Robert Tolksdorf

GEM – Praxis auf dem Atari ST

Resources, Dialogboxen und Menüs

Inhalt

Einleitung

"42"
Per Anhalter duch die Galaxis

Erst durch Resources bekommen Programme auf dem ATARI ST das richtige "GEM-Aussehen". Mit ihnen lassen sich Drop-Down-Menüs und Dialogboxen programmieren.

In diesem Buch wird detailliert auf alles eingegangen, was mit Resources zu tun hat. Von der Beschreibung der benötigten Systemroutinen über Resource-Construction-Sets bis zu vielen ausführlich kommentierten Beispielprogrammen.

Resources bilden neben den Fenstern einen Hauptbestandteil von GEM. "GEM-artige" Programme lassen sich auch schreiben, ohne Fenster zu benutzen. Oft reichen einfache Resources aus, um einem Programm durch Grafikanzeigen und Mausbedienung den nötigen Komfort zu geben.

Die Beispielprogramme demonstrieren jeden Teilaspekt der Resource-Programmierung. Behandelt werden normale Dialogboxen, Menüs oder Möglichkeiten der Programmsteuerung mit Icons.

Die Programme sind in Modula-2 geschrieben, da diese Sprache eine gute Lesbarkeit der Listings und eine Selbstdokumentation der Programme besser erlaubt als C. Allerdings ist es möglich, auch auf andere Sprachen zu wechseln. Dabei helfen die Kapitel 10 und 11 sowie der Anhang A.

Mit dem Buch wird der Programmier-Praktiker angesprochen, der seinen eigenen Programmen zu einer größeren Benutzerfreundlichkeit verhelfen will und dazu über die Resource-Programmierung ein Wissen benötigt, das über die reine Beschreibung der GEM-Funktionen hinausgeht.

Wer sich die Tipparbeit bei den Listings ersparen will, der kann diese fehlerträchtige Arbeit umgehen, indem er die Diskette zum Buch erwirbt.

Beim Arbeiten mit dem Buch und beim Verwenden der beschriebenen Möglichkeiten wünscht Ihnen der Autor viel Spaß!

Berlin, Dezember 1987

1. Der erste Schritt

Wer tastet sich nachts die Finger klamm?
Es ist der Bitkönig mit seinem Programm.
Er tastet geschwind, er tastet schnell,
Im Osten wird schon der Himmel hell.
Sein Haar ist ergraut, die Hände zittern
vom unablässigen Speicherfüttern.

Die Ballade vom Bitkönig

Zuerst lernen Sie einfache Routinen kennen, die Sie sofort ausprobieren können. Außerdem erhalten Sie erste Einblicke in die Arbeit der Resource-Funktionen.

1.1 Alarm, Alarm! - Die Alertboxen

Klicken Sie auf dem Desktop eine Datei an, die kein Programm ist, also nicht die Endungen *PRG*, *TOS*, *APP* oder *TTP* hat, erscheint auf dem Bildschirm eine Box mit der Aufforderung, einen von drei Buttons zu wählen: "AUSGEBEN", "DRUCKEN" oder "ABBRUCH". "ABBRUCH" ist dicker umrandet und gibt sich so als "Default" zu erkennen. Durch Drücken der <Return>-Taste wird diese Option automatisch ausgewählt.

Bild 1.1.1: Die Desktop-Alertbox

Was bedeutet diese Box, und wie kann man sie selbst erzeugen? Nun, es handelt sich um die einfachste Möglichkeit, eine grafische Auswahl zu ermöglichen. Das ganze heißt "Alertbox" und ist die erste Resource-Routine, mit der Sie arbeiten können.

Eine Alertbox besteht aus mehreren Elementen:
- einem Rechteck, das den Hintergrund bildet
- einem Bild (Icon)
- einem Text
- einigen Buttons, die ausgewählt werden können
- einem Default-Button.

Die Funktion, die das alles auf den Bildschirm bringt heißt *FormAlert* (in den Libraries bei C-Compilern *form_alert*) und benötigt als Parameter Informationen über diese Bestandteile.

FormAlert(DefButton:INTEGER; VAR String:ARRAY OF CHAR):INTEGER;
exbuttb=form_alert(defbuttn,string)

Die Routine wird aufgerufen mit

result:=FormAlert(DefButton,String);

Nun zu den Parametern, von denen zunächst *String* betrachtet werden soll. In dieser Zeichenkette sind die Informationen über das Icon, den Text und die Buttons enthalten. Ihr Aufbau ist:

"[" Icon "][" Text "][" Buttons "]".

Bei Icon steht die Nummer des Icons, das links in der Box erscheinen soll. Dabei besteht die Auswahl zwischen:
0 = Kein Icon
1 = Ausrufezeichen (Note)
2 = Fragezeichen (Wait)
3 = STOP-Zeichen.

1 2 3

Bild 1.1.2: Die drei Icons für Alertboxen

Bei der oben beschriebenen Box müßte hier also eine 1 stehen. Dabei ist wirklich das Zeichen "1" gemeint, demnach "[1]...".

Für *Text* stehen die Zeichen, die in der Box ausgegeben werden sollen. Sie können bis zu 200 Zeichen eingeben, die in fünf Zeilen bis zu je vierzig Zeichen dargestellt werden. In der niedrigen Auflösung sind es nur halb so viele, denn eine Alarm-Box darf nicht mehr als ein Viertel der gesamten Bildschirmfläche einnehmen. Soll ein Programm in allen drei Auflösungen laufen können, muß sich der Programmierer auf 20 Zeichen pro Zeile beschränken. Zum Trennen einzelner Zeilen in der Alertbox wird das Zeichen "|" verwendet. Findet die Routine dieses Zeichen, so erfolgt die Ausgabe eine Zeile tiefer.

Um die Meldung
"Alarm,
 Alarm!"
darzustellen, müßte hier also
"...[Alarm,|Alarm!]..."
stehen.

Schließlich werden noch die einzelnen Buttons definiert. Dabei können Sie bis zu drei Buttons verwenden, deren Text im letzten "[ ]"-Paar steht. In jedem Button lassen sich bis zu 20 Zeichen unterbringen.

Um die einzelnen Buttons zu trennen, wird wieder das "|"-Zeichen verwendet. Für die beiden Buttons "OK" und "Cancel", muß "...[OK|Cancel]" übergeben werden.

Damit sind alle veränderbaren Elemente der Box festgelegt. Es fehlt noch die Angabe des Default-Buttons. Dies steht in dem ersten Parameter *DefButton*. Hier wird die Nummer des gewünschten Buttons angegeben. Im obigen Beispiel wird eine 1 für den "OK"-Button übergeben.

Das Ergebnis der Funktion gibt an, welcher Button ausgewählt wurde, gleichgültig, ob durch die Maus oder durch die <Return>-Taste. "OK" ergäbe eine 1 und "Cancel" eine 2.

Nun können Sie sich ans Werk machen und ein kleines Programm schreiben, das die Alertbox vom Desktop erzeugt:

```
MODULE FormAlertDemo;

(* FormAlert importieren *)
FROM AESForms   IMPORT FormAlert ;

CONST AlertString =
'[1][|Diese Datei kann nur|gedruckt oder gezeigt werden.]
 [ANZEIGEN|DRUCKEN|ABBRUCH]';

VAR result:INTEGER;
```

```
BEGIN
  result:=FormAlert(3,AlertString); (* 3 = ABBRUCH als Default annehmen *)
END FormAlertDemo.
```

Mit "FROM Forms IMPORT FormAlert" wird die Funktion *FormAlert* aus dem Bibliotheksmodul *AESForms* dem Programm zugänglich gemacht. In C müßte hier mit *EXTERN* gearbeitet werden.

AlertString enthält Informationen über die Box. Es soll das Ausrufezeichen-Bild erscheinen, also "[1]". Der Text besteht aus zwei Zeilen, die durch "¦" getrennt werden. Davor soll jedoch eine Leerzeile stehen. Diese wird durch das einzelne "¦" am Anfang erzeugt.

Die Texte der drei Buttons werden der Reihe eingegeben und durch "¦" getrennt. Als Default-Button soll "ABBRUCH" gelten. Daher wird als *DefButton* eine 3 übergeben.

result enthält nach Beendigung der Funktion die Nummer des ausgewählten Buttons. Das Programm fängt hier nichts weiter mit dem Wert an, normalerweise werden aber mit einer *CASE*- oder *SELECT*-Anweisung, je nach Wert, unterschiedliche Aktionen ausgeführt. Dies können Sie sich auch am Verhalten des Desktops verdeutlichen.

Damit könnten Sie schon Abfragen mit der Maus in eigene Programme einbauen. Probieren Sie dies auch einmal in eigenen kleinen Beispielen aus!

1.2 TOS-Fehler - Entsetzen für den Anwender

Eine eingebaute GEM-Funktion, die *FormAlert* benutzt, ist *FormError*. Sie benötigt nur einen Parameter, nämlich eine Fehlernummer.

```
FormError(ErrorNum:INTEGER):INTEGER;
exbttn=form_error(num)
```

Bild 1.2.1: FormError

Beim Aufruf wird eine Alertbox dargestellt, die den Text "TOS Fehler #xx" und nur den "ABBRUCH"-Button enthält. "xx" steht für die Zahl, die als Fehlernummer in *ErrorNum* übergeben wurde.

Das Ergebnis der Funktion ist die Nummer des ausgewählten Buttons. Da nur einer vorhanden ist, wird immer eine 1 geliefert.

Zur Übung sollten Sie einmal probieren, eine Box mit der Fehlernummer 5 unter Verwendung von *FormAlert* auf den Bildschirm zu bringen. GEM selbst macht nichts anderes.

1.3 Alles einfach?

Bisher haben Sie nur diese beiden Funkionen benutzt, wobei Sie wußten, wie das gemacht wird und welches Ergebnis Sie erzeugen. Wie arbeitet *FormAlert* jedoch intern?

FormAlert übernimmt insgesamt sechs Aktionen, die wiederum auf anderen Funktionen beruhen:

- Aus *String* wird eine interne Datenstruktur erzeugt, die von den benutzten Routinen benötigt wird.
- Der Bildschirmbereich, den die Alertbox überschreibt, wird in einen internen Puffer übertragen.
- Die Alertbox wird in der Mitte des Bildschirms dargestellt.
- GEM wartet, bis ein Button ausgewählt wird. Dies kann mit der Maus oder der <Return>-Taste geschehen. Wird die Maus außerhalb der Box gedrückt, erklingt ein Warnton.
- Der oben genannte Puffer wird wieder in den Bildschirm übertragen. Dadurch ist der überschriebene Bereich wiederhergestellt.
- GEM ermittelt die Nummer des ausgewählten Buttons und gibt sie an den Funktionsaufrufer zurück.

Jeder dieser Schritte bedingt wiederum eine Reihe weiterer Operationen. Im nächsten Kapitel geht es um die interne Datenstruktur, in die GEM den Parameter *String* umwandelt und mit der auch alle anderen Resources dargestellt werden.

2. Objektstruktur

Da - aus dem Speicher tönt ein Geflüster,
Wer wühlt da in meinem Carry-Register?
Bleib ruhig, bleib ruhig, mein AES,
Irgendwann hört er auf, der Streß.

Ballade vom Bitkönig

2.1 Bäume - Gegen die Schwerkraft

Resources werden bei GEM in Bäumen dargestellt. Die Routinen verlangen als Parameter meistens die Adresse des ersten Baumelements. Was sind nun aber Bäume, und mit welcher Spielart dieser Struktur arbeitet GEM?

Bäume sind Datenobjekte, deren Struktur einem auf den Kopf gestellten Baum ähnelt: Es gibt eine Wurzel, Äste und Blätter. Die Wurzel dieser Bäume wird immer als "oben" angenommen - etwas gegen die Natur.

Im Bild 2.1.1 ist ein einfacher Baum dargestellt. Anhand dieses Beispiels sollen einige Begriffe geklärt werden. Bei GEM sehen die Bäume zwar etwas anders aus, dazu jedoch später.

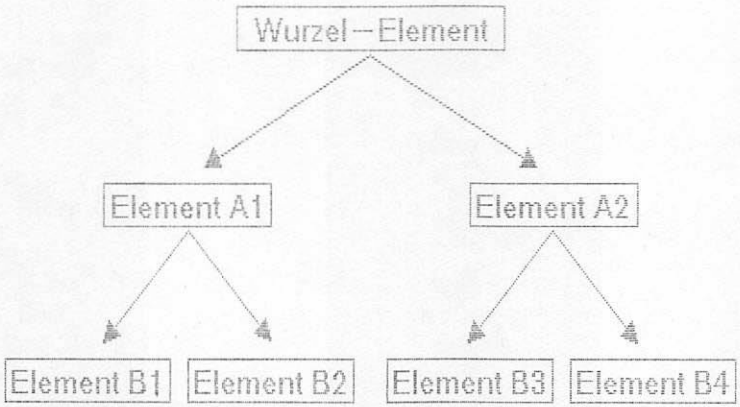


Bild 2.1.1: Ein einfacher Baum

Der Baum besteht aus einer Reihe von Objekten. Sie sind zumeist *RECORD*s oder *STRUCTURE*s, in denen die eigentlichen Informationen gehalten werden. Bei GEM sind dies Angaben über die Resource-Elemente und die Verwaltung für den Baum.

Diese Verwaltungsinformationen sind in der Regel Zeiger. Bei diesem einfachen Baum zeigen sie auf die Nachfolger eines Objekts.

Oben im Baum steht die Wurzel, die als einzige keinen Vorgänger hat. Im Beispiel hat sie zwei Nachfolger, nämlich A1 und A2. Das Wurzelobjekt hat somit zwei Zeiger, die auf die Nachfolger deuten.

Zwischen der Wurzel (Parent oder "Elter") und ihren Nachfolgern (Child oder "Kind") besteht eine hierarchische Beziehung. Die Ebene der Wurzel ist "höher" als die der Kinder.

