle,10,oben,dummy,dummy,dummy)
      IF whandle<>oben
        VOID WIND_SET(whandle,10,0,0,0,0)
        kein_redraw=1    ! Redraw kommt automatisch nach WIND_SET
      ENDIF
      '
      zeichne_grafik
      '
      RC_COPY XBIOS(3),xwork,ywork,wwork,hwork TO screen2,xwork,ywork
      '
      BSAVE ausgabe$,screen2,32000
      '
      VOID MFREE(screen2)
    ELSE
      VOID FORM_ALERT(1,"[3][Dafür reicht der|Speicher nicht!][Schade]")
    ENDIF
  ENDIF
RETURN
'
'
PROCEDURE sub_torten
  IF akt_form<>torten
    akt_form=torten
    '
    VOID MENU_ICHECK(menue_ptr,balken,0)  ! Balken-Haken löschen
    VOID MENU_ICHECK(menue_ptr,torten,1)  ! Torten-Haken setzen
    '
```

```
    IF fenster_offen=1      ! Nur, wenn das Fenster schon geöffnet
neu_zeichnen=1        ! ist, soll neu gezeichnet werden (sonst
    ENDIF                   ! würde das Fenster womöglich ohne
  ENDIF                     ! sinnvolle Daten geöffnet)
RETURN
'
'
PROCEDURE sub_balken
  IF akt_form<>balken
    akt_form=balken
    '
    VOID MENU_ICHECK(menue_ptr,torten,0)
    VOID MENU_ICHECK(menue_ptr,balken,1)
    '
    IF fenster_offen=1
      neu_zeichnen=1
    ENDIF
  ENDIF
RETURN
'
'
PROCEDURE sub_schrift
  '
  ' Menüs, Windows desaktivieren:
  '
  VOID WIND_UPDATE(3)     ! Mauskontrolle an das Programm (= an uns)
  CLIP xwork,ywork,wwork,hwork
  '
  VOID GRAF_MOUSE(5,0)    ! Fadenkreuz
  a$=""
  '
  ' Tastaturpuffer leeren:
  '
  WHILE INKEY$<>""
  WEND
  '
  GRAPHMODE 3           ! XOR-Modus
  text_abbr=0
  x=MOUSEX
  y=MOUSEY
  '
  REPEAT
    IF MOUSEK=1         ! Knopf gedrückt
      TEXT x,y,a$       ! Text wieder löschen
      GRAPHMODE 1       ! Replace-Modus
      TEXT x,y,a$       ! Text 'richtig' schreiben
      text_abbr=1
    ENDIF
    '
    IF MOUSEX<>x OR MOUSEY<>y
```

```
      TEXT x,y,a$       ! alten Text löschen
      x=MOUSEX          ! neue Koordinaten übernehmen
      y=MOUSEY
      TEXT x,y,a$       ! Text neu schreiben
    ENDIF
    '
    z$=INKEY$
    IF z$<>""
      IF z$=CHR$(8)     ! Backspace?
        IF a$<>""
          TEXT x,y,a$
          a$=LEFT$(a$,LEN(a$)-1)
          TEXT x,y,a$
        ENDIF
      ELSE
        TEXT x,y,a$
        a$=a$+z$
        TEXT x,y,a$
      ENDIF
    ENDIF
    '
  UNTIL text_abbr=1
  '
  ' Text-Ausgabe in der Options-Liste eintragen:
  '
  INC next_opt
  opts$(next_opt)=CHR$(akt_form)+CHR$(0)+MKI$(x)+MKI$(y)+a$
  '
  VOID WIND_UPDATE(2)      ! Mauskontrolle an GEM
  VOID GRAF_MOUSE(0,0)     ! Pfeil als Zeiger
  CLIP OFF
RETURN
'
'
PROCEDURE sub_linien
  VOID WIND_UPDATE(3)      ! Mauskontrolle an das Programm (= an uns)
  CLIP xwork,ywork,wwork,hwork
  '
  VOID GRAF_MOUSE(5,0)     ! Fadenkreuz
  '
  REPEAT           ! Gelegenheit für den Benutzer, die Maustaste nach
  UNTIL MOUSEK=0   ! dem Klick im Menü wieder loszulassen
  '
  REPEAT           ! Warten auf Klick
  UNTIL MOUSEK=1
  '
  GRAPHMODE 3      ! XOR-Modus
  x1=MOUSEX        ! Koordinaten merken
  y1=MOUSEY
  '
```

```
  x=x1
  y=y1
  LINE x,y,x,y    ! ein Punkt muß schon gesetzt werden
  '
  REPEAT
    IF x<>MOUSEX OR y<>MOUSEY
      LINE x1,y1,x,y    ! Linie löschen
      x=MOUSEX
      y=MOUSEY
      LINE x1,y1,x,y    ! neue Linie zeichnen
    ENDIF
  UNTIL MOUSEK=0
  LINE x1,y1,x,y        ! wieder löschen
  GRAPHMODE 1           ! Replace-Modus
  LINE x1,y1,x,y        ! richtig zeichnen
  '
  ' In die Grafik-Liste eintragen:
  '
  INC next_opt
  opts$(next_opt)=CHR$(akt_form)+CHR$(1)+MKI$(x1)+MKI$(y1)+MKI$(x)+...
                            ...MKI$(y)
  '
  VOID WIND_UPDATE(2)   ! Kontrolle zurück an GEM
  VOID GRAF_MOUSE(0,0)  ! Pfeil als Meuszeiger
  CLIP OFF
RETURN
'
'
PROCEDURE sub_ende
  IF FORM_ALERT(2,"[2][Programm bestimmt beenden?][Ja|Nein]")=1
    abbruch=1
  ENDIF
RETURN
'
'
PROCEDURE sub_info
  do_dialog(info_ptr,0,dummy)
RETURN
'
'
PROCEDURE sub_dedit    ! Dateneditor
  write_text(daten_ptr,s1,bez$(1))
  write_text(daten_ptr,s2,bez$(2))
  write_text(daten_ptr,s3,bez$(3))
  write_text(daten_ptr,s4,bez$(4))
  write_text(daten_ptr,s5,bez$(5))
  write_text(daten_ptr,s6,bez$(6))
  write_text(daten_ptr,s7,bez$(7))
  write_text(daten_ptr,s8,bez$(8))
  '
```

```
write_text(daten_ptr,w1,wert$(1))
write_text(daten_ptr,w2,wert$(2))
write_text(daten_ptr,w3,wert$(3))
write_text(daten_ptr,w4,wert$(4))
write_text(daten_ptr,w5,wert$(5))
write_text(daten_ptr,w6,wert$(6))
write_text(daten_ptr,w7,wert$(7))
write_text(daten_ptr,w8,wert$(8))
'
do_dialog(daten_ptr,s1,geklickt)
'
IF geklickt=ok
  read_text(daten_ptr,s1,a$)        ! Sieht umständlich aus, oder?
  bez$(1)=a$
  read_text(daten_ptr,s2,a$)        ! Geht aber nicht anders!
  bez$(2)=a$
  read_text(daten_ptr,s3,a$)
  bez$(3)=a$
  read_text(daten_ptr,s4,a$)
  bez$(4)=a$
  read_text(daten_ptr,s5,a$)
  bez$(5)=a$
  read_text(daten_ptr,s6,a$)
  bez$(6)=a$
  read_text(daten_ptr,s7,a$)
  bez$(7)=a$
  read_text(daten_ptr,s8,a$)
  bez$(8)=a$
  '
  read_text(daten_ptr,w1,a$)        ! Jetzt wird's noch schlimmer...
  wert$(1)=a$
  wert#(1)=VAL(a$)
  read_text(daten_ptr,w2,a$)
  wert$(2)=a$
  wert#(2)=VAL(a$)
  read_text(daten_ptr,w3,a$)
  wert$(3)=a$
  wert#(3)=VAL(a$)
  read_text(daten_ptr,w4,a$)
  wert$(4)=a$
  wert#(4)=VAL(a$)
  read_text(daten_ptr,w5,a$)
  wert$(5)=a$
  wert#(5)=VAL(a$)
  read_text(daten_ptr,w6,a$)
  wert$(6)=a$
  wert#(6)=VAL(a$)
  read_text(daten_ptr,w7,a$)
  wert$(7)=a$
  wert#(7)=VAL(a$)
```

```
    read_text(daten_ptr,w8,a$)
    wert$(8)=a$
    wert#(8)=VAL(a$)
    '
    neu_zeichnen=1    ! Flag: Grafik bitte neu zeichnen
    kein_redraw=1     ! Aber: Wenn wir sowieso ganz neu zeichnen, dann
    '            können wir ein Redraw durch die Dialogbox ignorieren!
  ENDIF
RETURN
'
' Grafik (neu) zeichnen:
'
PROCEDURE zeichne_grafik
  IF fenster_offen=0    ! Fenster noch nicht offen?
    whandle=WIND_CREATE(3,xdesk,ydesk,wdesk,hdesk)
    '
    ' Titelzeile anlegen:
    '
    s_ptr=MALLOC(20)
    '
    FOR i=1 TO LEN(" DIAGRAMM "+CHR$(0))
      POKE s_ptr-1+i,ASC(MID$(" DIAGRAMM "+CHR$(0),i,1))
    NEXT i
    '
    VOID WIND_SET(whandle,2,s_ptr DIV 65536,s_ptr MOD 65536,0,0)
    '
    VOID WIND_OPEN(whandle,xdesk+5,ydesk+5,wdesk-10,hdesk-10)
    '
    ' Arbeitsbereich ausrechnen:
    '
    VOID WIND_CALC(1,3,xdesk+5,ydesk+5,wdesk-10,hdesk-10,xwork,...
                          ...ywork,wwork,hwork)
    '
    fenster_offen=1
    '
    kein_redraw=1      !  Redraw kommt automatisch nach WIND_OPEN    '
    ' Da das Fenster jetzt offen ist, können wir die Menüpunkte
    ' 'Beschriften' und 'Linien' zulassen:
    '
    VOID MENU_IENABLE(menue_ptr,schrift,1)
    VOID MENU_IENABLE(menue_ptr,linien,1)
    VOID MENU_IENABLE(menue_ptr,gsich,1)
    '
  ENDIF
  '
  BOUNDARY 0    !  Kein Rand beim Füllen
  DEFFILL 1,0
  PBOX xwork,ywork,xwork+wwork-1,ywork+hwork-1
  '
  IF akt_form=balken
```