Die Kinder selbst haben ebenfalls Kinder, für die sie nun wieder Eltern sind. Dadurch kann man sich vorstellen, daß bei A1 und A2 jeweils ein "Unterbaum" beginnt.

Die Objekte der Ebene B haben keine Kinder mehr. Sie werden "Blätter" genannt. Machen Sie sich klar, daß ein Programm über die Zeiger von der Wurzel aus zu jedem Element des Baumes "traversieren" kann.

GEM benutzt eine etwas andere Baumstruktur, in der andere Zeiger mit anderen Bedeutungen verwendet werden. Bild 2.1.2 zeigt den Baum, wie er für die Alertbox aus Kapitel 1 nötig wäre.

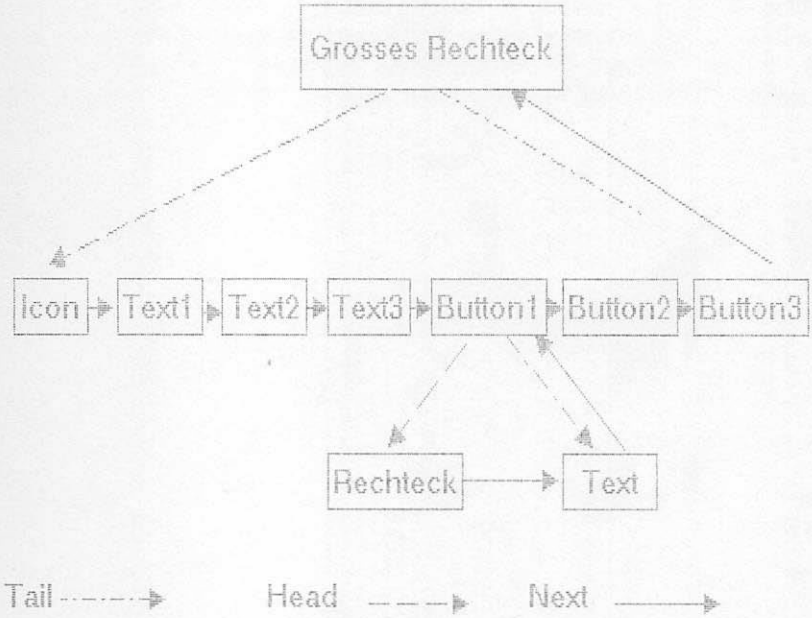


Bild 2.1.2: Ein GEM-Baum

Oben befindet sich die Wurzel, an der der Baum "aufgehängt" ist. Jedes Baumelement hat drei Zeiger, die den Baum zusammenhalten.

Der erste Zeiger ist der Kopfzeiger; bei GEM wird er *head* genannt. Er zeigt auf das erste Objekt der nächsttieferen Ebene. In unserm Beispiel ist dies das Icon.

Der zweite Zeiger, *tail*, zeigt auf das letzte Element der nächsten Baumebene. Hier handelt es sich um den dritten Button.

In der zweiten Ebene wird der dritte Zeiger benutzt. Er heißt *next* zeigt auf das jeweils nächste Objekt. Dadurch wird eine Ordnung innerhalb der Ebenen festgelegt.

Ist ein Element das letzte in einer Ebene, so zeigt *next* zurück auf das übergeordnete Objekt. Verdeutlichen Sie sich wieder, daß man auch jetzt mittels der Zeiger durch den Baum traversieren kann. Der Weg wäre dann ein großes Dreieck, dessen Spitzen auf die Wurzel sowie aufs erste und letzte Element der A-Ebene zeigen.

Hat ein Element einen Unterbaum, wie die Buttons im Beispiel, so dienen der *head*- und *tail*-Zeiger dazu, den Unterbaum zusammenzuhalten.

Die unbenutzten Zeiger werden auf den Wert *NIL* (Not In List) gesetzt. Das bedeutet, der Zeiger ist nicht gültig und soll nicht benutzt werden. Bei den meisten Programmiersprachen wird NIL intern durch den Wert -1 ersetzt.

Durch diese Ebenen wird auch eine Hierarchie zwischen den Objekten und Gruppen von Objekten festgelegt. Die Buttons liegen "unter" dem großen Rechteck, und werden deshalb innerhalb von diesem gezeichnet. Dieser Aspekt wird im Kapitel drei noch einmal wichtig.

Falls Sie die Baumstruktur noch nicht ganz verstanden haben, versuchen Sie einmal selbst, in unserm Beispiel von der Wurzel zum Text des ersten Buttons zu traversieren.

Das Listing zeigt eine Modula-2 Prozedur, die einen Baum oder einen Unterbaum traversiert. Durch die rekursiven Aufrufe wird sie extrem kurz, besucht jedoch jedes Baumobjekt.

```
PROCEDURE traverse(startobject:ObjectType);
VAR object:POINTER TO ObjectType;
BEGIN
  object:=startobject.head;
  REPEAT
    IF object.head#NIL THEN (* head benutzt *)
      traverse(object) (* Unterbaum durchwandern *)
    END;
    object:=object.next      (* zum nächsten *)
  UNTIL object=startobject; (* fertig *)
END;
```

In dieser Darstellung der Baumstruktur befindet sich noch ein kleiner Fehler, den es jetzt auszuräumen gilt. Auf GEM-Bäume wird nämlich nicht direkt über Zeiger, sondern mit *INTEGER*-Werten zugegriffen.

Dabei wird der Baum in ein Feld von Objekten gepackt, wobei jedes Element einen Index hat, der den Wert des "Zeigers" angibt. Die Wurzel hat normalerweise den Index 0.

GEM verwaltet die Lage eines Objekts im Speicher. Zu diesem Zweck gibt es eine interne Tabelle, auf die der Programmierer normalerweise nicht direkt zugreifen muß. Bei der Benutzung der GEM-Routinen werden zur Identifikation eines Objekts die Indizes verwendet.

So läßt sich ansatzweise darstellen, wie ein Objekt eines Resource-Baums aufgebaut ist (links für Modula-2 und Pascal und rechts in C-Schreibweise):

```
Object = RECORD                    struct object
           next,                   { int next;
           head,                     int head;
           tail: INTEGER;            int tail;
          ...                        ...
         END;                      } OBJECT;

Tree = ARRAY [1..N] OF Object;     OBJECT tree[N];
```

Bild 2.1.3: Vorläufiger Objektaufbau

Die *traverse*-Prozedur muß umgeschrieben werden, damit sie mit den Feld-Indizes arbeiten kann.

Im nächsten Kapitel können Sie sich mit dem genauen Aufbau von Objekten vertraut machen.

2.2 Objektarten - Tausend Möglichkeiten

Auf den folgenden Seiten geht es um die verschiedenen Arten von Objekten, die GEM kennt. Wie und wozu diese benutzt werden, wird in den nächsten Kapiteln beschrieben. Überlegen Sie bei der Arbeit am ST ab und zu, wo welche Objekte zu entdecken sind.

Zunächst muß der Objektaufbau erweitert werden. Es kommen mehrere neue Bestandteile hinzu:

```
Object = RECORD                    struct object
           next,                   { int next;
           head,                     int head;
           tail: INTEGER;            int tail;
           type: INTEGER;            int type;
          ...                       ...
           spec: ADDRESS;            long spec;
           x,y,                      int x,y,
           width,height:INTEGER;         width,height;
         END;                      } OBJECT;
```

Bild 2.2.1: Erweiterter Objektaufbau

Jedes Objekt hat eine bestimmte Größe, die aus den letzten vier Feldern hervorgeht. Nun kann man natürlich schlecht voraussagen, an welcher Position auf dem Bildschirm ein Objekt erscheinen wird. Deshalb enthalten *x* und *y* auch keine absoluten Werte, sondern relative. Sie beziehen sich auf die Position des Parents des jeweiligen Objekts.

Die *x*- und *y*-Werte des Wurzelobjekts sind logischerweise relativ zu dem Punkt, an dem der Baum gezeichnet werden soll. Wird die Position eines Wurzelobjekts erst später im Programm festgelegt, empfiehlt es sich, hier einfach eine Null einzusetzen.

Stellen Sie sich vor, eine 100 Pixel breite und 50 Pixel hohe Box liegt auf den Schirmkoordinaten 50,50 und in der Mitte des Rechtecks soll ein Button dargestellt werden: Für einen 40 mal 30 Pixel großen Button müßten die *x*- und *y*-Komponenten des Buttons 30 und 10 betragen.

Auf dem Bildschirm läge dieser Button bei den Koordinaten 50+30=80 und 50+10=60. Würde die Box an einer anderen Position gezeichnet, bliebe der Button trotzdem genau in deren Mitte.

width und *height* enthalten jeweils die Breite und Höhe des Objekts - ebenfalls in Pixeln. Im obigen Beispiel müßte hier für den Button 40 und 30 stehen.

type enthält eine Kennzahl, die den Typ des Objekts festlegt. In *spec* steht ein Zeiger (diesmal ein richtiger Zeiger - also eine Adresse) auf zusätzliche Daten, die je nach Objekttyp benötigt werden.

Die einzelnen Typen und ihre Kennziffern:

GraphicBox	(G_BOX)	20
GraphicText	(G_TEXT)	21
GraphicBoxText	(G_BOXTEXT)	22
GraphicImage	(G_IMAGE)	23
GraphicProgDef	(G_PROGDEF)	24
GraphicInvisibleBox	(G_IBOX)	25
GraphicButton	(G_BUTTON)	26
GraphicBoxChar	(G_BOXCHAR)	27
GraphicString	(G_STRING)	28
GraphicFormattedText	(G_FTEXT)	29
GraphicFormattedBoxText	(G_FBOXTEXT)	30
GraphicIcon	(G_ICON)	31
GraphicTitle	(G_TITLE)	32

Bild 2.2.2: Kennzahlen der Objekttypen

In Klammern finden sich die Namen, die beim Entwicklungspaket und in den normalen C-Bibliotheken verwendet werden. Ich verwende hier die Namen aus dem Modula-2 System, da sie weniger kryptisch und damit leichter verständlich sind.

Boxen

Die erste Gruppe von Objekttypen sind die Box-Typen. Dabei handelt es sich um einfache Rechtecke, wie sie als Hintergrund für Dialogboxen verwendet werden.

Der erste Typ ist *GraphicBox*. Dies ist der einfachste Typ und stellt nur ein Rechteck dar. Der Kasten kann jedoch verschiedene Variationen erfahren. So darf er beispielsweise einen Rahmen haben, mit einem Muster gefüllt sein oder verschiedene Farben annehmen.

Die Informationen über die Erscheinungsform des Kastens stehen in dem Feld *spec*, das hier nicht als Zeiger, sondern als ein 32-Bit-Wert benutzt wird.

Im *spec*-Feld bedeuten die einzelnen Bits:

Bits	Bedeutung
0–3	Fuellfarbe
4–6	Fuellmuster
7–11	unbenutzt
12–15	Rahmenfarbe
16–23	Rahmendicke
24–32	unbenutzt

Bild 2.2.3: Das *spec*-Feld bei GraphicBox-Typen

Das Füllmuster entspricht einem der Standards:

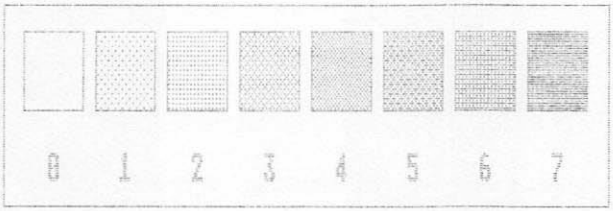


Bild 2.2.4: Die Codes für die Füllmuster

Die Farbe eines Musters steht in den Bits 0-3. Dabei werden die 16 verfügbaren Farben benutzt. Für monochrome Bildschirme sind nur die Farben 0 und 1 interessant.

Das Rechteck kann einen Rahmen haben. Die Stärke des Rahmens wird in den Bits 16-32 angegeben. 0 bedeutet keinen Rahmen. Negative Werte ergeben einen Rahmen "um" die Box, das heißt, der Rahmen wird außen "angeklebt". Bei positiven Werten zeichnet GEM den Rahmen nach "innen", wodurch sich der verbleibende Platz in der Box verkleinert.

Die Farbe dieses Rahmens kann extra festgelegt werden - dies geschieht mit den Bits 12 bis 15.

GraphicBox wird hauptsächlich als Hintergrund für weitere Objekte benutzt. Es steht also meistens in den höheren Ebenen des Objektbaums.

Der nächste Typ ist eine Erweiterung von *GraphicBox*. Bei *GraphicBoxChar* kann zusätzlich ein Zeichen angegeben werden, das dann in der Mitte der Box erscheint.

Dazu muß die Information in *spec* erweitert werden:

Bits	Bedeutung
0 – 3	Fuellfarbe
4 – 6	Fuellmuster
7	Transparent
8 – 11	Textfarbe
12 – 15	Rahmenfarbe
16 – 23	Rahmendicke
24 – 32	Zeichen

Bild 2.2.5: Das "spec"-Feld bei GraphicBoxChar-Typen

Zunächst kommt Bit 7 hinzu. Ist es 0, so wird das Zeichen "transparent" in die Box geschrieben, damit der Hintergrund durchscheint. Steht hier eine 1, wird das Zeichenfeld vorher gelöscht, und das Zeichen erscheint auf weißem Hintergrund.

Das Zeichen steht im obersten Byte von *spec*. Erlaubt sind alle Werte; die acht Bit ermöglichen vollen Zugriff auf den gesamten Zeichensatz.

In welcher Farbe das Zeichen erscheint, wird in den Bits 8 bis 11 bestimmt, womit man wieder Zugriff auf alle sechzehn Farben hat.

Gerade der Typ *GraphicBoxChar* wird in GEM-Programmen relativ häufig verwendet. Bei Fenstern haben die Felder zum Schließen oder Verändern der Größe diesen Typ. Beliebte Zeichen sind auch die Pfeile, die im Resource-Construction-Set aus dem Entwicklungspaket zum Weiterschalten der Farbauswahl dienen.

Der letzte Typ dieser Gruppe ist *GraphicInvisibleBox*. Hierbei wird eigentlich nur ein Rand gezeichnet, der über *spec* auch abgeschaltet werden kann.

Dieser Typ dient dazu, weitere Objekte zusammenzufassen. Es handelt sich um ein Parent im Objektbaum, das nicht auf dem Bildschirm erscheint. Wozu dies gut ist, wird deutlich, wenn Sie im Kapitel 9.2 die *Radio Buttons* näher kennenlernen.

Strings

Um einfache Texte darzustellen, gibt es drei weitere Objekttypen. Da ist zunächst *GraphicString*. Damit lassen sich Zeichenketten mit normalem Zeichensatz in schwarz auf den Bildschirm bringen.

spec ist ein Zeiger auf eine Zeichenkette, die mit einem Null-Zeichen abgeschlossen wird. Es muß hier also die Adresse des Strings stehen.

Der Typ *GraphicTitle* wird für die Titel der Drop-Down-Menüs verwendet. Man braucht ihn, um eigene Menüs zu konstruieren.

GraphicButton ist der Typ der Buttons. Eigentlich handelt es sich nur um ein Textobjekt, das zentriert in einem Rahmen dargestellt wird. Ein einfaches Objekt *GraphicButton* hat einen Rahmen, der genau ein Pixel dick ist. Die Rahmenstärke ist noch von anderen Flags abhängig, auf die ich in Kapitel 2.3 eingehen werde.

spec zeigt auf den Text des Buttons.

Texte

Bei den Text-Objekten handelt es sich ebenfalls um Zeichenketten. Diese können jedoch - im Unterschied zum Typ *GraphicString* - variiert werden.

Der einfachste Typ ist *GraphicText*. *spec* zeigt, wie bei allen Text-Typen, auf das sogenannte *TEdinfo*. Dies ist ein Record, in dem die folgenden Informationen enthalten sind:

```
TEdinfo = RECORD                        struct text_edinfo
            ptext    :ADDRESS;            { long  te_ptext;
            ptmpl    :ADDRESS;              long  te_ptmlt;
            pvalid   :ADDRESS;              long  te_pvalid;
            font     :INTEGER;              int   te_font;
            resvd1   :INTEGER;              int   te_junk1;
            just     :INTEGER;              int   te_just;
            colour   :INTEGER;              int   te_color;
            resvd2   :INTEGER;              int   te_junk2;
            thickness:INTEGER;              int   te_thickness;
            txtlen   :INTEGER;              int   te_txtlen;
            tmplen   :INTEGER               int   te_tmplen
          END;                            } TEDINFO;
```

Bild 2.2.6: Der TEdinfo-Aufbau

ptext ist ein Zeiger auf die Zeichenkette, die dargestellt werden soll. Der String wird wieder durch ein Null-Zeichen abgeschlossen. Zusätzlich ist seine Länge in *txtlen* abgelegt, wobei das abschließende Null-Zeichen mitgezählt wird. *txtlen* muß also um eins größer sein als der Text.

Man kann wählen, ob der Text groß (normal) oder klein (so wie der Text unter Icons) dargestellt werden soll. Für große Schrift muß in *font* eine 3 (Kennwert "IBM") stehen, ansonsten eine 5 (Kennwert "SMALL").

Der Text kann (im Objekt in *width* und *height* festgelegt) verschieden positioniert werden, dies wird in *just* definiert. Dabei gilt: 0 für links (TE_LEFT), 1 für rechts (TE_RIGHT) und 2 für zentriert (TE_CNTR).

GraphicBoxText: Hierbei handelt es sich um einen Text, der einen Hintergrund und einen Rahmen hat.

Dazu werden in *colour*, ähnlich wie bei *GraphicBox*, in verschiedenen Bit-Gruppen die nötigen Informationen geliefert.

Die Aufteilung lautet:

Bits	Bedeutung
0 – 3	Fuellfarbe
4 – 6	Fuellmuster
7 – 11	unbenutzt
12 – 15	Rahmenfarbe
16 – 23	Rahmendicke
24 – 32	unbenutzt

Bild 2.2.7: Das *colour*-Feld beim TEdinfo

Die Felder haben die gleichen Bedeutungen wie bei *GraphicBox*. Die Stärke des Rahmens wird in *thickness* angelegt. 0 bedeutet keinen Rahmen, negative Zahlen Ränder nach außen und positive Zahlen kennzeichnen Ränder nach innen.

Man kann mit diesem Typ farbige Buttons simulieren. Die Titelzeilen der Fenster sind nichts anderes als *GraphicBoxText*-Objekte mit einem bestimmten Hintergrund, einem Rahmen von einem Punkt Dicke und einem zentriertem Text.

GraphicFormattedText erlaubt dem Benutzer, einen String zu editieren. Auf die entsprechen GEM-Funktionen komme ich im Kapitel 3.4.

ptmplt (im TedInfo) zeigt auf einen Maskenstring, genannt "template". Darin sind mit dem Unterstrich ("_") versehene Platzhalter enthalten, in die der in *ptext* definierte String eingefügt wird.

Aus dem Template "__/__/__" und dem *ptext* "160886" wird "16/08/86" gemischt und dargestellt. Alle Zeichen, die im Template "_" stehen, können vom Benutzer editiert werden. Unser Beispiel entspricht der Datumseingabe im Kontrollfeld.

tmplen gibt die Länge des Templates an (wegen des abschließenden Null-Bytes wieder um eins erhöht).

In *pvalid* wird angegeben, welche Zeichen an den editierbaren Stellen akzeptiert werden.

Dabei hat man folgende Möglichkeiten:

Zeichen	Bedeutung
"9"	Alle Ziffern von 0 bis 9
"A"	Alle Großbuchstaben von "A" bis "Z" und Leerzeichen (keine Umlaute!)
"a"	Alle Groß- und Kleinbuchstaben sowie Leerzeichen (keine Umlaute!)
"N"	Alle Ziffern, alle Großbuchstaben und Leerzeichen (keine Umlaute!)
"n"	Alle Ziffern, Groß- und Kleinbuchstaben, Leerzeichen (keine Umlaute!)
"F"	Alle Zeichen zur Angabe eines File-Namens (auch Umlaute!) und "?", "*"
"p"	Alle Zeichen zur Angabe eines File-Namens mit Pfadnamen, "?", "*", "\", ":"
"P"	Alle Zeichen zur Angabe eines File-Namens mit Pfadnamen, "\", ":"
"X"	Alle Zeichen

Bild 2.2.8: Die Codes für *pvalid*

Für die Datumseingabe ergäbe sich ein *pvalid* von "999999". Das Zeichen "@" hat bei allen drei Strings eine Sonderfunktion. Tritt es in einer Zeichenkette auf, so wird angenommen, daß alle nachfolgenden Zeichen Leerzeichen sind. Man braucht einen solchen String also nicht zu löschen, sondern es reicht, wenn das erste Zeichen ein "@" ist.

GraphicFormattedBoxText ist einfach ein *GraphicFormattedText* mit Rahmen. Dadurch erhält auch das Feld *thickness* wieder Bedeutung. Die Handhabung dieser Objekte wird später näher erläutert.

Bilder

Häufig werden Bilder (Icons) verwendet, deren Funktion auf einen Blick klar wird. Es gibt hier zwei unterschiedliche Objekttypen.

GraphicImage ist ein Icon ohne Text. *spec* zeigt auf die Definition dieses Icons und benötigt dafür einen neuen Record:

```
BitBlk = RECORD                    struc bitblk
           pdata :ADDRESS;           { long  pdata;
           width :INTEGER;             int   width;
           height:INTEGER;             int   height;
           x     :INTEGER;             int   x;
           y     :INTEGER;             int   y;
           colour:INTEGER              int   color;
         END;                        } BITBLK;
```

Bild 2.2.9: Der BitBlk-Aufbau

pdata zeigt auf die Icon-Definition, in der die einzelnen Punkte bitweise dargestellt werden. *width* und *height* geben Breite und Höhe des Icons an. *width* muß eine gerade Zahl sein.

x und *y* erlauben einen Offset im Bit-Muster des Icons. *colour* enthält Angaben über die Farbe des Icons.

spec zeigt auf den sogenannten *IconBlk*-Record:

```
IconBlk = RECORD                   struct iconblk
            pmask :ADDRESS;          { long  pmask;
            pdata :ADDRESS;            long  pdata;
            ptext :ADDRESS;            long  ptext;
            iChar :INTEGER;            int   ichar;
            xChar :INTEGER;            int   xchar;
            yChar :INTEGER;            int   ychar;
            x     :INTEGER;            int   x;
            y     :INTEGER;            int   y;
            width :INTEGER;            int   width;
            height:INTEGER;            int   height;
            xText :INTEGER;            int   xtext;
            yText :INTEGER;            int   ytext;
            wText :INTEGER;            int   wtext;
            hText :INTEGER;            int   htext;
          END;                       } ICONBLK;
```

Bild 2.2.10: Der IconBlk-Aufbau

GraphicImages sind immer einfarbig. Die Punkte werden in der angegebenen Farbe gezeichnet, während der Hintergrund schwarz bleibt. Die Icons in den Alertboxen sind solche Objekte.

Bei den *GraphicIcons* sind verschiedene Farben, Masken sowie ein Text und zusätzlich ein einzelner Buchstabe möglich.Der erste Zeiger ist der Maskenzeiger. Er deutet auf ein Bit-Feld von der Größe des Icons, in dem angegeben wird, welche Punkte erscheinen und welche nicht. Steht an einer Stelle eine 0, so wird der entsprechende Punkt im Bild ausmaskiert.

Diese Technik läßt sich am Papierkorb des Desktops verdeutlichen. Wird er angeklickt, so erscheint wirklich nur der Papierkorb invertiert, nicht etwa ein Rechteck. Das heißt, die "überstehenden" Punkte wurden ausmaskiert.

pdata zeigt auf die eigentliche Definition des Icons. In *x*, *y*, *width* und *height* werden die Position sowie Breite und Höhe des Icons (und damit auch seiner Maske) festgehalten. *width* muß durch 8 ohne Rest teilbar sein.

ptext zeigt auf einen mit Null abgeschlossenen String, der zusätzlich dargestellt wird (wie die Laufwerksnamen, z.B. "HARDDISK" auf dem Desktop). Wo der Text relativ zur linken oberen Ecke im Icon erscheinen soll und wieviel Platz er einnimmt steht in *xText*, *yText*, *wText* und *hText*.

Schließlich läßt sich noch ein einzelner Buchstabe im Icon darstellen. Dies wird auf dem Desktop bei der Laufwerkskennung verwendet. Die Position dieses Zeichens steht in *xChar* und *yChar*.

Das Zeichen selbst steht zusammen mit den Informationen über Vorder- und Hintergrundfarbe in *iChar*. Die 16 Bit werden wie folgt aufgeteilt:

Bits	Bedeutung
0 – 3	Fuellfarbe
4 – 6	Fuellmuster
7 – 11	unbenutzt
12 – 15	Rahmenfarbe

Bild 2.2.11: Das "colour"-Feld beim TEDinfo

Dabei zählen alle Punkte, in deren Icon-Definition eine 1 stand als Vordergrund, die anderen (die bei *GraphicImage* schwarz erscheinen würden) als Hintergrund.

Eigene Objektroutinen

Der letzte Objekttyp ist *GraphicProgDef*. Hier zeichnet nicht das GEM, sondern ein eigenes Programm. *spec* zeigt dabei auf den ApplBlk-Record:

```
ApplBlk = RECORD                struc applblk
            code :ADDRESS;        { long  code;
            param:LONGINT;          long  param;
          END;                    } APPLBLK;
```

Bild 2.2.12: Der ApplBlk-Aufbau

code zeigt auf den Beginn der Zeichenroutine. *param* ist ein 32-Bit-Wert, der einen oder mehrere Parameter (oder einen Zeiger auf einen Parameterblock) enthalten kann.

Hiermit lassen sich recht elegante Lösungen für Probleme finden, die mit den normalen Objekttypen nicht zu bewältigen sind. So bieten die vielen Paint- und Draw-Programme Zugriff auf alle verfügbaren Muster.

Normalen Objekttypen, wie zum Beispiel *GraphicBox*, können nur 7 Muster darstellen. Um die Musterauswahl auch über Resources zu ermöglichen, ist eine Routine zu schreiben, die einfach ein gemustertes Rechteck ausgibt. Als Parameter werden Index und Style des betreffenden Musters verwendet und schon können auch diese Auswahlfelder als Objekte dargestellt werden.

Denkbar sind auch dreieckige Dialogboxen oder Boxen, die Töne von sich geben. Die Möglichkeiten sind beinahe unbegrenzt!

Man kann *GraphicProcDef* übrigens nicht mit den Resource-Editoren erzeugen, da sie einfach nicht die Möglichkeit bieten. Welche Umstände sich daraus ergeben, wird in den nächsten Kapiteln deutlich.

2.3 Flags und Status

Der Objekt-Record ist noch nicht komplett. In diesem Kapitel werden Sie die fehlenden zwei Felder kennenlernen. Zunächst jedoch die endgültige Definition:

```
Object = RECORD                        struct object
           next,                       { int next;
           head,                         int head;
           tail   :INTEGER;              int tail;
           type   :INTEGER;              int type;
           flags  :INTEGER;              int flags;
           status:INTEGER;               int status;
           spec   :ADDRESS;              long spec;
           x,y,                          int x,y,
           width,height:INTEGER;             width,height;
         END;                          } OBJECT;
```

Bild 2.3.1: Endgültiger Objektaufbau

Mit dem *status*-Feld verleihen Sie den Objekten Attribute. Das Feld wird als Bit-Vektor aufgefaßt, so daß mehrere Optionen gültig sein können. Der benötigte Wert ist durch bitweises Odern oder einfaches Addieren der Kennwerte zu ermitteln:

Attribut	Hex	Dez
Normal	00H	1
Selected	01H	2
Crossed	02H	4
Disabled	04H	8
Outlined	08H	16
Shadowed	10H	32

Bild 2.3.2: Die Codes für das *status*-Feld

Bei *Normal* wird ein Objekt wie oben beschrieben dargestellt.