```
    zeichne_balken
  ELSE
    zeichne_torte
  ENDIF
  '
  IF next_opt>0          ! Linien und Texte eintragen:
    CLIP xwork,ywork,wwork,hwork
    FOR i=1 TO next_opt
      IF ASC(opts$(i))=akt_form
        IF MID$(opts$(i),2,1)=CHR$(0)
          TEXT CVI(MID$(opts$(i),3,2)),CVI(MID$(opts$(i),5,2)),...
                          ...MID$(opts$(i),7)
        ELSE
          LINE CVI(MID$(opts$(i),3,2)),CVI(MID$(opts$(i),5,2)),...
                    ...CVI(MID$(opts$(i),7,2)),CVI(MID$(opts$(i),9,2))
        ENDIF
      ENDIF
    NEXT i
    CLIP OFF
  ENDIF
RETURN
'
'
PROCEDURE zeichne_balken
  '
  ' Feststellen, wieviele Werte zu zeichnen sind:
  '
  FOR i=1 TO 8
    EXIT IF wert$(i)=""
  NEXT i
  '
  last=i-1   ! Der zuletzt geprüfte Eintrag ist ja schon leer
  '
  ' Wertebereich berechnen:
  '
  max_wert#=wert#(1)
  min_wert#=wert#(1)
  IF last>1
    FOR i=2 TO last
      max_wert#=MAX(wert#(i),max_wert#)
      min_wert#=MIN(wert#(i),min_wert#)
    NEXT i
  ENDIF
  '
  IF min_wert#<0
    IF max_wert#=>0
      wertebereich#=max_wert#-min_wert#
      null_linie=TRUNC(ywork+40+(hwork-61)*max_wert#/wertebereich#)
    ELSE
      wertebereich#=-min_wert#
```

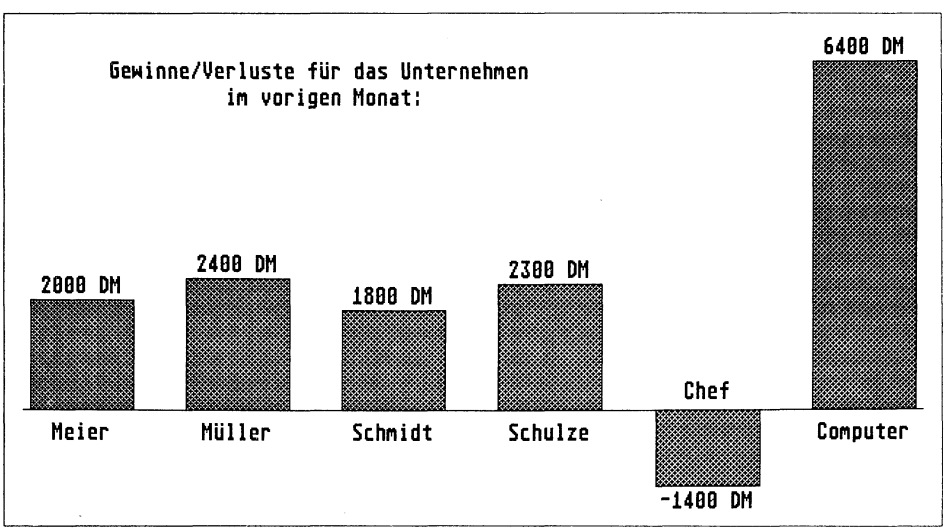
```
      null_linie=ywork+20
    ENDIF
  ELSE
    wertebereich#=max_wert#
    null_linie=ywork+hwork-41
  ENDIF
  '
  faktor#=(hwork-60)/wertebereich#
  '
  LINE xwork+5,null_linie,xwork+wwork-6,null_linie
  '
  breite#=(wwork-20)/(last+0.5*(last-1))
  '
  BOUNDARY 1
  DEFFILL 1,2,5
  '
  FOR i=1 TO last
    x1=TRUNC(xwork+10+1.5*(i-1)*breite#)
    x2=TRUNC(x1+breite#)
    y=TRUNC(null_linie-faktor#*wert#(i))
    '
    PBOX x1,null_linie,x2,y
    '
    IF SGN(wert#(i))=1
      ADD y,10
    ENDIF
    IF SGN(wert#(i))=0
      SUB y,3
    ENDIF
    '
    TEXT x1+(breite#-8*LEN(wert$(i)))\2,y-SGN(wert#(i))*15,wert$(i)
    '
    IF SGN(wert#(i))=-1
      by=null_linie-8
    ELSE
      by=null_linie+20
    ENDIF
    '
    TEXT x1+(breite#-8*LEN(bez$(i)))\2,by,bez$(i)
  NEXT i
RETURN
'
'
PROCEDURE zeichne_torte
  '
  ' Feststellen, wieviele Werte zu zeichnen sind:
  '
  FOR i=1 TO 8
    EXIT IF wert$(i)=""
  NEXT i
```

```
'
last=i-1
'
' Summe aller Beträge ausrechnen:
'
summe=0
FOR i=1 TO last
  ADD summe,ABS(wert#(i))
NEXT i
'
winkel=0
'
BOUNDARY 1     ! Umrandung wieder einschalten
'
ELLIPSE xwork+wwork\2,ywork+hwork\2+15,wwork\2-50,hwork\2-50,1800,3600
'
LINE xwork+50,ywork+hwork\2,xwork+50,ywork+hwork\2+15
LINE xwork+wwork-50,ywork+hwork\2,xwork+wwork-50,ywork+hwork\2+15
'
FOR i=1 TO last
  DEFFILL 1,2,i
  teilwinkel=TRUNC(3600*ABS(wert#(i))/summe)
  PELLIPSE xwork+wwork\2,ywork+hwork\2,wwork\2-50,hwork\2-50,...
                        ...winkel,winkel+teilwinkel
  '
  IF winkel+teilwinkel>1800 OR winkel+teilwinkel=0
    xx=xwork+wwork\2+(wwork\2-50)*COS((3600-winkel-teilwinkel)*...
                                      ...PI/1800)
    yy=ywork+hwork\2+(hwork\2-50)*SIN((3600-winkel-teilwinkel)*...
                                      ...PI/1800)
    LINE xx,yy,xx,yy+15
    '
    IF winkel<1800
      w=1800
      t=teilwinkel-1800+winkel
    ELSE
      w=winkel
      t=teilwinkel
    ENDIF
    xx=xwork+wwork\2+(wwork\2-50)*COS((3600-w-t\2)*PI/1800)
    yy=ywork+hwork\2+(hwork\2-50)*SIN((3600-w-t\2)*PI/1800)
    FILL xx,yy+8
  ENDIF
  '
  ADD winkel,teilwinkel
NEXT i
'
FOR i=1 TO last
  y=ywork+5+((i-1)\4)*(hwork-25)
  x=xwork+10+((i-1) MOD 4)*((wwork-20)\4)
```

```
    DEFFILL 1,2,i
    PBOX x,y,x+16,y+16
    TEXT x+21,y+13,bez$(i)+" ("+wert$(i)+")"
  NEXT i
  '
RETURN
'
'
PROCEDURE redraw
  LOCAL x,y,w,h
  '
  IF kein_redraw=1
    kein_redraw=0
  ELSE
    '
    ' Bitte keine Mausaktivitäten:
    '
    VOID WIND_UPDATE(1)
    '
    VOID WIND_GET(whandle,11,x,y,w,h)   ! 1. Rechteck aus Rechteckliste
    '
    WHILE NOT w=0
      '
      ' Mit Clip werden die Ausgaben auf das Rechteck begrenzt,
      ' das neu zu zeichnen ist. Es ist das Schnittrechteck aus dem
      ' Rechtecklisten-Rechteck und dem Rechteck aus der Nachricht.
      '
      IF RC_INTERSECT(FN p(4),FN p(5),FN p(6),FN p(7),x,y,w,h)
        CLIP x,y,w,h
        zeichne_grafik
      ENDIF
      '
      VOID WIND_GET(whandle,12,x,y,w,h)
    WEND
    '
    CLIP OFF
    VOID WIND_UPDATE(0)
  ENDIF
RETURN
'
'
PROCEDURE sub_laden
  FILESELECT path$,default$,ausgabe$
  IF ausgabe$<>"" AND RIGHT$(ausgabe$,1)<>"\"
    OPEN "I",#1,ausgabe$
    FOR i=1 TO 8
      LINE INPUT #1,bez$(i)
      LINE INPUT #1,wert$(i)
    NEXT i
    CLOSE #1
```