Hat das Objekt den Status *Selected*, so erscheint es invertiert auf dem Bildschirm. Im Dialog wird das entsprechende Bit durch Anklicken gesetzt.

Bild 2.3.3: Selected

Crossed bewirkt, daß GEM über das Objekt ein Querkreuz legt. Das Objekt erscheint durchgestrichen. Diese Option wird meines Wissens in kaum einem Programm verwendet.

Bild 2.3.4: Crossed

Checked findet hauptsächlich bei Menüeinträgen Verwendung, mit denen eine Option ein- oder ausgeschaltet wird. Es erscheint dann ein kleines Häkchen vor dem Text.

Bild 2.3.5: Checked

Disabled Objekte können normalerweise nicht angewählt werden. Sie werden grau dargestellt und man findet sie zum Beispiel bei den Trennstrichen in Menüs. Es läßt sich hiermit aber auch darstellen, daß eine Option nicht gesetzt ist. Dies wird im Kontrollfeld bei den beiden Icons zum Schalten der Tastaturklicks und des Warntons verwendet.

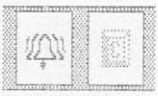


Bild 2.3.6: Disabled

Outlined zeichnet um ein Objekt einen zusätzlichen Rahmen.

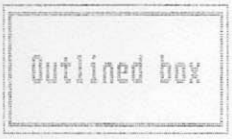


Bild 2.3.7: Outlined

Shadowed bewirkt einen Effekt, der das Objekt scheinbar einen kleinen Schatten auf den Hintergrund werfen läßt.

Bild 2.3.8: Shadowed

Attribut	Hex	Dez
None	000H	0
Selectable	001H	1
Default	002H	2
Exit	004H	4
Editable	008H	8
RadioButton	010H	16
LastObject	020H	32
TouchExit	040H	64
HideTree	080H	128
Indirect	100H	256

Bild 2.3.9: Die Codes für das *flags*-Feld

Die restlichen Bits in *status* werden nicht verwendet. Es lassen sich hier jedoch Informationen für eigene Objekt-Typen unterbringen.

flags ist ebenfalls ein Bit-Vektor. Allerdings enthält er Angaben, die den Ablauf eines Dialogs beeinflussen. Welcher Wert hier steht, oder stehen soll, wird wiederum durch bitweises Odern oder Addieren der Kennwerte berechnet.

Selectable bedeutet, daß das betreffende Objekt in einem Dialog angewählt werden kann. Dies betrifft alle aktiven Elemente wie Buttons oder Icons. Dagegen ist es natürlich wenig sinnvoll, den Hintergrund anwählen zu können oder ein Image, das nur zur Gestaltung einer Dialogbox dient. Buttons, die *Selectable* sind, erhalten automatisch einen Rahmen von zwei Punkten Dicke.

Ist ein Objekt *Default*, wird es beim Drücken der <Return>-Taste automatisch ausgewählt und der Dialog verlassen. GEM kennzeichnet ein Default-Objekt immer mit einem Rahmen von drei Punkten Dicke. So ist in der Desktop-Box die Option "ABBRUCH" am dicken Rahmen als Default-Button zu erkennen.

Hat ein Objekt das *Exit*-Flag, wird ein Dialog bei dessen Auswahl beendet. Zwangsläufig muß in jeder Box mindestens ein Objekt dieses Flag besitzen.

Objekte, bei denen das *Editable*-Flag gesetzt ist, können im Dialog auch über die Cursor-Tasten ausgewählt und editiert werden. In der Standardbox zum Anwählen eines File-Namens betrifft dies die beiden Objekte, in denen der Pfad und der Dateiname stehen. Bei dem ausgewählten Objekt erscheint ein kleiner Cursor in Form eines senkrechten Strichs.

Die *RadioButtons* sollen hier nur kurz angesprochen werden, da sie in Kapitel 9.2 detaillierter erläutert werden. Es handelt sich bei ihnen um eine Reihe von Objekten mit demselben Parent, von denen immer nur eines ausgewählt sein kann. Denken Sie dabei beispielsweise an die Einstellung der Baudrate für die serielle Schnittstelle. Das Anwählen eines Objekts verursacht automatisch das "Abwählen" des zuvor angewählten Objekts.

LastObject zeigt an, daß ein Objekt das letzte im jeweiligen Objektfeld ist. Es gibt immer nur ein Objekt, das dieses Flag besitzt.

TouchExit ist eine Abwandlung von *Exit*. Bei *Exit* wird der Dialog erst beendet, wenn der Benutzer den Mausknopf losgelassen hat. Bei *TouchExit* reicht das Drücken des Knopfes.

HideTree bewirkt, daß ein Unterbaum unsichtbar wird, und beim Zeichnen nicht mehr auf dem Bildschirm erscheint. Dies betrifft alle untergeordneten Objekte. Man kann damit leicht einen Teil eines Dialoges ausblenden und somit Speicherplatz sparen.

Bei *Indirect* zeigt *spec* auf den Wert von *spec*. Er wird also indirekt abgelegt.

Damit ist die Struktur der Objekte und der Objektbäume soweit geklärt, daß Sie sich anschließend den Routinen des Object-Managers zuwenden können.

3. Der Objekt-Manager

Mein Meister, mein Meister und siehst du nicht dort,
Aus dem Akku schleicht sich das Vorzeichen fort!
Nur ruhig, nur ruhig, mein liebes Kind,
Ich hole es im Overflow, ganz bestimmt.

Ballade vom Bitkönig

Der Objekt-Manager ist eine Sammlung von AES-Routinen, die sich mit Objekten befassen. Sie sind aufgrund ihrer logischen Verwandtschaft zusammengefaßt. In den Kapiteln 4 bis 7 werden Sie noch einige andere "Manager" kennenlernen, denn mit einem einzigen Manager-Paket läßt sich noch nicht viel anfangen. Es werden immer mehrere benötigt.

Im TDI-Modula-2 System sind die Manager jeweils in eigenen Modulen zusammengefaßt, die bei Bedarf elegant und übersichtlich importiert werden. Bei C muß man in den großen Topf der AES-Bibliotheken hineingreifen und sich die erforderlichen Routinen einzeln herausfischen.

Die angegebenen C-Routinen sind meist Funktionen. Der Rückgabewert zeigt an, ob ein Fehler aufgetreten ist (0) oder nicht (andere Zahl). Die Modula-2 Routinen verzichten auf diesen Wert, da beim Zeichnen auf den Bildschirm kaum ein Fehler auftreten kann.

3.1 Baumstruktur verändern

Die drei folgenden Routinen zum Verändern der Baumstruktur gehen davon aus, daß sich bereits ein Baum im Speicher befindet und die Indizes der Objekte sowie die Adresse des Baumes bekannt sind. Sie dienen dazu, den Baum umzustrukturieren.

Das Objekt *Child* wird durch die erste Prozedur

ObjectAdd (Tree : ADDRESS; Parent, Child : INTEGER);
ob_adreturn=objc_add(ob_tree, ob_parent, ob_child)

(hier wird der Index, oder besser die Objektnummer übergeben!) zum Kind des Objekts *Parent*. Beide Objekte müssen im Baum *Tree* stehen, dessen Adresse übergeben wird.
GEM hantiert hier nur mit Zeigern. Dazu sollten Sie sich die schon beschriebene Baumstruktur ins Gedächtnis zurückrufen. *tail* von *Parent* deutet auf das letzte Kind. Dessen *next* zeigt auf *Parent* zurück. Nun wird *Child* hinten angehängt. *next* von *tail*-Objekt zeigt nun auf *Child*, ebenso *tail* vom *Parent*. Schließlich muß nur noch *next* von *Child* auf *Parent* gerichtet werden, und der Baum ist wieder korrekt verzeigert und *Child* wurde eingefügt.

Die Routine

ObjectDelete(Tree : ADDRESS; Object : INTEGER)
ob_dereturn=objc_delete(ob_tree, ob_object)

löscht *Object* aus dem Baum. Dazu werden ebenfalls nur die Zeiger manipuliert.
Zunächst muß der Vorgänger von *Object* gesucht werden. Dessen *next* nimmt nun den *next*-Wert von *Object* an, womit *Object* eigentlich schon entfernt ist. Wenn *Object* das letzte Kind war, muß noch berücksichtigt werden, daß die *tail*-Komponente des Parents auf den Vorgänger zu richten ist.

Mit der Routine

ObjectOrder (Tree : ADDRESS; Object, NewPos : INTEGER)
ob_orreturn=(ob_tree, ob_object, ob_newpos)

wird die Position eines Kindes in der Kinderliste seines Parents verändert. Die neue Position steht in *NewPos*. Diese wird relativ zum letzten Kind angegeben. Eine 0 verschiebt das Kind ganz nach hinten in der Liste und umgekehrt.
Dazu dienen wiederum Zeigermanipulationen.

3.2 Objekte zeichnen und wiederfinden

Es folgt eine Routine, die Objekte auf den Bildschirm bringt und deshalb eine der wichtigsten ist:

```
ObjectDraw( Tree : ADDRESS; StartOb, Depth, XClip, YClip,
            WClip, HClip : INTEGER)
ob_drreturn = objc_draw (ob_tree, ob_drstartob, ob_drdepth,
                         ob_drxclip, ob_dryclip,
                         ob_drwclip, ob_drhclip)
```

Tree gibt wieder an, in welchem Baum die Objekte stehen. *StartOb* ist das Objekt, bei dem mit dem Zeichnen begonnen werden soll.

In *Depth* steht, wie "tief" der Baum gezeichnet werden soll, d.h. wieviele Ebenen berücksichtigt werden. Bei 0 wird nur das Startobjekt gezeichnet. Mit einer 1 kommen auch dessen Kinder zum Zug, mit einer 2 auch die "Enkel". Will man den ganzen Baum (oder Unterbaum) ausgeben, so kann hier auch eine sehr hohe Zahl stehen.

Mit den letzten vier Parametern wird das Clipping-Rechteck festgelegt. GEM zeichnet nur innerhalb dieses Rechtecks. Alle "überstehenden" Objekte oder Teile werden abgeschnitten (engl. "clip"). *XClip* und *YClip* geben die linke obere Ecke und *WClip*, *HClip* die Breite und Höhe dieses Rechtecks an. Alle Werte werden als direkte Bildschirmkoordinaten eingetragen.

Will man kein Clipping, was aber selten der Fall sein sollte, muß die gesamte Bildschirmfläche angegeben werden.

Oftmals ist es nötig zu wissen, an welcher Bildschirmstelle sich ein Objekt befindet. Da die Koordinaten im Baum immer relativ zum Parent sind, müßte man dazu selbst durch den Baum wandern und mitrechnen. Dafür gibt es aber die Prozedur

```
ObjectOffset( Tree : ADDRESS; Object : INTEGER;
              VAR Xoff, Yoff :INTEGER)
ob_ofreturn = objc_offset (ob_oftree, ob_ofobject,
                           &ob_ofxoff, &ob_ofyoff)
```

In *Tree* und *Object* stehen wie gewohnt die Angaben über den Baum und das betreffende Objekt. *Xoff* und *Yoff* enthalten nach Ausführung die X- und Y-Koordinaten des Objekts in absoluten Pixel-Werten.

Schließlich möchte man vielleicht wissen, ob sich an einem bestimmten Punkt auf dem Bildschirm ein Objekt befindet. Dazu gibt es die Funktion

```
ObjectFind ( Tree : ADDRESS; StartOb, Depth,
             MX, MY : INTEGER) : INTEGER
ob_fobum = objc_find (ob_ftree, ob_fstartob, of_fdepth,
                      ob_fmx, ob_fmy)
```

Der Baum *Tree* wird ab dem Objekt *StartOb* bis zur Tiefe *Depth* abgesucht. Liegt der Punkt *MX*, *MY* "innerhalb" eines Objekts, so gibt die Routine dessen Nummer als Ergebnis zurück. Ist an dieser Stelle kein Objekt vorhanden, so erhält man eine -1.

Je tiefer die Routine sucht, um so genauer wird ein Objekt gefunden. Sucht man nur 2 Ebenen tief, so könnte man die Nummer eines Objekts erhalten, das nur als Hintergrund für weitere dient.

Indem Sie mit dieser Routine die Mauskoordinaten übergeben, können Sie leicht feststellen, ob der Benutzer ein Icon angeklickt hat.

3.3 Objekte verändern

Da man bei einem Baum, der im Speicher liegt, nur auf Umwegen an die Objekte herankommt, steht zur Statusveränderung die folgende Routine zur Verfügung:

```
ObjectChange ( Tree : ADDRESS; Object, resrvd,
               XClip, YClip, WClip, HClip,
               NewState, Redraw : INTEGER)
ob_creturn = objc_change (ob_ctree, ob_cobject, ob_cresrvd,
                ob_cxclip, ob_cyclip, ob_cwclip, ob_chclip,
                ob_cnewstate, ob_credraw)
```

NewState enthält den neuen Wert für die Status-Komponente. *Redraw* bestimmt, ob das Objekt neu gezeichnet werden soll. Dazu muß eine 1 übergeben werden, sonst eine 0.

Die Parameter *resrvd* bis *HClip* entsprechen den Parametern *Depth* bis *HClip* vom *ObjectDraw*-Aufruf. GEM-kann diese unverändert weiterverwenden. In *resrvd* sollte eine 0 stehen, da im Normalfall nur das veränderte Objekt neu gezeichnet werden muß.

Sie können aber auch noch auf eine andere Art Werte in der Resource verändern. Wie schon bemerkt, kann der Resource-Baum auch als ein Feld aufgefaßt werden. Die Objektnummern entsprechen den Indizes in diesem Feld. Haben Sie nun im Programm einen Zeiger auf ein Feld von Objekten, so können Sie darüber auch alle Einträge eines Objekts direkt ansprechen. Diese Methode ist schneller als *ObjectChange*, und wird auch in den Beispielprogrammen benutzt.