```
    '
    FOR i=1 TO 8                ! Umwandeln der Strings in Zahlen
      wert#(i)=VAL(wert$(i))
    NEXT i
    '
    neu_zeichnen=1    ! Weil wir ohnehin alles neu zeichnen, ist...
    kein_redraw=1     ! ... kein Redraw durch File-Selector mehr nötig
    '
    next_opt=0        ! keine Grafik-Objekte aus alter Datei mitnehmen
  ENDIF
RETURN
'
'
PROCEDURE sub_sich
  FILESELECT path$,default$,ausgabe$
  IF ausgabe$<>"" AND RIGHT$(ausgabe$,1)<>"\"
    OPEN "O",#1,ausgabe$
    FOR i=1 TO 8
      PRINT #1,bez$(i)
      PRINT #1,wert$(i)
    NEXT i
    CLOSE #1
  ENDIF
RETURN
'
' Nützliche Unterprogramme für Objekte:
'
PROCEDURE select(baum,index)
  OB_STATE(baum,index)=OB_STATE(baum,index) OR 1
RETURN
'
PROCEDURE deselect(baum,index)
  OB_STATE(baum,index)=OB_STATE(baum,index) AND -2
RETURN
'
PROCEDURE do_dialog(baum,start_ed,VAR exit_button)
  LOCAL x,y,w,h
  '
  VOID FORM_CENTER(baum,x,y,w,h)            ! Baum zeichnen, ...
  VOID FORM_DIAL(0,x,y,w,h,x,y,w,h)
  VOID FORM_DIAL(1,25,25,25,25,x,y,w,h)
  VOID OBJC_DRAW(baum,0,12,x,y,w,h)
  '
  exit_button=FORM_DO(baum,start_ed)        ! bearbeiten lassen ...
  deselect(baum,exit_button)
  '
  VOID FORM_DIAL(2,25,25,25,25,x,y,w,h)     ! und löschen
  VOID FORM_DIAL(3,x,y,w,h,x,y,w,h)
RETURN
'
```

```
PROCEDURE write_text(baum,index,text$)
  LOCAL adr,i,a$
  '
  adr=LPEEK(OB_SPEC(baum,index))
  a$=text$+CHR$(0)
  '
  FOR i=1 TO LEN(a$)
    POKE adr+i-1,ASC(MID$(a$,i,1))
  NEXT i
RETURN
'
PROCEDURE read_text(baum,index,VAR text$)
  LOCAL adr,i
  '
  adr=LPEEK(OB_SPEC(baum,index))
  text$=""
  i=0
  '
  WHILE PEEK(adr+i)<>0      ! bis zum Nullbyte lesen
    text$=text$+CHR$(PEEK(adr+i))
    INC i
  WEND
RETURN
'
```



5.11.4 Accessory: Programm-Kommunikation

Dieses Accessory paßt nicht so recht in die Gruppe der bisher vorgestellten Programme, weil es keinen wirklichen Nutzen bringt. Da die Idee an sich aber sehr praktisch sein kann, will Ich Ihnen das Programm hier dennoch als Beispiel-Accessory vorstellen.

Es geht dabei um den Austausch von Daten zwischen mehreren Programmen. Das Ungewöhnliche ist nun, daß diese Programme sich gleichzeitig im Hauptspeicher des Rechners befinden sollen; der Sender einer Nachricht und der Empfänger laufen also im Multitasking-System des GEM. Das bedeutet: Um eine derartige Kommunikation zu erreichen, brauchen wir mindestens zwei Programme.

Ich habe mich für zwei Accessories entschieden; das System arbeitet aber ebenso gut mit einem Accessory und einer Applikation (zwei Applikationen erlaubt GEM ja leider nicht gleichzeitig). Der Sender soll TALK (= sprechen) heißen, der Empfänger LISTEN (= zuhören). Nur TALK kann eine Nachricht an LISTEN senden; umgekehrt funktioniert es nicht.

Die Nachricht selbst können wir relativ einfach über das normale Nachrichten-System des AES verbreiten. Sie wissen ja bereits, daß eine typische AES-Nachricht aus acht Worten besteht, wobei das erste Wort den Typ der Nachricht angibt (Menüpunkt angeklickt, Fenster soll neue Größe bekommen, Accessory aufgerufen, Redraw-Meldung usw.). Jede Nachricht hat dazu bekanntlich eine feste Nummer. Dem Programmierer ist es nun gestattet, neue Nachrichten zu definieren und sie von einem Programm an ein anderes schicken zu lassen. Dabei sollte sichergestellt sein, daß das Empfängerprogramm die Nachricht auch versteht. Eine solche Nachricht könnte z.B. lauten: "Gib bitte den Text aus, dessen Startadresse in puffer[2] (high) und puffer[3] (low) steht!"

Das ist auch die Nachricht, die mein Beispielprogramm LISTEN verstehen kann. Ich habe ihr willkürlich die Nummer 99 zugeteilt. Das Empfängerprogramm sieht so aus:

```
/*************************************************/
/*   Accessories sprechen miteinander (Empfänger)   */
/*   Laser C         MP  08-01-89        LISTEN.C   */
/*************************************************/

#include "gem_inex.c"
```

```
int  ac_id,      /* Identifikations-Nr. des Accessorys */
     puffer[8];  /* Wie gewohnt: AES-Nachrichtenpuffer */

long *longs;     /* Pointer auf puffer, s.u. */

char str[100],   /* Hier basteln wir einen String zusammen */
     *message;   /* Hier kommt die Nachricht an */

main()
{
  /* Problem: Wir benötigen in einem ARRAY sowohl Worte als auch */
  /* Langworte. Lösung: Wir weisen einem zweiten ARRAY (bzw. */
  /* Pointer; ist ja fast dasselbe) die Startadresse des ersten */
  /* Arrays zu. Formal sind die Arrays verschieden (verschiedene */
  /* Typen: 1. int, 2. long), doch sie zeigen auf die gleichen Daten. */

  longs = (long *) puffer;

  gem_init();

  ac_id = menu_register (ap_id, "  Empfänger");   /* ACC anmelden */

  while (1)       /* Endlosschleife */
  {
    evnt_mesag (puffer);   /* Warten auf AES-Nachricht */

    switch (puffer[0])     /* Ereignistyp auswerten */
    {
      /* Möglichkeit 1: */
      /* Der Anwender hat ganz normal den Eintrag im Desk-Menü */
      /* angeklickt. Dann steht in puffer[0] wie üblich 40. */
      /* Wir geben nur einen kleinen Hinweis: */

      case 40:                          /* Accessory gefragt? */
        if (puffer[4] == ac_id)         /* etwa unser Accessory? */
          form_alert (1,"[1][Ich bin der Empfänger.|Schicken Sie mir...
                    ... doch mal|eine Nachricht!][Mach ich!]");
      break;

      /* Möglichkeit 2: */
      /* In puffer[0] steht unser 'Geheimcode' 99. Das AES kennt */
      /* diese Nachricht nicht, also muß sie vom Sende-Accessory */
      /* kommen. Im Wort 2 und 3 (oder im zweiten Langwort) der */
      /* Nachricht finden wir einen Zeiger auf den String, der die */
      /* übermittelte Nachricht enthält: */

      case 99:                  /* Aha, Nachricht vom Sender! */
        message = (char*) *(longs+1);   /* Zweites Langwort */
        strcpy (str, "[1][Empfänger erhielt Nachricht:| |");
        strcat (str, message);  /* Hier ist die Nachricht! */
```

```
        strcat (str, "][Verstanden.]");

        form_alert (1, str);   /* Das ganze bitte anzeigen! */
        break;
    }
  }
}
```

Der Sender muß natürlich die Fähigkeit besitzen, eine fremde Nachricht in das AES-Nachrichtensystem einzuschleusen. Das geht mit der Funktion appl_write des AES. Dieser muß als Parameter die Zahl der zu übertragenden Bytes (bei acht Worten also 16), die eigentliche Nachricht und die Identifikationsnummer des Zielprogramms übergeben werden. Letzteres ist der Rückgabewert des appl_init-Aufrufs im Zielprogramm. Doch wie soll der Sender an diesen Funktionswert kommen?

Dafür gibt es die Funktion appl_find. Der ist der Dateiname eines Programms oder eines Accessories zu übergeben. Wenn sich das Programm im Speicher befindet, so wird die Identifikations-Nummer des Programms zurückgeben. Wenn nicht, so erhalten Sie als Zeichen für einen Fehler -1 zurück. Wichtig ist, daß der Dateiname keine Extension enthält und nur aus Großbuchstaben besteht. Der Name muß genau acht Zeichen lang sein (eventuell mit Leerzeichen auffüllen).