3.4 Objekte editieren

Die letzte Routine des Objekt-Managers erlaubt dem Benutzer, Texte der Objekttypen *GraphicText* und *GraphicBoxText* zu editieren:

```
ObjectEdit ( Tree : ADDRESS; Object, Char, IdX, Kind : INTEGER;
             VAR NewIdX : INTEGER)
ob_edreturn = ibjc_edit (ob_edtree, ob_edobject, ob_edchar,
                         ob_edidx, ob_edkind,
                         &ob_ednewidx)
```

Dahinter verbergen sich vier Funktionen, die über *Kind* angewählt werden. Die erste, *EditStart*, ist für künftige Erweiterungen von GEM reserviert und hat den Kennwert 0.

Die zweite (Kennwert 1) heißt *EditInit*. Durch sie werden die Strings *ptext* und *ptmplt* gemischt. Gleichzeitig erscheint der Cursor in Form eines senkrechten Strichs.

Mit *EditChar* (Kennwert 2) kann der Benutzer Text eingeben. Die in *pvalid* gegebenen Beschränkungen werden vom GEM übernommen.

Tree und *Object* bezeichnen den Baum und Index des betreffenden Objekts. *Char* enthält die Zeicheneingabe des Benutzers. *IdX* gibt an, ab welcher Stelle der Text editiert werden soll; der Wert ist also ein Index im Eingabe-String. Soll der Cursor am Anfang des Eingabetextes erscheinen, muß hier eine 0 stehen.

Entsprechend dazu enthält *NewIdX* nach der Editierung den Index im Eingabestring, bei dem der Cursor zuletzt stand.

Schließlich muß die Editierung durch einen Aufruf mit dem Parameter *Kind* mit dem Wert 3 beendet werden (*EditEnd*). Dies bewirkt, daß der Cursor wieder verschwindet. *ObjectEdit* muß also dreimal hintereinander mit *Kind* von 1 bis 3 aufgerufen werden.

Man wird normalerweise die Objekte nicht einzeln editieren lassen, sondern diese Aufgabe dem Form-Manager (genauer der Routine *FormDo*) aus Kapitel 5 überlassen. Trotzdem folgt noch ein kleines Beispiel, in dem viele Routinen des Objekt-Managers benutzt werden. (In dem entsprechenden Beispielprogramm gehe ich etwas anders vor.)

Betrachten Sie dazu das Kontrollfeld, speziell die Eingabe des Datums und das Ein- und Ausschalten des Tastaturklicks. Die Vorgänge sind hier leicht vereinfacht dargestellt, was aber nicht stört.

Zunächst muß ein Objektbaum für das Kontrollfeld im Speicher stehen. Dieses ist in "CONTROL.ACC" enthalten. Wählt nun der Benutzer das Accessory über die Menüleiste aus, wird zunächst ein Fenster geöffnet, in dem das Feld Platz findet. Wie dies funktioniert, soll hier nicht näher behandelt werden.
Nun müssen noch einige Informationen geholt werden, wie das Datum und die Uhrzeit, die Farbeinstellungen sowie auch die Einstellung von Tastaturklick und Warnton.

Das Datum und die Uhrzeit werden in die entsprechenden TedInfo-Records in die Felder *ptext* gepackt. *ptmplt* und *pvalid* bleiben immer gleich.

Je nach Einstellung des Tastaturklicks wird für das betreffende Icon-Objekt (die Taste mit einem "C") in *Status* das Flag *Disabled* gesetzt oder gelöscht. Dies geschieht mit *ObjectChange*.

Im Fenster kann nun das Kontrollfeld dargestellt werden. Dazu wird *ObjectDraw* aufgerufen. Die nötigen Koordinaten ergeben sich aus der Lage des Fensters.

Das Programm wartet nun auf einen Maus-Klick. Erfolgt dieser, so läßt sich mit *ObjectFind* anhand der Mauskoordinaten herausfinden, welcher Teil des Kontrollfeldes ausgewählt wurde.

Wählt der Benutzer das Tastaturklick-Icon, dreht das Programm das *Disabled*-Flag mit *ObjectChange* um und vermerkt dies intern. Dabei ist *Redraw* auf 1 gesetzt, denn das Icon muß ja neu gezeichnet werden. Danach wird wieder auf einen Maus-Klick gewartet.

Ist das Datumsfeld gewählt, so wird zunächst das *Selected*-Flag mit *ObjectChange* und eingeschaltetem *Redraw* gesetzt. Das Feld erscheint anschließend invers.

Das Programm ruft nun dreimal *ObjectEdit* auf und der Benutzer kann das neue Datum eingeben. *Selected* wird gelöscht und das Datum intern vermerkt.

Alle anderen Aktionen auf dem Kontrollfeld laufen analog ab. Sollte zwischenzeitlich ein anderes Fenster nach oben geholt und danach wiederum das Kontrollfeld gewählt worden sein, so könnte unter Umständen die Hälfte des Feldes überschrieben sein. In diesem Fall kann ein *ObjectDraw* mit entsprechend gesetztem Clipping-Rechteck die überschriebene Hälfte schnell wiederherstellen.

Bisher habe ich beschrieben, wie die Objektbäume aussehen, und wie die Routinen des Objekt-Managers mit ihnen arbeiten. Wie werden die Bäume nun aber erstellt? Diese Frage ist die Überleitung zu den Dateien mit Objektbäumen: den ".RSC"-Files.

4. Der Resource-Manager

Oh Meister, oh Meister, hörst du das Singen?
Es sind die Spikes, die auf dem Adressbus schwingen.
Mein Kind, mein Kind, höre mir zu,
Ich löte Kondensatoren, dann haben wir Ruh!

Ballade vom Bitkönig

Jedes "ordentliche" GEM-Programm hat ein Resource-File, das standardmäßig mit der Endung ".RSC" versehen ist. Darin sind alle Objekte enthalten, die das Programm benutzt.

Diese Auslagerung hat vor allem den Vorteil, daß die Resources unabhängig vom Programm geändert werden können. Die Übersetzung in eine andere Sprache ist so ohne Neukompilierung des eigentlichen Programms möglich.

Den Zugriff auf die Files regelt der Resource-Manager. Die Erstellung der Dateien geschieht mit Hilfe eines Resource-Construction-Programms.

4.1 Resource-Files laden

Ein Resource-File laden Sie mit der folgenden Routine:

```
ResourceLoad ( VAR FName : ARRAY OF CHAR)
re_lreturn = rsrc_load (re_lpfname)
```

FName ist der vollständige File-Name der Resource-Datei. Sie wird von GEM geladen, das Speicherplatz reserviert und die Daten eventuell noch intern verändert.

Das Funktionsergebnis der C-Routine besagt bei einer positiven Zahl, daß ein Fehler aufgetreten ist. In Modula-2 muß man diesen Wert aus *AESCallResult* vom Modul *GEMAESbase* lesen.

Falls die Funktion fehlschlägt, ist es nicht mehr möglich, mit den Resources zu arbeiten - und das gilt dann in der Regel für das gesamte Programm. Man gibt dann zumeist eine Alertbox aus, da eine Weiterarbeit nicht möglich ist.

Da der Object-Manager die Adresse des Objektbaums benötigt, braucht man diese von GEM, denn es ist dem Programm nicht bekannt, wohin GEM das File geladen hat.
Möchte man wissen, welchen Inhalt z.B. das *spec*-Feld von Objekten hat, um etwa eine dazugehöriges TedInfo zu verändern, bietet sich die folgende Routine an:

```
ResourceGetAddr ( Type, Index : INTEGER; VAR Addr : ADDRESS)
re_greturn = rsrc_gaddr (re_gtype, re_index, &re_gaddr)
```

Mit *Type* läßt sich angeben, welche Adresse benötigt wird. Dabei gibt es 17 Codes:

RTree	0	Adresse des Baumanfangs
RObject	1	Adresse eines Objektes
RTedInfo	2	Adresse eines TedInfos
RIconBlock	3	Adresse eines Icon-Blocks
RBitBlock	4	Adresse eines Bit-Blocks
RString	5	Adresse eines Strings
RImageData	6	Adresse eines Graphic-Images
RObSpec	7	Adresse des "spec"-Feldes
RText	8	Adresse von "ptext"
RTemplate	9	Adresse von "ptmplt"
RValid	10	Adresse von "pvalid"
RIconBlkMask	11	Adresse einer Icon-Maske
RIconBlkData	12	Adresse der Icon-Daten
RIconBlkText	13	Adresse des Icon-Textes
RBitBlkData	14	Adresse der Bit-Blockdaten
RFreeString	15	Adresse eines Free-Strings
RFreeImage	16	Adresse eines Free-Images

Bild 4.1.1: Die Codes für ResourceGetAddr

Um herauszufinden, wo der Objektbaum beginnt, muß die Routine mit *RTree* aufgerufen und als *Index* wird eine 1 angegeben werden. Die daraufhin gelieferte Adresse des zuletzt geladenen Baums steht nach der Ausführung dieser Routine in *Addr*.
Das Funktionsergebnis in C ist hier wieder belanglos, wenn die Parameter korrekt sind.

4.2 Resources verändern

Auch die Umkehrung, nämlich das Setzen einer Adresse in der Resource, ist möglich mit

ResourceSetAddr (Type, Index : INTEGER; Addr : ADDRESS)
re_sreturn = rsrc_saddr (re_stype, re_sindex, re_saddr)

Die Codes für *Type* sind die gleichen wie oben.
Die relativen Positionsangaben der Objekte stehen im Resource-File in Zeichen, das heißt, ein Objekt kann etwa einen XY-Offset von 2 und 3 Zeichen zu seinem Parent haben. Je nach eingestellter Bildschirmauflösung ergibt sich daraus eine unterschiedliche Pixel-Positionen. (Bei höchster Auflösung nimmt ein Zeichen 8*16 Pixel ein, bei niedriger nur 8*8!)

Für diese Umrechnung gibt es eine eigene Funktion:

ResourceObjectFix (Tree : ADDRESS; Object : INTEGER)
re_oreturn=rsrc_obfix(re_tree, re_object)

Dabei werden die Offsets des betreffenden Objekts umgerechnet und die Resource an unterschiedliche Bildschirmauflösungen angepaßt. Dies geschieht allerdings auch schon automatisch beim Aufruf von *ResourceLoad*.

4.3 Resources löschen

Da die Speicherverwaltung der Objektbäume intern erfolgt, muß der Speicher durch eine GEM-Routine freigemacht werden. Dazu dient

ResourceFree
re_freturn = rsrc_free()

Danach ist die Resource nicht mehr verfügbar. Es ist aber nicht unbedingt nötig, einen *ResourceFree*-Aufruf durchzuführen. Wird ein Programm beendet, gibt GEMDOS den zuvor vom Programm benutzten Speicher nämlich automatisch wieder frei. Dazu zählt auch der Platz, den die Objektbäume aus dem Resource-File belegt hatten.

4.4 Resources erstellen

Zur Erstellung der Resources wird ein spezielles Programm benötigt: ein Resource-Construction-Programm. Es gibt momentan zwei Programme dieser Art, die weitere Verbreitung gefunden haben.

Da ist zunächst das "RCS" aus dem Entwicklungspaket von Digital Research. Das Programm hat leider einige Schwächen und stürzt bei der Verwendung von Icons gerne ab. Das Resource-Construction-Programm ist das "MMRCP" aus dem Megamax-C-Paket. Das gleiche Programm wird als Lizenzversion auch im Modula-2-Toolkit vertrieben.

Dieses Programm ist erheblich sicherer und auch komfortabler. Außerdem hat es einen integrierten Icon-Editor zu bieten. Es folgt eine kurze Beschreibung der Funktionen des "MMRCP". Glücklicherweise ähnelt es dem "RCS" so weitgehend, daß die Beschreibung für beide gelten kann.

Objekte editieren

Nach dem Start des Programms müssen Sie zunächst eine Resource öffnen. Dies geschieht mit dem *File*-Menü. Eine neue Resource kann mit *New* geöffnet werden.

Daraufhin erscheint ein Fenster, in dem die Objektbäume dargestellt werden. Sie können nun einen Baum aus dem Angebot auf dem Bildschirm mit der Maus in die Resource übernehmen.

Es gibt vier Baum-Arten. *Unknown* ergibt sich, wenn "MMRCP" keine Informationen über die Resource hat. Diese werden in einem zusätzlichen ".DEF"-File gehalten. (Die Modula Lizenzversion benutzt die Endung ".DFN", um Verwechslungen mit den Definitionsdateien für externe Module zu vermeiden.)

Will man vorhandene Resources verändern, so fehlt diese Datei natürlich und "MMRCP" hat weder Informationen über die Art der Bäume noch kennt es die Namen der Bäume oder deren Objekte. Es ist nicht sinnvoll *Unknown*-Bäume zu erstellen.

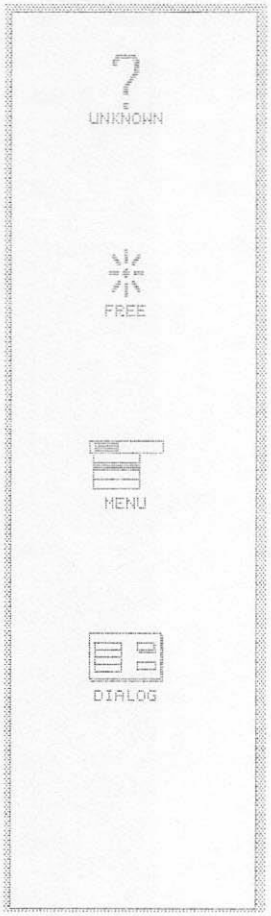
Free ist ein Baum, in dem alle Objekte Pixel-orientierte Positionen haben, das heißt, sie können beliebig bewegt werden. Allerdings sehen die Resources dann in den verschiedenen Auflösungsstufen nicht mehr gleich aus, da mit absoluten Werten gearbeitet wird.