```
/************************************************/
/*   Accessories sprechen miteinander (Sender)  */
/*   Laser C         MP  08-01-89       TALK.C  */
/************************************************/

#include "gem_inex.c"

int  ac_id,
     typ,
     listen_id,
     puffer[8];

long *longs;

char message[100];

main()
{
  longs = (long *) puffer;

  gem_init();

  ac_id = menu_register (ap_id, "  Sender");
```

```
while (1)       /* Endlosschleife */
{
  evnt_mesag (puffer);

  /* Dieses Accessory kann nur durch Klick im Desk-Menü aktiviert */
  /* werden. Deshalb haben wir auch nur den Ereignistyp 40 zu */
  /* prüfen: */

  /* Test: Unser Accessory? */
  if (puffer[0] == 40 && puffer[4] == ac_id)
  {
    /* Die Nachricht wird später mit appl_write geschrieben. Dazu */
    /* benötigen wir aber die Applikations-ID des Empfängers. Die */
    /* besorgen wir uns mit appl_find. Dabei stellen wir auch */
    /* fest, ob der Empfänger überhaupt geladen wurde */

    listen_id = appl_find ("LISTEN  ");

    if (listen_id == -1)     /* Nicht gefunden? */
      form_alert (1, "[3][Der Empfänger befindet sich|noch nicht...
                      ... im Speicher!][Ach?]");
    else
    {
      typ = form_alert (0, "[2][Welche Nachricht wünschen...
               ... Sie?| |1. Hallo Empfänger|2. Schönes Wetter...
               ... heute!|3. Atari ist super!][1|2|3]");

      switch (typ)
      {
        case 1: strcpy (message, "Hallo Empfänger");
                break;

        case 2: strcpy (message, "Schönes Wetter heute!");
                break;

        case 3: strcpy (message, "Atari ist super!");
                break;
      }

      /* Jetzt können wir die Nachricht senden. In puffer[0] */
      /* schreiben wir unseren 'Geheimcode' (99), der dem */
      /* Empfänger klarmacht, was wir eigentlich wollen. */
      /* Die Startadresse der Nachricht wird in puffer[2] */
      /* und [3] abgelegt (über longs, siehe Kommentar im */
      /* Listing LISTEN.C). */

      puffer[0]  = 99;              /* Code für Privatgespräch */
      *(longs+1) = (long) message;  /* entspricht puffer[2] und [3] */

      appl_write (listen_id, 16, puffer);  /* 16 Bytes senden */
```

```
      }
    }
  }
}
```

5.11.5 Applikation: Dateien codieren

Wenn Sie sich für das Thema Datenschutz interessieren, dann ist dieses Programm etwas für Sie: Es verschlüsselt beliebige Dateien oder Programme mit einem frei wählbaren Schlüssel. Nur wer den Schlüssel kennt, kann die Datei später wieder benutzen.

Zum Prinzip: Die Datei wird codiert, indem jedes Daten-Byte mit einem Zeichen des Codes exklusiv-oder verknüpft wird (d.h. ein gesetztes Bit im Code bewirkt, daß das entsprechende Bit der Daten negiert wird). Ist der Code z.B. zwei Zeichen lang, dann wird das erste Datenbyte mit dem ersten Codebyte verknüpft, das zweite Datenbyte mit dem zweiten Codebyte, das dritte Datenbyte wieder mit dem ersten Codebyte usw. Dabei hat die Exklusiv-oder-Verknüpfung den Vorteil, daß die gleiche Routine auch zum Entschlüsseln der Datei benutzt werden kann; denn eine zweite Negation hebt die erste auf.

Zum Ablauf des Programms: Die File-Selector-Box wird zur Eingabe eines Dateinamens aufgerufen. Sodann wird mit Fsfirst die Länge der gewählten Datei ermittelt und diese komplett in den Speicher geladen (wenn der Platz nicht reicht, gibt's eine Fehlermeldung). Nach der eigentlichen Codierung (s.o.) wird die Datei zurückgeschrieben. Der Anwender kann nun eine zweite Datei bearbeiten oder das Programm beenden.

Das Programm wurde in Assembler geschrieben und benötigt die beiden Include-Dateien GEM_INIT.Q und GEM_INEX.Q aus dem Ordner GEM.5. Bitte lassen Sie sich nicht durch den großen Umfang des Listings abschrecken; Sie wissen ja bereits, daß in Assembler alles ein paar Zeilen mehr benötigt als in Hochsprachen. Ein Hinweis noch: Der File-Selector verändert die DTA (Disk-Transfer-Address; siehe Listing), die ja für die Fsfirst-Funktion gesetzt werden muß. Der Aufruf der Funktion Fsetdta muß also nach dem Aufruf des File-Selectors erfolgen.

```
;
; Dateien codieren/decodieren (Demo-Applikation)
; Assembler         MP 11-01-89          CODE.Q
;
```

```
gemdos      = 1     ;Konstanten zu GEMDOS-Routinen
dgetdrv     = $19
fsetdta     = $1a
fopen       = $3d
fclose      = $3e
fread       = $3f
fwrite      = $40
fseek       = $42
dgetpath    = $47
malloc      = $48
mfree       = $49
fsfirst     = $4e

DIALOG      = 0     ;Konstanten zum Resource-File
FNAME       = 3
KEY         = 5
ABB         = 7
OK          = 6


            INCLUDE 'GEM_INEX.Q'


            TEXT

main:       jsr     gem_init    ;wie üblich

; rsrc_load:

            move.w  #110,control
            clr.w   control+2
            move.w  #1,control+4
            move.w  #1,control+6
            clr.w   control+8
            move.l  #rscname,addr_in
            jsr     aes

            tst.w   int_out     ;Fehler?
            beq     rsc_err

; rsrc_gaddr ermittelt Startadresse des Dialogs:

            move.w  #112,control
            move.w  #2,control+2
            move.w  #1,control+4
            clr.w   control+6
            move.w  #1,control+8
            clr.w   int_in      ;0 für 'Baum gesucht'
            move.w  #DIALOG,int_in+2  ;Index des Baumes
            jsr     aes
```

```
            move.l  addr_out,baum_adr ;Ergebnis: die Startadresse

            lea     code,a0       ;Schlüsselfeld leeren
            clr.b   (a0)
            move.w  #KEY,d0
            jsr     write_text

; Für den Fileselector muß zunächst mal ein Pfadname
; gebastelt werden:

            clr.b   default       ;kein Default-Name bei Programmstart
            lea     fsel_pfad,a2

; Aktuelles Lauswerk erfragen

            move.w  #dgetdrv,-(sp)
            trap    #gemdos
            addq.l  #2,sp
            addi.b  #'A',d0       ;Funktionsergebnis -> Buchstabe
            move.b  d0,(a2)
            move.b  #':',1(a2)    ;Doppelpunkt hinter Laufwerk

; Pfad des aktuellen Laufwerks erfragen

            clr.w   -(sp)         ;aktuelles Laufwerk
            pea     2(a2)
            move.w  #dgetpath,-(sp)
            trap    #gemdos
            addq.l  #8,sp

; \ und Maske anhängen

            clr.w   d1            ;Stringlänge ermitteln
fsel_lp:    tst.b   0(a2,d1.w)    ;Nullbyte?
            beq.s   fsel_le
            addq.w  #1,d1
            bra.s   fsel_lp

fsel_le:    move.b  #'\',0(a2,d1.w)

            lea     maske,a3      ;Zeiger auf *.*
fsel_l2:    move.b  (a3)+,1(a2,d1.w)
            beq.s   fsel_e2
            addq.w  #1,d1
            addq.w  #1,d2
            bra.s   fsel_l2

fsel_e2:
mainloop:
```

```
; Aufruf der GEM-File-Selector-Box

            move.w  #90,control  ;Funktions-Opcode
            clr.w   control+2
            move.w  #2,control+4
            move.w  #2,control+6
            clr.w   control+8

            move.l  #fsel_pfad,addr_in     ;vorbereiteter Pfadname
            move.l  #default,addr_in+4     ;und Default-Dateiname

            jsr     aes

            tst.w   int_out+2    ;Abbruch statt OK angeklickt?
            beq     fsel_q       ;dann abbrechen
            lea     default,a3
            tst.b   (a3)         ;Dateiname überhaupt ausgewählt?
            beq     fsel_q

; Jetzt müssen wir nur noch aus dem Pfad- und dem Dateinamen
; einen kombinierten Pfad- und Dateinamen machen:

            lea     fsel_pfad,a0 ;zur Bearbeitung kopieren
            lea     pfadname,a2
cpy_loop:   move.b  (a0)+,(a2)+
            bne.s   cpy_loop

fsel_l4:    cmpi.b  #'\',-(a2)   ;Backslash suchen
            bne.s   fsel_l4

            addq.l  #1,a2        ;Backslash stehen lassen

fsel_l5:    move.b  (a3)+,(a2)+  ;Dateiname kopieren
            bne.s   fsel_l5

; Testen, ob die Datei überhaupt existiert. Dabei stellen
; wir auch gleich die Länge in Bytes fest:

            pea     dta_buff     ;DTA-Buffer festlegen
            move.w  #fsetdta,-(sp)
            trap    #gemdos
            addq.l  #6,sp

            clr.w   -(sp)        ;normale Datei suchen
            pea     pfadname     ;Name
            move.w  #fsfirst,-(sp)
            trap    #gemdos
            addq.l  #8,sp

            tst.w   d0           ;Fehler?
```

```
            bmi     error

; Speicher für die Datei besorgen:

            move.l  filesize,-(sp)
            move.w  #malloc,-(sp)
            trap    #gemdos
            addq.l  #6,sp
            movea.l d0,a6       ;Startadresse merken
            tst.l   d0          ;nicht genug Speicher da?
            bmi     mem_err

; Datei öffnen und laden:

            move.w  #2,-(sp)    ;Öffnen zum Lesen und Schreiben
            pea     pfadname
            move.w  #fopen,-(sp)
            trap    #gemdos
            addq.l  #8,sp
            move.w  d0,fhandle  ;Handle merken
            tst.w   d0          ;Fehler?
            bmi     error

            pea     (a6)        ;Startadresse für Laden
            move.l  filesize,-(sp)    ;zu lesende Bytes
            move.w  fhandle,-(sp)
            move.w  #fread,-(sp)
            trap    #gemdos
            adda.l  #12,sp

; Datei codieren:

            lea     pfadname,a0 ;Dateinamen in Formular schreiben
            move.w  #FNAME,d0
            jsr     write_text

            jsr     show_dialog ;Anzeigen des Baumes

; form_do läßt Dialog abarbeiten:

            move.w  #50,control
            move.w  #1,control+2
            move.w  #1,control+4
            move.w  #1,control+6
            clr.w   control+8
            move.w  #KEY,int_in ;Edit-Feld
            move.l  baum_adr,addr_in
            jsr     aes
            move.w  int_out,knopf     ;gedrückter Knopf
```

```
                jsr     deselect        ;Selected-Status löschen

                jsr     hide_dialog     ;Dialogbox vom Bildschirm entfernen

break:

                move.w  knopf,d0        ;Welcher Knopf wurde gedrückt?
                cmpi.w  #OK,d0
                bne     fsel_q          ;Abbruch -> Ende

                lea     code,a0         ;Code auslesen
                move.w  #KEY,d0
                jsr     read_text

                lea     code,a0         ;a0 wurde zerstört
                tst.b   (a0)            ;Code überhaupt eingegeben?
                beq     no_code

                lea     (a6),a4         ;Startadresse
                lea     (a6),a5
                adda.l  filesize,a5     ;Endadresse

code_rep:       lea     code,a0         ;Codewort
code_loop:      tst.b   (a0)            ;Ende des Codes?
                beq.s   code_rep        ;dann von vorne anfangen
                move.b  (a0)+,d0        ;Code-Byte holen
                eor.b   d0,(a4)+        ;Exklusiv-Oder-Verknüpung
                cmpa.l  a4,a5           ;Ende erreicht?
                bgt.s   code_loop       ;noch nicht, dann weiter

break2:

; Datei zurückschreiben:

                clr.w   -(sp)           ;Zeiger an den Anfang der Datei
                move.w  fhandle,-(sp)
                clr.l   -(sp)           ;Position relativ zum Dateianfang
                move.w  #fseek,-(sp)
                trap    #gemdos
                adda.l  #10,sp

                pea     (a6)            ;Datei zurückschreiben
                move.l  filesize,-(sp)
                move.w  fhandle,-(sp)
                move.w  #fwrite,-(sp)
                trap    #gemdos
                adda.l  #12,sp

                move.w  fhandle,-(sp)     ;Datei schließen
                move.w  #fclose,-(sp)
```

```
            trap    #gemdos
            addq.l  #4,sp

; Speicher wieder freigeben:

            pea     (a6)
            move.w  #mfree,-(sp)
            trap    #gemdos
            addq.l  #6,sp

no_code:    bra     mainloop


mem_err:    lea     out_of_mem,a5     ;Alert-Box zeigen
            jsr     alert
            bra     fsel_q

rsc_err:    lea     rsc_txt,a5
            jsr     alert
            bra     fsel_q

error:      lea     not_found,a5
            jsr     alert

fsel_q:     jsr     gem_exit

; rsrc_free:

            move.w  #111,control
            clr.w   control+2
            move.w  #1,control+4
            clr.w   control+6
            clr.w   control+8
            jsr     aes

            clr.w   -(sp)         ;Pterm0
            trap    #gemdos       ;(Programmende)


alert:
; Anzeigen einer Alert-Box. String in a5.

            move.w  #52,control   ;Opcode
            move.w  #1,control+2
            move.w  #1,control+4
            move.w  #1,control+6
            clr.w   control+8
            move.w  #1,int_in     ;Default-Button
            move.l  a5,addr_in    ;beschreibender String
            jsr     aes
```

```
            rts

; Unterprogramme zum Umgang mit GEM:

show_dialog:
; Dieses Unterprogramm malt einen Objektbaum auf den
; Bildschirm. Dazu muß sich dessen Startadresse unter
; 'baum_adr' befinden.

; form_center

            move.w  #54,control
            clr.w   control+2
            move.w  #5,control+4
            move.w  #1,control+6
            clr.w   control+8
            move.l  baum_adr,addr_in
            jsr     aes
            move.w  int_out+2,d4 ;Koordinaten sichern
            move.w  int_out+4,d5
            move.w  int_out+6,d6
            move.w  int_out+8,d7

; form_dial rettet Fensterränder etc. (0)

            move.w  #51,control
            move.w  #9,control+2
            move.w  #1,control+4
            clr.w   control+6
            clr.w   control+8
            clr.w   int_in       ;Unterfunktion 0
; keine Werte für das kleine Rechteck fo_dilittlx/y/w/h
            move.w  d4,int_in+10 ;großes Rechteck do_dibigx/y/w/h
            move.w  d5,int_in+12
            move.w  d6,int_in+14
            move.w  d7,int_in+16
            jsr     aes

; form_dial zeichnet 'Zoom'-Rechteck (1):

            move.w  #51,control
            move.w  #9,control+2
            move.w  #1,control+4
            clr.w   control+6
            clr.w   control+8
            move.w  #1,int_in    ;Unterfunktion 1
            move.w  d4,int_in+2  ;Größe des kleinen Rechtecks
            move.w  d5,int_in+4
            move.w  #1,int_in+6
```

```
                move.w  #1,int_in+8
                move.w  d4,int_in+10 ;großes Rechteck do_dibigx/y/w/h
                move.w  d5,int_in+12
                move.w  d6,int_in+14
                move.w  d7,int_in+16
                jsr     aes

; Dialog zeichnen mit objc_draw:

                move.w  #42,control
                move.w  #6,control+2
                move.w  #1,control+4
                move.w  #1,control+6
                clr.w   control+8
                clr.w   int_in       ;0=Wurzelobjekt zuerst zeichnen
                move.w  #12,int_in+2 ;max. 12 Ebenen (willkürlich)
                move.w  d4,int_in+4
                move.w  d5,int_in+6
                move.w  d6,int_in+8
                move.w  d7,int_in+10
                move.l  baum_adr,addr_in
                jmp     aes


hide_dialog:
; Entfernt das Formular vom Bildschirm.
; Adresse des Objektbaums wieder in baum_adr

; form_center

                move.w  #54,control
                clr.w   control+2
                move.w  #5,control+4
                move.w  #1,control+6
                clr.w   control+8
                move.l  baum_adr,addr_in
                jsr     aes
                move.w  int_out+2,d4 ;Koordinaten sichern
                move.w  int_out+4,d5
                move.w  int_out+6,d6
                move.w  int_out+8,d7

; form_dial zeichnet schrumpfendes-Rechteck (2):

                move.w  #51,control
                move.w  #9,control+2
                move.w  #1,control+4
                clr.w   control+6
                clr.w   control+8
                move.w  #2,int_in    ;Unterfunktion 2
```

```
                move.w  d4,int_in+2  ;Größe des kleinen Rechtecks
                move.w  d5,int_in+4
                move.w  #1,int_in+6
                move.w  #1,int_in+8
                move.w  d4,int_in+10 ;großes Rechteck do_dibigx/y/w/h
                move.w  d5,int_in+12
                move.w  d6,int_in+14
                move.w  d7,int_in+16
                jsr     aes

; form_dial sendet Redraw-Meldungen an Fenster (3)

                move.w  #51,control
                move.w  #9,control+2
                move.w  #1,control+4
                clr.w   control+6
                clr.w   control+8
                move.w  #3,int_in    ;Unterfunktion 3
; keine Werte für das kleine Rechteck fo_dilittlx/y/w/h
                move.w  d4,int_in+10 ;großes Rechteck do_dibigx/y/w/h
                move.w  d5,int_in+12
                move.w  d6,int_in+14
                move.w  d7,int_in+16
                jmp     aes


deselect:
; Schaltet den Button 'knopf' auf 'nicht-selected'-Status
; Objektbaum muß in baum_adr stehen

                movea.l baum_adr,a0
                move.w  knopf,d0     ;Index des Objekts
                mulu    #24,d0       ;* 24 (jedes Objekt hat 24 Bytes)
                addi.w  #10,d0       ;+10 als Offset für ob_state
                andi.w  #-2,0(a0,d0.w)    ;Bit 0 (selected) löschen
                rts


write_text:
; Schreibt String in Text- oder Edit-Objekt (Index in d0)
; String in a0, Objektbaum in baum_adr

                movea.l baum_adr,a1
                mulu    #24,d0
                movea.l 12(a1,d0.w),a1    ;Adresse des TEDINFO-Blocks
                movea.l (a1),a1      ;te_ptext enthält den Text

wrt_lp:         move.b  (a0)+,(a1)+
                bne.s   wrt_lp
```

```
            rts


read_text:
; Gegenstück zu write_text. Zieladresse ist in a0
; zu übergeben.

            movea.l baum_adr,a1
            mulu    #24,d0
            movea.l 12(a1,d0.w),a1
            movea.l (a1),a1

rd_lp:      move.b  (a1)+,(a0)+
            bne.s   rd_lp
            rts


            DATA

maske:      DC.b '*.*',0
rscname:    DC.b 'CODE.RSC',0

not_found:  DC.b '[3][Diese Datei existiert|aber gar nicht!]'
            DC.b '[So?]',0

out_of_mem: DC.b '[3][Für diese Datei ist mein|Speicher '
            DC.b 'zu klein!][Erweitern?]',0

rsc_txt:    DC.b '[3][Ich kann das RSC-File|nicht finden!]'
            DC.b '[Abbruch]',0