Bei *Menu* und *Dialog* können die Objekte nur zeichenweise bewegt werden. Dies sind je nach Auflösung 8 oder 16 Pixel. In *Menu* werden Menüs gehalten und in *Dialog* die normalen Dialog-Boxen.

Das RCS bietet zusätzlich noch den Typ *Alert* für Alertboxen an. Diese muß man mit dem "MMRCP" im Programm erzeugen.

Die Einteilung in verschiedene Baumarten ist übrigens nur im "MMRCP" aktiv. Es handelt sich um eine logische Gliederung zur Arbeitserleichterung.

Beim Einfügen eines Baums in die Resource erscheint eine Dialog-Box, in der Sie nachträglich die Baumart ändern und dem Baum einen Namen geben können. Es ist sinnvoll, jeden Baum mit einem "sprechenden" Namen zu benennen. Der Name erscheint zusammen mit dem Index des Baums im Include-File.



unbekannte Baumart

pixelorientierter Baum

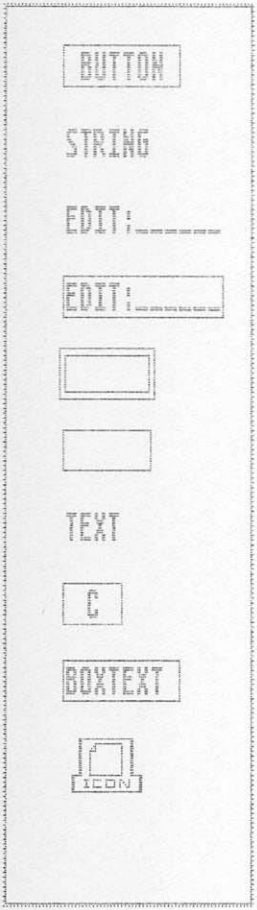
Menübaum

Dialogbaum

Bild 4.4.1: Die Baumarten

Sie öffnen einen Baum durch einen Doppel-Klick auf das jeweilige Baum-Icon in der Resource und haben dann eine Auswahl von Objekten, die in den Baum übernommen werden können. Jedes dieser Objekte kann mit der Maus in den Baum übernommen und positioniert werden. Bei Dialogbäumen ist als Hintergrund automatisch eine *GraphicBox* vorhanden, die zwar verändert aber nicht entfernt werden kann.

Die Größe der Objekte wird verändert, indem Sie in ihre rechte untere Ecke klicken und sie dann mit der Maus "aufziehen". Um die *Flags*- und *Status*-Komponenten sowie ein eventelles *TedInfo* zu verändern, wird das jeweilige Objekt mit einem Doppel-Klick "geöffnet". Das folgende Bild zeigt die Elemente für Dialoge und für *Free*-Bäume.



GraphicButton

GraphicString

GraphicFormattedText

GraphicFormattedBoxText

GraphicInvisibleBox

GraphicBox

GraphicText

GraphicBoxChar

GraphicBoxText

GraphicIcon

Bild 4.4.2: Die Dialog-Partbox

Die Typen der Objekte sind in dem Bild angegeben.
Je nach Objektart erscheinen unterschiedliche Dialog-Boxen, in denen Sie mit der Maus die Einträge in *Flags* und *Status* setzen und löschen können. Außerdem kann Text eingegeben werden.
An den Buttons läßt sich hier auch gut ausprobieren, wie sich die *Selectable-*, *Exit-* und *Default*-Flags auf die Umrandung auswirken.

Das folgende Bild zeigt die Dialogbox für Buttons und Strings:

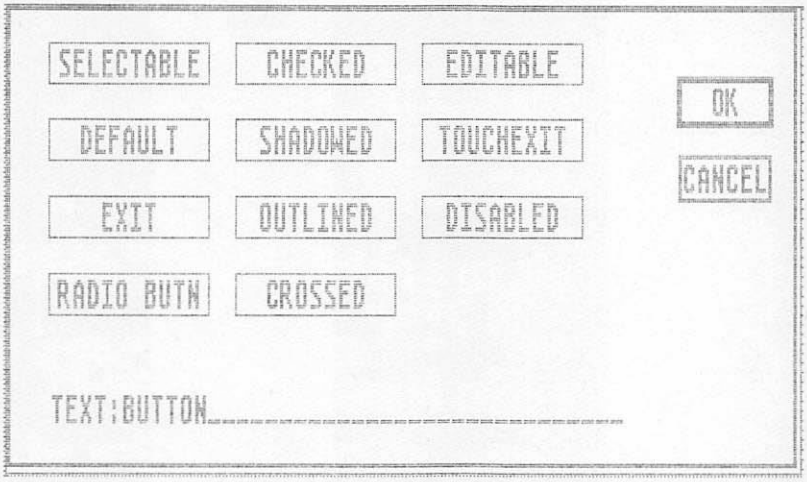


Bild 4.4.3: Button editieren

Bei einer *GraphicBox* ergibt sich folgender Dialog:

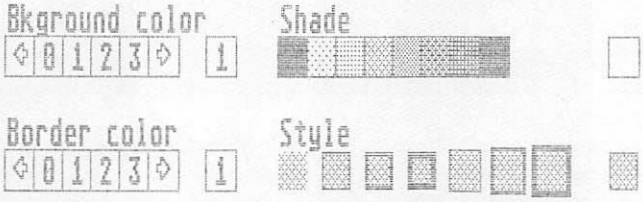


Bild 4.4.4: GraphicBox editieren

Es lassen sich mit der Maus also zusätzlich das Muster, die Stärke der Umrandung sowie deren Farben setzen.

Bei *GraphicBoxChar* kommt ein weiteres Feld zur Eingabe des Buchstabens hinzu:

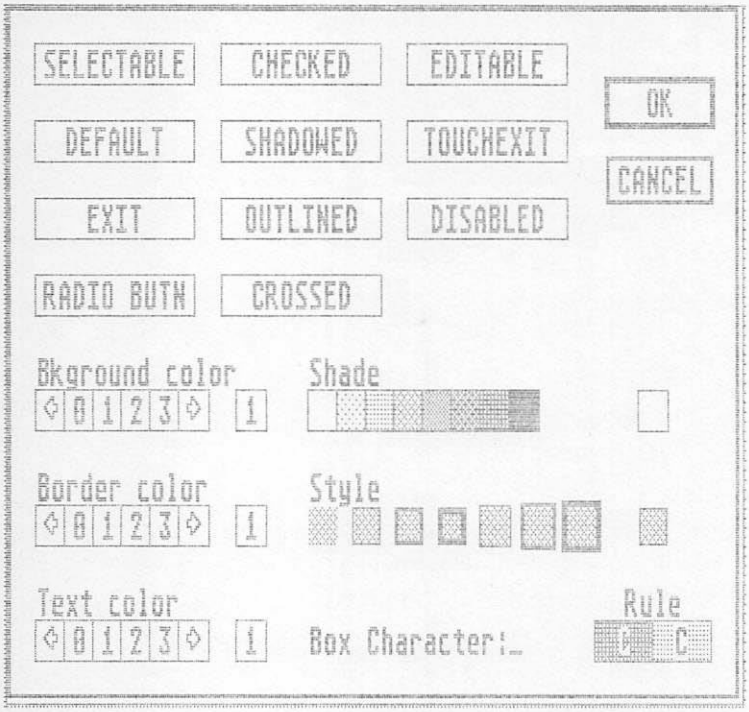


Bild 4.4.5: GraphicBoxChar editieren

"Textcolor" setzt die Textfarbe. Mit "Rule" können Sie entscheiden, ob der Text transparent dargestellt werden soll.

Die Objektarten *GraphicText*, *GraphicFormattedText*, *GraphicBoxText* und *GraphicFormattedBoxText* haben jeweils ein TedInfo, das ebenfalls vollständig im Dialog editiert werden kann.

Damit man das Platzhalterzeichen "_" in *PText*, *Ptmplt* und *PValid* von "unbenutzten" Zeichen unterscheiden kann, zeigt "MMRCP" diese durch die Tilde ("~") an. Dies ist nur eine Umformung im "MMRCP", in der Resource stehen dort natürlich die Unterstriche.

Verdeutlichen Sie sich noch einmal, welche Felder in der Dialog-Box welchen Feldern im TedInfo-Record entsprechen. Dabei werden auch die Vorteile der grafischen Eingabe deutlich.

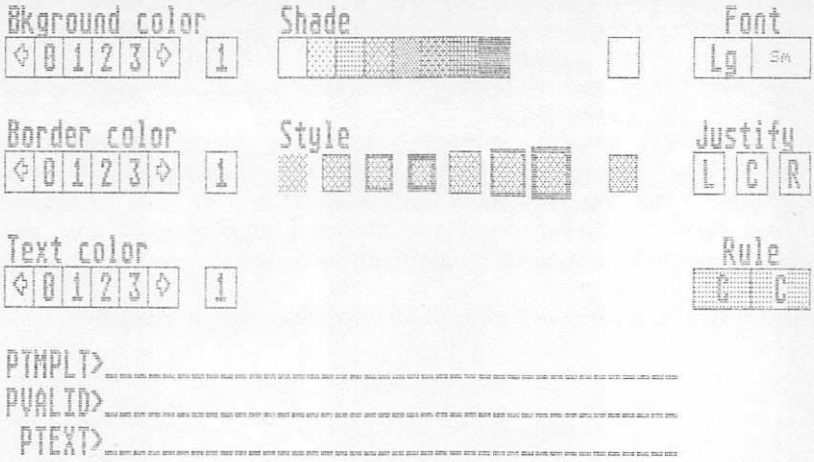


Bild 4.4.6: GraphicBoxText editieren

GraphicIcons werden in zwei Schritten editiert. Zunächst erhält man eine ähnliche Box wie zum Beispiel bei *GraphicString* zum Setzen der Flags. Zusätzlich können Text und einzelne Zeichen des Icons eingegeben werden.

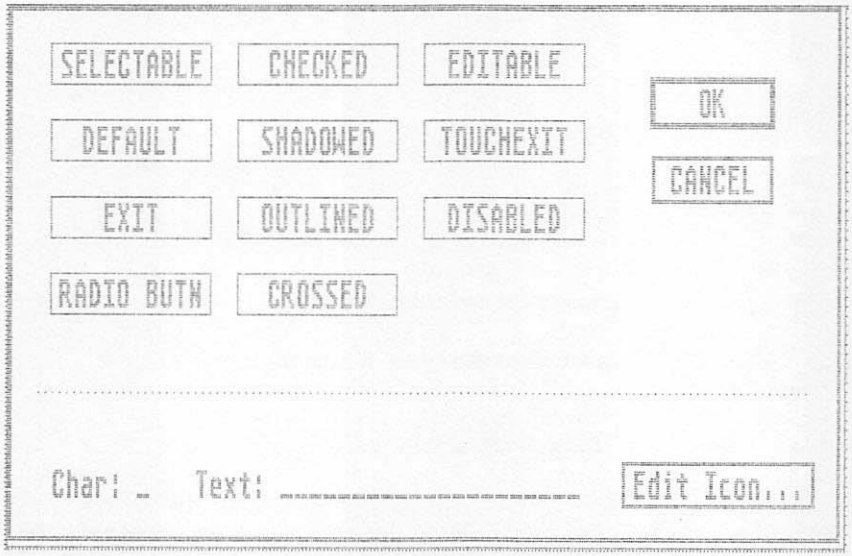


Bild 4.4.7: GraphicIcon editieren

Mit "Edit Icon" gelangen Sie in den Icon-Editor (Dieser wählt gleichzeitig ein "Cancel" in der ersten Box aus. Dadurch gehen dort alle Veränderungen verloren!). Das Icon wird nun in einem Raster dargestellt. Die Position des Textes und des einzelnen Zeichens werden grau angezeigt.

Sie können mit der Maus im Punkteraster die einzelnen Pixel des Icons setzen oder löschen. Mit dem Rollbalken bewegen Sie sich durch das Icon. (Es handelt sich übrigens nicht um ein richtiges Fenster - es werden nur verschiedene Objekte angezeigt, die wie ein Fenster aussehen und ähnlich behandelt werden!)

Rechts haben Sie die Auswahl zwischen verschiedenen Kommandos:

Clear	Löschen
All black	alles schwarz
Show Mask	zeige Maske
Show Icon	zeige Icon
Copy to Mask	kopiere in die Maske
B/W Color	schwarzweiß / Farbe
Icon Text	Icon / Text
Character	Zeichen
OK	Icon-Bearbeitung beenden
CANCEL	Icon-Bearbeitung abbrechen

Bild 4.4.8: Menü des Icon-Editors

Die Funktionen "Show Icon" und "Show Mask" schalten zwischen der Editierung des Icons und seiner Maske hin und her. Mit "Copy to Mask" wird der Inhalt des Icons in die Maske übertragen.

"Clear" und "All black" füllen das Icon oder seine Maske komplett weiß (alle Pixel löschen) oder schwarz (alle Pixel setzen).

Durch "B/W" und "Color" können Sie zwischen einer Editierung in schwarzweiß oder Farbe umschalten.

Mit "Icon", "Text" und "Character" kann zwischen drei Modi ausgewählt werden. "Icon" erlaubt die Editierung des Icons. Mit "Text" kann man die Position und Größe des Textes verändern und mit "Character" das einzelne Zeichen, deren Größe sich allerdings nicht ändern läßt.

Der jeweils editierfähige Teil des Objekts wird schwarz dargestellt; der Rest grau. Das Icon kann übrigens wie der Text verschoben und in seiner Größe variiert werden.

Zur Auswahl eines "Menü"-Baums haben Sie folgende Auswahl:

Bild 4.4.9: Die Menü-Partbox

Bei den angebotenen Elementen handelt es sich um einen GraphicTitle und zwei GraphicStrings. Das "TITLE"-Element wird für die Menü-Titel verwendet, "ENTRY" für einen Menü-Eintrag. Die graue Linie ist ein *GraphicString*, der nur aus "-"-Zeichen besteht und das Flag *Disabled* gesetzt hat. Die Strings "TITLE" und "ENTRY" sind nur Vorbelegungen, die Sie entsprechend den gewünschten Menüeinträgen verändern müssen.