            BSS
            ALIGN

knopf:      DS.w 1
baum_adr:   DS.l 1
fhandle:    DS.w 1

dta_buff:   DS.b 26             ;DTA-Buffer für Fsfirst
filesize:   DS.l 1              ;hier steht die Größe in Bytes
            DS.b 14             ;(der Rest ist reserviert)

fsel_pfad:  DS.b 40
pfadname:   DS.b 40
default:    DS.b 13
code:       DS.b 40

            END
```

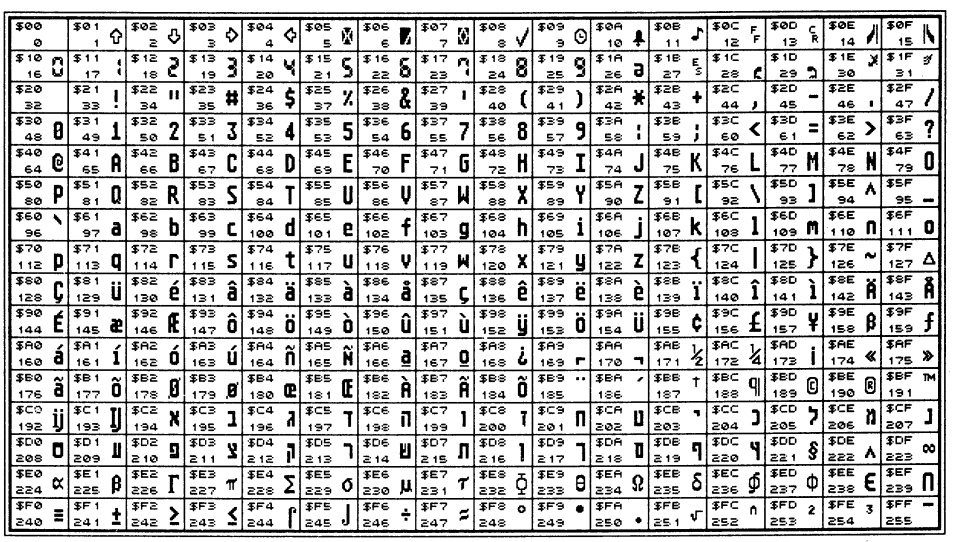
Jetzt sind wir am Ende des Buches angekommen und Sie haben viel Neues kennengelernt. Sie sind nun in der Lage, eigene Applikationen und Accessories zu entwickeln. Wenn Sie dazu weitere Anregungen suchen, oder auch Kniffe kennenlernen wollen, so kann ich Ihnen das Buch "Die besten Tips & Tricks zum Atari ST" wärmstens empfehlen. Auch in diesem Buch finden Sie in jeder Programmiersprache interessante Anwendungen, die fertig auf der beiliegenden Diskette ebenfalls vorhanden sind. Werfen Sie ruhig einmal einen Blick hinein, es lohnt sich wirklich. Und nun wünsche ich Ihnen noch viel Spaß bei der Entwicklung eigener Programme.

Anhang

Anhang A

ASCII-Codes

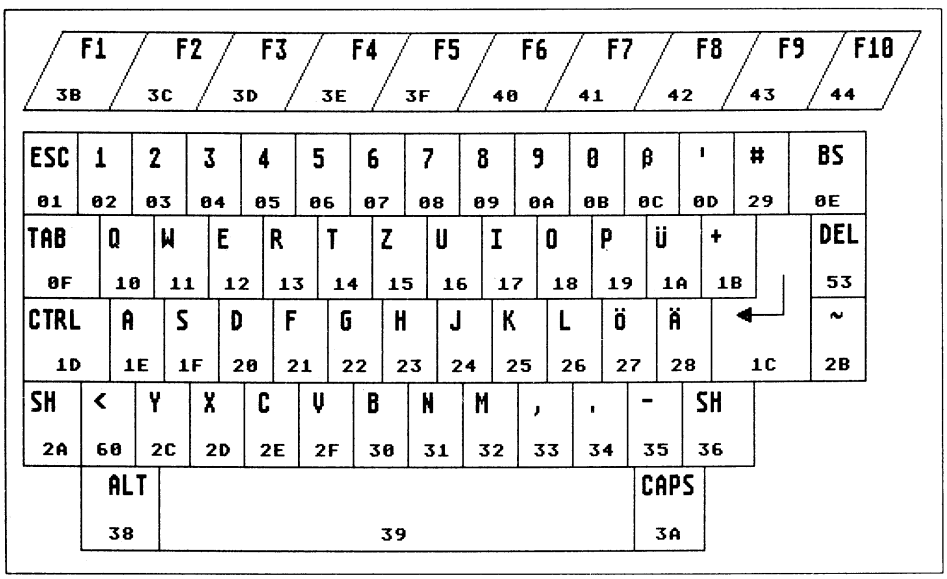
Hinweis: Unter TOS können die Zeichen mit den Codes 0 bis 31 nur mit der BIOS-Funktion Bconout (Nr. 3) ausgeben werden. Als device, also als Gerätenummer, muß eine 5 angegeben werden.

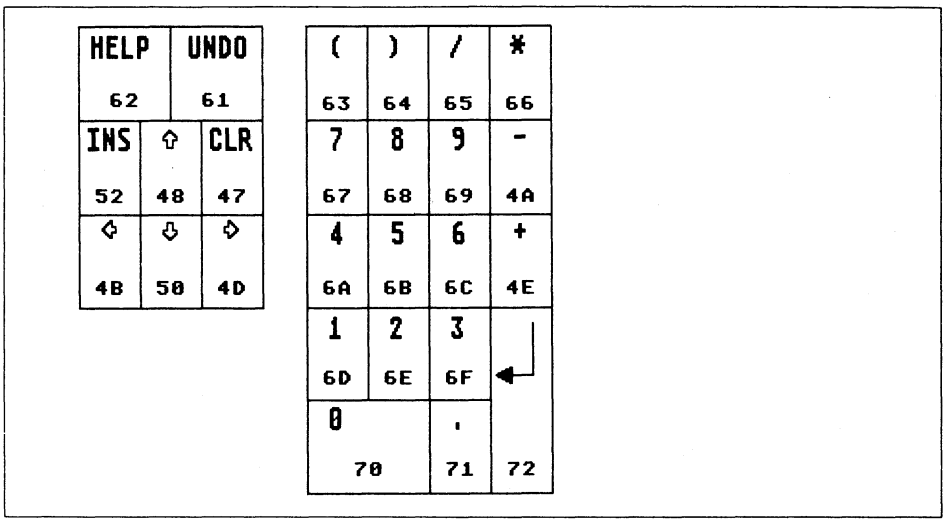


Anhang B

Scan-Codes

Diese Tabelle zeigt alle Tasten der ST-Tastatur. Jede der Tasten liefert dem System einen bestimmten Scan-Code. Sie können diese Tabelle immer dann benutzen, wenn Sie Sondertasten (Funktions-, Cursortasten) abfragen müssen oder zwischen den Ziffern des Zahlenblocks und des Hauptblocks unterscheiden wollen. Alle Codes sind im Hexadezimalsystem angegeben.




HELP
62
UNDO
61
INS
52
⇧
48
CLR
47
⇦
4B
⇩
50
⇨
4D
(
63
)
64
/
65
*
66
7
67
8
68
9
69
-
4A
4
6A
5
6B
6
6C
+
4E
1
6D
2
6E
3
6F
0
70
.
71
72

Anhang C

Noten und Frequenzen

Diese Tabelle ist für die Soundprogrammierung wichtig. Für eine Reihe von Oktaven sind hier die Frequenzen aller Noten und die zugehörigen Periodendauern aufgelistet. Es wurde auch schon die Aufteilung der Periodendauer in ein High- und ein Low-Byte vorgenommen.

Note	Frequenz	Periodendauer	High-Byte	Low-Byte
C	65.41	1911	7	119
Cis=Des	69.30	1804	7	12
D+	73.42	1703	6	167
Dis=Es	77.78	1607	6	71
E	82.41	1517	5	237
F	87.31	1432	5	152
Fis=Ges	92.50	1351	5	71
G	98.00	1276	4	252
Gis=As	103.83	1204	4	180
A	110.00	1136	4	112
Ais=B	116.54	1073	4	49
H	123.47	1012	3	244
c	130.82	956	3	188
cis=des	138.59	902	3	134
d	146.84	851	3	83
dis=es	155.57	804	3	36
e	164.82	758	2	246
f	174.62	716	2	204
fis=ges	185.00	676	2	164
g	196.00	638	2	126
gis=as	207.66	602	2	90
a	220.00	568	2	56
ais=b	233.08	536	2	24
h	246.94	506	1	250

Note	Frequenz	Periodendauer	High-Byte	Low-Byte
c^1	261.63	478	1	222
$cis^1 = des^1$	277.18	451	1	195
d^1	293.67	426	1	170
$dis^1 = es^1$	311.13	402	1	146
e^1	329.63	379	1	123
f^1	349.23	358	1	102
$fis^1 = ges^1$	369.99	338	1	82
g^1	392.00	319	1	63
$gis^1 = as^1$	415.31	301	1	45
a^1	440.00	284	1	28
$ais^1 = b^1$	466.16	268	1	12
h^1	493.88	253	0	253
c^2	523.26	239	0	239
$cis^2 = des^2$	554.36	225	0	225
d^2	587.34	213	0	213
$dis^2 = es^2$	622.26	201	0	201
e^2	659.26	190	0	190
f^2	698.46	179	0	179
$fis^2 = ges^2$	739.98	169	0	169
g^2	784.00	159	0	159
$gis^2 = as^2$	830.62	150	0	150
a^2	880.00	142	0	142
$ais^2 = b^2$	932.32	134	0	134
h^2	987.76	127	0	127
c^3	1046.52	119	0	119
$cis^3 = des^3$	1108.72	113	0	113
d^3	1174.68	106	0	106
$dis^3 = es^3$	1244.52	100	0	100
e^3	1318.52	95	0	95
f^3	1396.92	89	0	89
$fis^3 = ges^3$	1479.96	84	0	84
g^3	1568.00	80	0	80
$gis^3 = as^3$	1661.24	75	0	75
a^3	1760.00	71	0	71
$ais^3 = b^3$	1864.64	67	0	67
h^3	1975.52	63	0	63

Anhang D

Verzeichnis der VT-52-Sequenzen

Wenn Sie mit den TOS-Funktionen zur Bildschirm-Text-Ausgabe arbeiten, dann gibt es bestimmte Zeichenkombinationen, die nicht als Buchstaben oder Ziffern auf dem Bildschirm erscheinen, sondern bestimmte Sonderfunktionen auslösen. Da diese Funktionen sehr praktisch sind, habe ich sie auch in meinen Beispielprogrammen benutzt.

Alle diese Sequenzen werden durch den ASCII-Code der Escape-Taste, also durch den Wert 27, eingeleitet. In C schreibt man diese Zahl oktal; dann wird daraus \33.

Die wichtigsten dieser Steuercodes möchte ich Ihnen kurz vorstellen. Beachten Sie, daß zwischen Groß- und Kleinbuchstaben unterschieden wird:

```
ESCAPE E
```

löscht den Bildschirm und bringt den Cursor in die linke obere Ecke des Schirms.

```
ESCAPE H
```

bringt den Cursor in die linke obere Bildschirmecke, ohne den Schirm dabei zu löschen.

```
ESCAPE Y zeile spalte:
```

Setzt den Cursor auf eine Position, die durch die Zeichen (!!!, ggf. mit CHR$() umwandeln) zeile und spalte ausgedrückt wird. Für zeile und spalte gilt: Ein Wert von 32 bedeutet oberste Zeile oder linker Bildrand. Der Steuerstring CHR$(27)+'Y'+CHR$(32)+ CHR$(33) bringt den Cursor also in die erste Zeile an die zweite Spalte.

```
ESCAPE e:
```

Der Cursor erscheint auf dem Bildschirm an der aktuellen Cursorposition.

```
ESCAPE f:
```

Der Cursor wird ausgeschaltet.

ESCAPE p:

Alle folgenden Zeichen erscheinen invertiert. Beim Monochrom-Monitor werden Zeichen jetzt weiß auf schwarzem Hintergrund ausgegeben.

ESCAPE q:

Macht ESCAPE p rückgängig. Die Zeichen erscheinen wieder normal.

Zwei weitere wichtige Codes werden ebenfalls sehr häufig verwendet. Ihnen wird jedoch kein Escape vorangestellt:

13: Carriage Return (Wagenrücklauf)

Der Cursor wird an den linken Rand der aktuellen Cursorzeile gesetzt.

10: Line Feed (Zeilenvorschub)

Der Cursor wird um eine Zeile nach unten verschoben. Befindet er sich bereits in der untersten Zeile des Bildschirms, so wird der Bildschirminhalt um eine Zeile nach oben gerollt.

Gewöhnlich werden beide Codes hintereinander benutzt, um bei der Text-Ausgabe unter TOS am Anfang der nächsten Zeile weiterzuschreiben. Hängt man ein weiteres Line Feed an, so entsteht zusätzlich eine Leerzeile zwischen dem bisherigen und dem im folgenden auszugebenden Text.

Anhang E

Programmverzeichnis

Soweit nicht anders angegeben, befindet sich auf der Diskette im Buch jedes Programm fünfmal: für GFA-BASIC (.GFA), für Omrikon-BASIC (.BAS), für C (.C), für Assembler (.Q) und als lauffähiges Programm (.TOS, .PRG oder .ACC). Manche Programme benötigen zur Laufzeit zusätzliche Dateien (in Klammern angegeben); .DEF- und .DFN-Dateien (nur im Ordner DEMOS.511) dienen zur Bearbeitung der Resource-Files mit einem Resource-Construction-Programm.

Ordner: TOS.1

Dateiname		Kapitel
FONTOUT		1.2
TASTDEMO	(nicht für Omikron)	1.2
ONLINE		1.2
CHROUT	(nicht für Omikron)	1.3
STROUT	(nicht für Omikron)	1.3
FILES	(+ READ.ME)	1.4.1
SHOWFILE	(+ READ.ME)	1.4.2
DIR		1.4.2
UMLEITEN		1.6
NACHLAD	(+ ZULADEN.TOS)	1.7
NACHLAD2	(+ ZULADEN.TOS)	1.7
VIRUS		1.8

Ordner: GRAFIK.2

Dateiname		Kapitel
MULTISCR	(nicht für GFA-BASIC)	2.1
SPRITE		2.2

Ordner: INTERRUP.3

Dateiname		Kapitel
VBL-QUEUE	(nicht für BASIC)	3.1
TIMER_A	(nicht für BASIC)	3.2
RASTER	(nur Assembler)	3.3
RASTER2	(nur Assembler)	3.3

Ordner: SOUND.4

Dateiname		Kapitel
SOUND1	(nur für GFA-BASIC)	4.2
SOUND2		4.3
SOUND3		4.3
SOUND4		4.4

Ordner: GEM.5

Dateiname		Kapitel
GEMINIT.Q	(nur für Assembler)	5.3
DEMOAPP		5.4
GEM_INEX	(nur für C/Assembler)	5.5
VDI_DEMO	(nicht für GFA-BASIC)	5.5.1
LINIEN	(nicht für GFA-BASIC)	5.5.2
COPYRAST	(nicht für GFA-BASIC)	5.5.3
BUTTON		5.6
MULTI		5.6
FSEL		5.7
WIND1		5.8
WIND2		5.8.1
WIND3		5.8.2
BASHEAD	(nur Omikron und .TOS)	5.9.1
MENU1	(+ .RSC, .H und .H2)	5.9.1
MENU2	(+ .RSC, .H und .H2)	5.9.1
DIALOG1	(+ .RSC, .H und .H2)	5.9.2
DIALOG2	(+ .RSC, .H und .H2)	5.9.2
DIALOG3	(+ .RSC, .H und .H2)	5.9.2
DIALOG4	(+ .RSC, .H und .H2)	5.9.2
DEMOACC	(nicht für GFA-BASIC)	5.10

Ordner: DEMOS.511

Dateiname		Kapitel
ASCII	(ACC)	5.11.1
LISTER	(+ .RSC, .PAR, .DEF, .DFN)	5.11.2
DIAGRAMM	(+ .RSC, .DEF, .DFN, .DIF)	5.11.3
LISTEN	(ACC)	5.11.4
TALK	(ACC)	5.11.4
CODE	(+ .RSC, .DEF, .DFN)	5.11.5

Stichwortverzeichnis

S

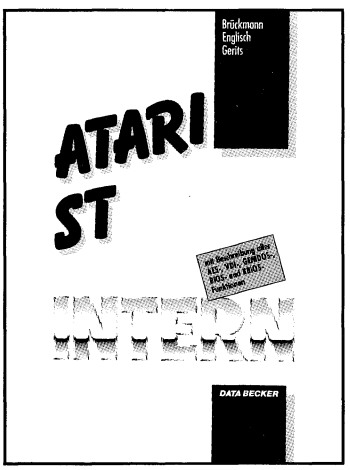
T

U

X

Y

Dieser INTERN-Band ist das Standardbuch zur Programmierung der Atari-ST-Computer. Sie finden alle Informationen zum Aufbau und zur Funktion Ihres Rechners, die zur professionellen Programmierung unentbehrlich sind. Dabei sind die Beispiele in den vier wichtigsten Programmiersprachen angegeben: Omikron-Basic V3.0 (ST-Basic), GFA-Basic, C und Assembler. So wird jeder Programmierer in seiner Sprache unterstützt. Durch den übersichtlichen und konsequent didaktischen Aufbau ist das Buch ideal als Nachschlagewerk geeignet.



Aus dem Inhalt:

- Das TOS vom 06.02.86 (1040ST/F), vom 22.04.87 (Blitter-TOS) und das brandneue TOS 1.4
- Aufbau und Anschlüsse des 68000-Prozessors
- Funktionen und Register des Custom-Chips
- Der Blitter und seine Programmierung
- Der Soundgenerator YM-2149
- Programmierung des Tastaturprozessors
- Alles über die Schnittstellen der ST-Modelle
- DMA-Betrieb mit Timing-Diagrammen
- Der Erweiterungsbus im Mega-ST
- Speicheraufbau
- Die Systemvariablen und ihre Bedeutung
- TOS: GEMDOS, BIOS und XBIOS-Funktionen mit ausführlichen Beschreibungen und Beispielen in BASIC, C und Assembler
- GEM: VDI und AES-Funktionen mit einführenden Erläuterungen und Beispielen in den vier wichtigsten Programmiersprachen
- Grafikprogrammierung
- Dokumentiertes BIOS-ROM-Listing des ‚normalen' und des Blitter-TOS
- Tabellen zum schnellen Nachschlagen: GEMDOS, BIOS, XBIOS

Brückmann, Englisch, Gerits
Atari ST intern, Band 1
Hardcover, 732 Seiten, DM 69,—
ISBN 3-89011-119-X

Bei der Arbeit mit dem Atari ST tauchen immer wieder Probleme auf. Genau dort setzt dieses Buch an. Es beantwortet alle Fragen zu den Bereichen Desktop, Massenspeicher, Drucker, Schnittstellen... kurz zu allem, was zum Atari ST gehört. Der gut strukturierte Aufbau des Buches mit zusätzlichen Symbolen zum schnellen Finden der Lösungen in lexikonähnlicher Form macht dieses Buch zum unentbehrlichen Nachschlagewerk, das das nötige Hintergrundwissen liefert.

Aus dem Inhalt:

- Desktop: Umgang mit Icons, Menüleiste und Windows, Bedienung von Objektauswahlboxen
- Massenspeicher: Schnelles Kopieren mit einem Laufwerk, Umbenennen von Ordnern, Tips zu Festplatten
- Drucker: Zeichensätze, DIP-Schalter, Druckeranpassung
- Schnittstellen: Centronics, RS232, DMA Datenübertragung
- Computerwissen: Aufbau des Computers, Zahlensysteme
- Software: Tips & Tricks zu Standardsoftware
- Pflege und Wartung: Einbau von TOS-Roms, kleine Reparaturen selbst gemacht
- Glossar

Liesert
Das große Atari ST Handbuch
Hardcover, 370 Seiten, DM 49,—
ISBN 3-89011-273-0

Der Mega ST, das Flaggschiff von Atari, verfügt über eine Vielzahl von Anwendungsmöglichkeiten, die sich u. a. aus dem extrem großen Arbeitsspeicher dieses Rechners ergeben. Insbesondere in den Bereichen Desktop Publishing (DTP) und Programmierung erweist sich der Mega ST als vielseitig und anderen Geräten weit überlegen. In diesem Buch gehen die Autoren ausführlich auf die Möglichkeiten dieses Rechners ein und weisen darüber hinaus auf alle technischen Einzelheiten hin, deren Kenntnis für jeden Mega-ST-Besitzer unerläßlich ist. Besonders interessant ist es für alle, die das dokumentierte Blitter-TOS für die eigene Programmentwicklung nutzen wollen.