Die Hintergründe der Einträge sind von Typ *GraphicBox*. Alle Objekt können Sie wie gewohnt in ihrer Größe verändern. Dabei ist darauf zu achten, daß die Einträge genau die Breite ihres Hintergrunds haben. Ansonsten wird bei ihrer Benutzung in einem Programm nicht die gesamte Zeile des Eintrags invertiert.

Objekte anwählen

Die Objekte werden mit der Maus ausgewählt. Dabei gibt es noch einige Kombinationen:

Anklicken
- Das Objekt wird ausgewählt.

Doppel-Klick
- Das Objekt wird "geöffnet".

Anklicken + <Control>-Taste
- Das Parent des Objekts wird ausgewählt.

Button gedrückt lassen
- Das Objekt kann verschoben werden. Wird es in der rechten unteren Ecke angeklickt, kann seine Größe verändert werden.

Button gedrückt lassen + <Shift>-Taste
- Vor dem Verschieben des Objektes wird eine Kopie angefertigt, die am alten Platz bleibt. Dieses Objekt hat dann keinen Namen mehr!

Button gedrückt lassen + <Control>-Taste
- Das Parent kann verschoben werden.

Button gedrückt lassen + <Shift>-Taste + <Control>-Taste
- Das Parent kann verschoben werden und eine Kopie von ihm bleibt am alten Platz.

Menüpunkte

Das *Edit*-Menü stellt Ihnen einen Zwischenspeicher zur Verfügung. Mit *Cut* wird ein angewähltes Objekt mitsamt Namen aus der Resource gelöscht und in den Puffer kopiert. Mit *Paste* können Sie dieses Objekt mit Namen in eine andere Resource hineinkopieren. Es ist dann noch im Zwischenspeicher vorhanden - aber ohne Namen.

Copy kopiert das angewählte Objekt ohne Namen in den Puffer. Mit *Erase* wird ein Objekt endgültig entfernt.

Im *File*-Menü können Sie mit *New* eine neue Resource erstellen oder mit *Open...* eine schon vorhandene Resource einlesen. Fehlt das ".DEF"-File, so sind alle Bäume *Unknown* und die Objekte haben keine Namen. Man kann mit *Open* auch - wie mit dem Doppel-Klick - einen Baum oder ein Objekt öffnen.

Merge... liest eine weitere Resource ein und hängt sie an die schon vorhandene an. *Save* und *Save as...* speichern die Resource, das ".DEF" und das Include-File unter dem alten oder einem neuen Namen ab.

Close schließt - wie das Anklicken des Schließ-Buttons im Fenster - einen Baum. Ist kein Baum geöffnet, so wird die Resource geschlossen, ohne daß etwas abgespeichert wird. Den gleichen Effekt hat *Abandon*.

Mit *Quit* wird das Programm verlassen. Haben Sie vorher nicht abgespeichert, erfolgt eine Rückfrage.

Mit dem *Options*-Menü können Sie verschiedene Operationen mit den Objekten ausführen. *Info* zeigt einige Informationen über ein Objekt oder einen Baum an. (In der TDI-Lizenzversion funktioniert dies nicht!)

Name... ist einer der wichtigsten Befehle. Da jedes Objekt, mit dem man arbeiten will, einen Namen haben muß, um im Include-File aufzutauchen, brauchen Sie diesen Befehl unbedingt. Dies betrifft besonders alle Objekte in den Menüs. Der Name darf nur aus Großbuchstaben bestehen und kann eine Maximallänge von acht Zeichen haben.

Hide setzt das *HideTree*-Flag für das angewählte Objekt oder den Unterbaum. Es erscheint dann nicht mehr auf dem Bildschirm. Mit *Unhide* wird dies rückgängig gemacht.

Sort... sortiert alle Kinder des angewählten Objekts. Dabei wird die Reihenfolge entsprechend ihrer Position gesetzt. Sie haben die Auswahl zwischen vier Sortierreihenfolgen. Beim Sortieren werden nur die Zeiger im Baum verändert - alle Objekte behalten den gleichen Index.

Mit *Recreate* werden die Objekte ebenfalls nach ihrer Position sortiert, nur verändern sich hierbei ihre Indizes, so daß sie im Baum verschoben werden. Man benötig diese Funktion, wenn die Reihenfolge von editierbaren Feldern im Baum der Reihenfolge auf

dem Bildschirm entsprechen soll. Ist dies nicht der Fall, so wird bei einem Dialog mit den Cursor-Tasten kreuz und quer durch die Felder gesprungen.

Flatten erzeugt Geschwister aller Kinder des ausgewählten Objekts.

Snap "zieht" das gewählte Objekt an eine Zeichengrenze. Dies hat nur Sinn bei *Free*-Bäumen. Bei den anderen geschieht das automatisch.

Um ein Icon von Diskette zu laden, müssen Sie *Load...* benutzen (funktioniert bei der TDI-Version ebenfalls nicht!). Da aber ein Icon-Editor eingebaut ist, werden Sie diesen Punkt wohl nur dann benötigen, wenn Sie alte Icons, die mit einem anderen Programm erzeugt wurden, benutzen wollen.

Natürlich können nicht immer alle Menüpunkte ausgewählt werden. Dies ist von der jeweiligen Situation abhängig. Einige Kommandos können auch per Tastatur aufgerufen werden. Dann steht im Menü ein Buchstabe, der zusammen mit der <Control>-Taste gedrückt werden muß.

Unterschiede zum RCS

Das RCS war das erste Programm, mit dem Resources erstellt werden konnten. Es wurde mit dem Entwicklungspaket von ATARI ausgeliefert, ist aber inzwischen auch einzeln erhältlich.

Es bietet die gleichen Kommandos wie das MMRCP und enthält zusätzlich noch die Baumart "Alert". Man kann eine Alertbox erzeugen, indem man bis zu sechs GraphicStrings und eins der drei Icons verwendet. In Dialogen können auch GraphicImages benutzt werden.

Gegenüber dem MMRCP hat das RCS jedoch einige gravierende Nachteile aufzuweisen.

Probleme tauchen vor allem bei der Benutzung von Icons auf. Soll eine Dialogbox angezeigt und editiert werden, in der Icon-Objekte vorkommen, löst dies in der Regel einen Systemabsturz aus. Teilweise wird sogar das GEM abgehängt, so daß der ST nicht mehr ansprechbar ist. Der Inhalt einer RAM-Disk geht dabei in der Regel verloren.

Abhilfe läßt sich hier schaffen, indem Sie Icons nur zum Schluß in die Resource einfügen. Dadurch läßt sie sich aber nicht mehr editieren. Besser wird das Problem umgangen, indem Sie die Icons im Programm selbst mit *ObjectAdd* in die Resource einfügen. Trotzdem ist auch diese Lösung unbefriedigend.

Das zweite große Problem betrifft das *Merge*-Kommando. Hier vergißt oder verschiebt das RCS die Namen der Objekte fast nach Belieben. Sie können sich nur dadurch helfen, daß Sie in der "hinzuzu-mergenden" Datei alle Namen löschen und nach dem Einbinden neu definieren.

Weiterhin ist das RCS weniger anwenderfreundlich als das MMRCP. Alle Objektarten, die in die Resource übernommen werden können sind in einem eigenen Fenster dargestellt. Um an sie heranzukommen, müssen Sie dieses Fenster erst aktivieren. Hinterher muß wiederum das Resource-Fenster nach oben geholt werden. Um ein einziges Objekt einzufügen sind also diverse Mausaktionen nötig. Außerdem fehlt die Möglichkeit, Kommandos auch mit der Tastatur aufrufen zu können.

Schließlich hat das RCS keinen eingebauten Icon-Editor. Sie brauchen also ein zusätzliches Programm zur Icon-Erstellung und müssen zwischen beiden wechseln.

Alle Operationen des RCS können also genausogut - aber einfacher und schneller - mit dem MMRCP ausgeführt werden.

4.5 Internes Format der Resource-Files

Das interne Format der Resource-Files, wie es vom Resource-Manager verlangt wird, ist normalerweise für den Programmierer weniger wichtig. Trotzdem ist es interessant und zeigt auch etwas von den internen Mechanismen des GEM.

Ein Resource-File besteht hauptsächlich aus drei Komponenten: dem Resource-Header, dem Tree-Index und den Objekten.

Der Resource-Header steht am Anfang eines Resource-Files. DA befinden sich Informationen über die enthaltenen Objekte und Strukturen. Hier finden Sie die Aufteilung innerhalb des Files, die Anzahl der Strukturen sowie Objekte, TedInfos und so weiter.

Der Resource-Header ermöglicht es, alle anderen Informationen an beliebiger Stelle im Resource-File zu halten. Dies kann - je nach Resource-Construction-Programm - verschieden sein. Ob sich durch die Position der Elemente ein unterschiedliches Zeitverhalten ergibt, ist nicht bekannt.

Im Header ist auch die Position eines Feldes mit den jeweiligen Tree-Indizes vermerkt. Über dieses Feld von Adressen ermittelt GEM die wirkliche Position der einzelnen Objekte, wenn nur deren Index übergeben wird.

Es folgt die Definition von Header-Struktur und Tree-Index.

```
TreeIndex = ARRAY [0..NrObjects-1] OF ADDRESS;

RscHeader = RECORD
              version,         (* Resource-Version          *)
              ixObjects,       (* Offset zu den Objekten    *)
              ixTedInfo,       (* Offset zu den Tedinfos    *)
              ixIconBlks,      (* Offset zu den IconBlocks  *)
              ixBitBlks,       (* Offset zu den BitBlocks   *)
              ixStrings,       (* Offset zu den Strings     *)
              ixFreeStrings,   (* Offset zu den FreeStrings *)
              ixImages,        (* Offset zu den Images      *)
              ixFreeImages,    (* Offset zu den FreeImages  *)
              ixTreeIndx,      (* Offset zum TreeIndex      *)
              nrObjects,       (* Anzahl der Objekte        *)
              nrTrees,         (* Anzahl der Bäume          *)
              nrTedInfos,      (* Anzahl der TedInfos       *)
              nrIconBlks,      (* Anzahl der IconBlocks     *)
              nrBitBlks,       (* Anzahl der BitBlocks      *)
              nrStrings,       (* Anzahl der Strings        *)
              nrImages,        (* Anzahl der Images         *)
              length           (* Größe des RSC             *)
                        :CARDINAL;
            END
```

Bild 4.5.1: Der Aubau von Resource-Files

Wie schon bemerkt, sind diese Informationen über den Aufbau von Resource-Files normalerweise nicht nötig, zudem auch nicht bekannt ist, wie GEM intern mit ihnen umgeht oder sie verändert.

Es kann jedoch sein, daß man mit Resource-Files direkt arbeiten will, um sie beispielsweise zu verändern. Im Toolkit zum Modula-2-System von TDI ist ein Programm enthalten, das aus einem Resource-File ein Modula-2-Programm erstellt, das mitkompiliert werden kann. Für solche Anwendungen sind diese Informationen wichtig.

5. Der Form-Manager

Mein Meister, mein Meister, hörst du das Grollen?
Die wilden Bits durch den Speicher tollen!
Nur ruhig, nur ruhig, das haben wir gleich,
Die sperren wir in den Pufferbereich.

Ballade vom Bitkönig

Eine Spielart von Objektbäumen wird bei GEM als *Forms* (Formulare) zusammengefaßt. Dies sind die Dialogboxen in ihren verschiedenen Formen. Die Bezeichnung Formular ist sehr zutreffend, da die Boxen aus festen und editierbaren Feldern bestehen. Der Form-Manager stellt Routinen bereit, die dem Benutzer erlauben, mit Formularen zu arbeiten.

5.1 Dialogboxen zentrieren

Es ist günstig, Dialoge in der Mitte des Bildschirms darzustellen, um sie optisch besser zu positionieren. Zum Zentrieren gibt es die Funktion

```
FormCenter ( Tree : ADDRESS ; VAR X, Y, W, H : INTEGER )
fo_cresvd = form_center (fo_ctree, &fo_cx, &fo_cy,
                         &fo_cw, &fo_ch)
```

Sie errechnet für einen Baum, dessen Adresse in *Tree* steht, die Koordinaten, die nötig sind, um ihn zentriert auszugeben (in *X* und *Y*). Zusätzlich wird noch der Platz zurückgegeben, den der Baum belegt hatte (in *W* und *H*). GEM trägt diese Werte direkt in den Objektbaum ein.

Diese Funktion kann auf alle beliebigen Objekte und Unterbäume angewendet werden.

Das Ergebnis der C-Funktion ist für später reserviert. Momentan wird hier immer eine 1 erzeugt.

5.2 Dialoge vorbereiten

Die Routine

```
FormDialogue ( Flag, LIX, LIY, LIW, LIH,
               BIX, BIY, BIW, BIH : INTEGER )

fo_direturn = form_dial (fo_diflag,
                         fo_dilx, fo_dily, fo_dilw, fo_dilh,
                         fo_dibx, fo_diby, fo_dibw, fo_dibh)
```

übernimmt insgesamt vier Funktionen, die über *Flag* ausgewählt werden.

FormStart (Kennwert 0) reserviert den in *LBX*, *LBY*, *LBW* und *LBH* angegebenen Bildschirmbereich. Die anderen Parameter sind dabei mit 0 zu übergeben.

Die Funktion hat zwar keine sichtbare Auswirkung, sorgt jedoch dafür, daß keine anderen Applikationen auf diesen Bereich zugreifen können.

FormGrow (Kennwert 1) dient nur einem optischen Effekt. Die Funktion zeichnet ein anwachsendes Rechteck, dessen Anfangs- und Endgröße in den *LI_*- (little) beziehungsweise in den *BI_*-Parametern (big) angegeben wird.