Aus dem Inhalt:

- Die Bedienung und die Handhabung des Mega ST
- Die Software für den Mega ST (Textverarbeitungen, Datenbanken, Tabellenkalkulationen usw.)
- Desktop Publishing mit dem Mega ST (Beckerpage ST usw.)
- Die Programmierung des Mega ST (AES, VDI, BIOS, XBIOS, GEMDOS usw.)
- Die Pheripherie des Mega ST (Festplatte, Laserdrucker usw.)
- Der Aufbau des Rechners
- Die Nutzung des großen RAM-Speichers – RAM-Disk bis 3,5 MByte mit Autokonfiguration
- Nützliche Beispielprogramme und ein ausführlich dokumentiertes Blitter-TOS-Listing

Dittrich, Englisch, Severin
Das große Mega ST Buch
Hardcover, 538 Seiten, DM 69,—
ISBN 3-89011-196-3

Dieses Buch möchte vermeiden, daß beim Programmieren das Rad noch einmal erfunden werden muß. Struktogramme zu Standard-Algorithmen und einführende Erläuterungen fügen sich zu einem Nachschlagewerk zusammen, mit dem sowohl Anfänger als auch Fortgeschrittene zurechtkommen. Zu den möglichst portabel gehaltenen Algorithmen kommen in einem zweiten Teil nützliche Tools für den ST. Gleichzeitig bietet dieses Buch einen Einstieg in das neue GFA-BASIC Version 3.0, eine komplett dokumentierte AES-Liste der neu hinzugekommenen AES-Bibliothek sowie eine Gegenüberstellung der Befehle GFA V2.0 und V3.0, um Ihnen die Anpassung Ihrer alten Programme zu erleichtern.

Aus dem Inhalt:

- Einführung in die Benutzung von Struktogrammen
- Textverarbeitung: Flatter- und Blocksatz, Silbentrennung, Wordwrap
- Datenverarbeitung: Sequentielle, Index-Sequentielle und Random-Access-Dateien
- Grafik: Vektorgrafik, Drehen, Spiegeln und Verschieben von Objekten, Barcodes
- Mathematik: Matrizenrechnung, Nullstellen, Integral- und Differentialrechnung, Finanzmathematik, Zahlensysteme, Statistik
- Kalenderberechnung
- Strategie-Algorithmen
- Tools: Automatisches Backup, REM-Killer, Diagramme, 3-D-Funktionsplotter
- GFA V3.0: AES-Liste, Befehlsliste GFA V2.0 – V3.0

Liesert, Linden
Das große GFA-Programmier-Handbuch
Tools & Algorithmen
Hardcover, 480 Seiten, DM 59,–
ISBN 3-89011-258-7

Bücher zum ATARI ST

Die besten Tips und Tricks zum ATARI ST sind eine wahre Fundgrube für jeden ATARI-ST-Besitzer. Anhand vieler nützlicher Routinen werden die fantastischen Möglichkeiten dieses Rechners ausführlich erklärt. Programmier- und Hardwaretips vermitteln zusätzlich eine Menge über die Rechnerstruktur und dessen Programmierung in GFA-Basic, C und Assembler.

Aus dem Inhalt:

- GEM-Starter
- Uhrzeit resetfest
- echtes Multitasking
- Sprite-Programmierung
- schnelle Grafikroutinen
- Dia-Show
- Sound-Programmierung
- Konvertierungsprogramme
- Floppy-Speeder
- Short-Cuts für beliebige Programme
- Accessory-Aufbau
- Filesearch für Festplatte
- Bildschirmschoner
- Ordner umbenennen
- Einschaltverzögerung für Festplatte

Pauly, Schepers, Schulz
Die besten Tips & Tricks
Hardcover, 428 Seiten, DM 59,—
ISBN 3-89011-210-2

Wer bisher glaubte, professionelle Programme könne man nur in C oder gar Assembler entwickeln, wird mit diesem Buch eines Besseren belehrt. Nach einem ausführlichen Basic-Grundkurs erfahren Sie alles über die Dateiverwaltung, die Nutzung von Betriebssystemroutinen, die Grafikprogrammierung oder die Programmierung unter GEM. Nützliche Tools und Libaries runden das Buch ab.

Aus dem Inhalt:

- Variablentypen in ST-Basic
- Strings und Stringmanipulation
- Strukturierte Programmierung
- Rekursionen
- Formatierte Ein- und Ausgabe
- Sequentielle und relative Dateiverwaltung
- Logische Verknüpfungen
- Betriebssystemprogrammierung
- GEM-Programmierung
- Eigene File-Selector-Box
- Multitasking
- Omikron.Compiler
- Nützliche Libaries

Maier
Das große ST-BASIC-Buch
Hardcover, 407 Seiten, DM 49,—
ISBN 3-89011-283-8

Endlich gibt es GFA V3.0! Zu diesem umfangreichen BASIC gehört auch ein umfangreiches Buch, in dem detailliert jeder Befehl behandelt wird. Dabei liefert es keine nackte Befehlsübersicht, sondern wirklich brauchbares Material in Hülle und Fülle. Anhand zahlreicher Beispielprogramme lernen Sie dieses leistungsfähige BASIC spielend zu beherrschen.

Aus dem Inhalt:

- Das Editor-Menü
- Variablentypen und -organisation
- Diskettenoperationen
- Strukturierte Programmierung
- Mausabfrage in eigenen Programmen
- Sound-Programmierung
- Beschreibung des Resource-Construction-Set
- Verwendung von Multitasking-Befehlen
- Programmieren von Pull-Down-Menüs
- Abfrage von Ereignissen (Events)
- Window-Programmierung
- Zugriff auf GEMDOS, BIOS und XBIOS
- Komplette AES- und VDI-Library-Beschreibung
- Verwenden eigener Fonts mit GDOS
- Eine komplette Adressenverwaltung als RAM-Kartei
- Komplette Befehlsübersicht über alle Befehle des GFA-BASIC

Litzkendorf
Das große GFA Basic 3.0 Buch
Hardcover incl. Diskette, 828 Seiten, DM 49,—
ISBN 3-89011-222-6

Das steht drin:

Ein so leistungsfähiger Rechner wie der ATARI ST erwartet von seinen Programmierern umfangreiche Gundkenntnisse, um ihn zu beherrschen. Der zweite INTERN-Band vermittelt ihnen genau diese Kenntnisse und zeigt Ihnen in vielen Beispielprogrammen, wie Sie das Betriebssystem und die Hardware sinnvoll einsetzen können. Fast alle Programme sind in den vier Sprachen GFA-BASIC, OMIKRON-BASIC (ST-Basic), C und Assembler abgedruckt. Den Schwerpunkt des Buches bildet die Programmierung von GEM-Applikationen und -Accessories.

Aus dem Inhalt:

- Benutzung von GEMDOS, BIOS und XBIOS-Funktionen
- Dateien unter GEMDOS
- Speicherverwaltung
- Umlenken der Ein-/Ausgabe
- logische und physikalsche Bildschirme
- Sprite-Programmierung
- Interrupts (auch in C)
- Soundprogrammierung
- GEM-Programmierung:
  Benutzung von AES und VDI, Grafikfunktionen, Ereignisverwaltung, die File-Selector-Box, Windows, Menüs, Dialogboxen und Accessories
- umfangreiche Demo-Applikationen und Accessories

Und geschrieben hat dieses Buch:

Martin Pauly, seit Jahren begeisterter ST-Fan. Er kennt nicht nur das Betriebssystem und die Hardware seines Rechners in- und auswendig, sondern beherrscht auch die verschiedensten Programmiersprachen. Mit diesem Buch möchte er allen ST-Besitzern zeigen, wie die vielfältigen Möglichkeiten des ATARI ST eingesetzt werden.

ISB N 3-89011-324-9 DM +079.00

DM 79,–
ÖS 616,–
sFr 77,–

07900

9 783890 113241

DATA BECKER