FormShrink (Kennwert 2) erzeugt den umgekehrten Effekt: ein schrumpfendes Rechteck. Die Parameter werden ebenso wie bei *FormGrow* benutzt. Beide Funktionen sind nicht notwendig; Sie können sie auch weglassen, wenn der optische Effekt nicht benötigt oder erwünscht wird.

Mit *FormFinish* (Kennwert 3) wird der angegebene (und vorher reservierte) Bildschirmbereich wieder freigegeben. Alle Applikationen, die diesen Bereich vorher benutzt haben, erhalten von GEM die Mitteilung, daß der Bereich wiederhergestellt werden muß.

Die ersten beiden Funktionen werden vor und die anderen nach einem Dialog aufgerufen.

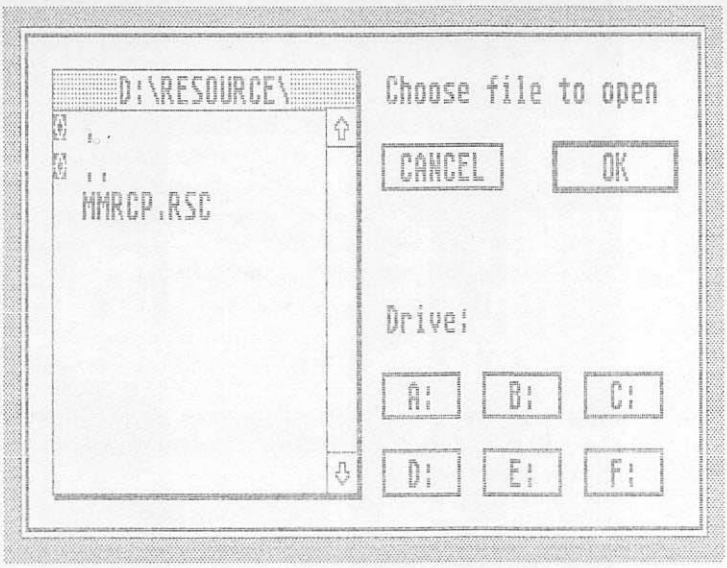


Bild 5.4.2: Die FileSelect-Box aus dem Megamax-System

Zwei weitere Routinen des Form-Managers, *FormAlert* und *FormError* haben Sie schon in Kapitel 1.1 und 1.2 kennengelernt.

6. Der Menü-Manager

Er tastet wild, er tastet besessen,
Mist - den Programmstop vergessen!
Plötzlich hört man, wie er schreit,
Es fehlt am Platz genau 1 Byte.
Jetzt ist der Puffer übergeschwappt
Er kam zu spät - der Interrupt.
Ballade vom Bitkönig

Der Menü-Manager ist für die Verwaltung der Menüzeile zuständig. Er vereinfacht das Ansprechen der Menüobjekte, indem er mehrere Aufrufe des Objekt-Managers zusammenfaßt.

6.1 Menüzeile anzeigen

Um einen Menübaum anzuzeigen, wird

MenuBar (Tree: ADDRESS; Show:INTEGER)
me_breturn = menu_bar(me_btree, me_bshow)

benutzt. *Tree* gibt die Adresse des Menübaums an. GEM zeichnet nun die Menüzeile und ist ab sofort bereit, mit dem Menübaum zu arbeiten, der unabhängig vom laufenden Programm gebraucht wird.

Show ist ein Flag, nach dem die Menüzeile angezeigt (1) oder gelöscht (0) wird. Das Löschen der Menüzeile wird man kaum brauchen.

Die Routine installiert die Adresse des Baums intern, trägt die Namen der Accessories in das "Desk"-Menü ein und führt dann ein *ObjectDraw* ab der Koordinate 0,0 aus.

6.2 Titel verändern

Nachdem der Benutzer einen Menüpunkt ausgewählt hat, bleibt der Titel des Menüs invers dargestellt. Es ist die Aufgabe des Programms, diesen wieder normal darzustellen. Dazu dient die Routine

MenuTitelNormal (Tree:ADDRESS; Title, Normal:INTEGER)
me_nreturn = menu_tnormal(me_ntree, me_ntitle, me_nnormal)

Tree ist wiederum der Menübaum und *Title* der Index des betreffenden Titelobjekts. *Normal* ist ein Flag, das darüber entscheidet, ob der Titel normal (1) oder invers (0) gezeichnet wird. Das Invertieren eines Titels vom Programm aus macht dort Sinn, wo durch Tastatureingaben Menü-Kommandos ausgelöst werden können.

Die Routine setzt oder löscht die *Selected*-Flags der Titel und zeichnet das Objekt anschließend neu.

6.3 Einträge verändern

Sie können den Text eines Menü-Eintrags während des Programmlaufs ändern. Dazu brauchen Sie die Routine

MenuText (Tree:ADDRESS; Item: INTEGER; Text:ADDRESS)
me_treturn = menu_text(me_ttree, me_titem, me_ttext)

Damit erhält der Eintrag *Item* im Menübaum an der Stelle *Tree* einen neuen Text, dessen Adresse in *Text* übergeben wird.

Manchmal ist es sinnvoll, den Text eines Menüs zu ändern, wenn dies dem Programmzustand entspricht (z.B. "Grafik ein" zu "Graphik aus"). Allzu häufig wechselnde Menütexte sind allerdings nicht angebracht.

Die Länge des neuen Texts sollte die des alten nicht übersteigen, damit er nicht über den Menühintergrund hinausragt.

Die Routine ändert nur das *Spec*-Feld des Objekts, in dem die Adresse des Texts steht.

Das Ein- und Ausschalten von Menüoptionen geschieht mit

MenuItemEnable (Tree:ADDRESS; Item, Enable:INTEGER)
me_ereturn = menu_ienable(me_etree, me_eitem, me_eenable)
Dabei bestimmt *Enable*, ob der Menü-Eintrag grau und nicht mehr anwählbar (0) oder normal und anwählbar (1) erscheinen soll.

Bild 6.3.1: Ein- und ausgeschaltete Menüeinträge

Die Routine setzt oder löscht nur das *Disable*-Flag des betreffenden Objekts.
Zur Anzeige, ob eine Option angewählt ist oder nicht, wird ein kleines Häkchen (engl. Checkmark) vor dem Eintrag verwendet. Es kann mit

MenuItemCheck (Tree:ADDRESS; Item, Check:INTEGER)
me_creturn = menu_icheck(me_ctree, me_citem, me_ccheck)

gesetzt und gelöscht werden. Darüber entscheidet *Check*. Bei einer 1 erscheint vor dem Eintrag ein Häkchen, bei einer 0 nicht.

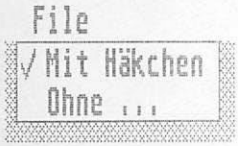


Bild 6.3.2: Menüeinträge mit und ohne Häkchen

Es wird wiederum nur das *Checked*-Flag bei dem Objekt gesetzt oder gelöscht.

6.4 Accessory-Einträge

Das "Desk"-Menu enthält sechs freie Einträge, in die GEM die Namen der Accessories einträgt. Accessories selbst können mit

MenuRegister (Apid:INTEGER; VAR PString:ARRAY OF CHAR):INTEGER
me_rmenuid = menu_register(me_rapid, me_rpstring);

GEM ihren Namen mitteilen und eine Eintragung veranlassen.

Apid ist dabei die Identifikationsnummer der Accessory-Applikation, die mit *ApplInitialise* vergeben wird. In *PString* steht der Eintragstext.

Das Ergebnis enthält die Nummer des Accessories (0 bis 5) oder eine -1, falls kein Platz mehr frei war und das Accessory nicht installiert werden konnte.

7. Weitere Manager

Der Rechner schreit auf in höchster Qual,
Da zuckt durch das Fenster ein Sonnenstrahl -
Das Terminal schimmert im Morgenrot,
Das Programm ist gestorben, der Bitkönig tot!

Ballade vom Bitkönig

Neben den schon beschriebenen Bibliotheken sind noch einige andere nötig, um mit Resources umzugehen. Die wichtigsten werden in diesem Kapitel beschrieben.

7.1 Der Event-Manager

Der Event-Manager deckt das Mitteilungssystem von GEM ab. Damit kann ein Programm zu einer bestimmten Aktion aufgefordert werden.

GEM versendet Mitteilungen, wenn zum Beispiel ein Fenster verschoben wurde. Das Programm muß dann auf diese Verschiebung, d.h. auf die entsprechende Mitteilung, reagieren. Mitteilungen können auch von Programmen an andere Programme geschickt werden; diese Technik macht jedoch nur bei aufeinander abgestimmten Programmpaketen Sinn und wird deshalb hier nicht näher beschrieben.

Mit den Routinen des Event-Managers wartet ein Programm auf eine solche Mitteilung oder auf das Eintreten eines anderen Ereignisses, zum Beispiel auf eine Tastatureingabe.

Tritt das erwartete Ereignis ein, so wird das wartende Programm wieder erweckt und kann weiterarbeiten. Accessories warten etwa darauf, daß sie aus dem "Desk"-Menü angewählt werden. Tritt dies ein, verschickt GEM eine Mitteilung und das Accessory kann anfangen zu arbeiten.

Die erste Routine wartet darauf, daß eine Taste gedrückt wird

```
EventKeyboard():INTEGER;
ev_kreturn = evnt_keybd()
```

Der Funktionswert entspricht dem Code für die gedrückte Taste.

Mit der zweiten Routine erwartet ein Programm einen Maus-Klick.

```
EventButton (Clicks, BMask, BState:INTEGER;
             VAR BmX, BmY, BButton, Bkstate:INTEGER):INTEGER;
ev_breturn=event_button(ev_bclicks, ev_bmask, ev_bstate,
             &ev_bmx, &ev_bmy, &ev_bbutton, &ev_bkstate)
```

Clicks enthält die Anzahl der erwarteten Maus-Klicks. In *BMask* steht, welche Knöpfe beachtet werden sollen:
1 = linker Knopf
2 = rechter Knopf
3 = beide Knöpfe

BState gibt an, auf welchen Zustand der Knöpfe das Programm warten soll. Hierbei gilt:
0 = beide Knöpfe losgelassen
1 = linker Knopf gedrückt
2 = rechter Knopf gedrückt
3 = beide Knöpfe gedrückt

In *BmX* und *BmY* stehen die Koordinaten des Mauszeigers, zum Zeitpunkt des Maus-Klicks. *BButton* enthält die Zustände der Mausknöpfe. Die Bedeutung des Wertes ist dabei identisch mit *BState*.

Bkstate beinhaltet Angaben über den Zustand der Tastatur beim Maus-Klick. Dabei gilt:
1 = rechte Shift-Taste gedrückt
2 = linke Shift-Taste gedrückt
4 = Control-Taste gedrückt
8 = Alternate-Taste gedrückt

Sind mehrere Tasten gleichzeitig gedrückt, so enthält *Bkstate* die Addition dieser Werte. Daduch ist es möglich, verschiedene Kombinationen (etwa Control-Taste plus rechter Mausknopf) abzufragen.

Das Ergebnis der Funktion gibt an, wie viele Klicks erfolgt sind.

Die nächste Routine arbeitet mit dem Mauszeiger.

```
EventMouse(MoFlags, MoX, MoY, MoWidth, MoHeight:INTEGER;
           VAR MomX, MomY, MoButton, MokState:INTEGER)
evnt_mouse(ev_moflags, ev_mox, ev_moy, ev_mowidth, ev_moheight,
           &ev_momx, &ev_momy, &ev_mobutton, &ev_mokstate)
```

Hier wird darauf gewartet, daß der Mauszeiger in ein bestimmtes Rechteck eintritt oder es verläßt. In *MoFlags* steht eine 0 für das Eintreten ins Rechteck oder eine 1, wenn der Zeiger das Rechteck verläßt.

MoX, *MoY*, *MoWidth* und *MoHeight* definieren dieses Rechteck. In *MomX* und *MomY* erhält man die Koordinaten des Mauszeigers zurück, die er zu dem Zeitpunkt hatte, als er die Rechteckgrenzen übertrat.

MoButton und *MokState* enthalten den momentanen Zustand der Mausknöpfe und der Tastatur. Ihre Bedeutung ist identisch mit den oben beschriebenen Werten *BButton* und *BkState*.

Mit der Routine

```
EventMessage(Pbuff:ADDRESS);
evnt_mesag(ev_mgpbuff)
```

erwartet das Programm eine Mitteilung. *Pbuff* ist dabei die Adresse eines 16 Bytes großen Puffers, der die Mitteilung aufnehmen soll. Dieser ist als Feld aus Integer-Werten zu organisieren.

Im ersten Wert steht eine Kennzahl für die Art der Mitteilung, und der zweite enthält die Kennzahl des Programms, das die Mitteilung geschickt hatte. Der dritte Wert gibt die Länge der Mitteilung an, aber abzüglich der normalen 16 Bytes (dies ist nur dann nötig, falls die Mitteilung wirklich von einem Programm kam und nicht von GEM).

Die Mitteilungsarten und die Bedeutungen der Werte im Mitteilungspuffer finden Sie hier aufgelistet:

MenuSelected (10)	: Ein Menü-Eintrag wurde ausgewählt 4.Wert: Menü-Titel 5.Wert: Menü-Eintrag
WindowRedraw (20)	: Fensterinhalte sollen neu gezeichnet werden 4.Wert: Fenster-Kennzahl 5.Wert: X-Koordinate 6.Wert: Y-Koordinate 7.Wert: Breite 8.Wert: Höhe

```
WindowTopped (21)          : Ein Fenster wurde nach "oben" geholt
                             4.Wert: Fenster-Kennzahl
WindowClosed (22)          : Ein Fenster wurde geschlossen
                             4.Wert: Fenster-Kennzahl
WindowFulled (23)          : Ein  Fenster soll auf die volle  Größe
                             gebracht werden
                             4.Wert: Fenster-Kennzahl
WindowArrowed (24)         : Einer der Pfeil-Knöpfe am  Fensterrand
                             oder die Rollbalken wurden angeklickt
                             4.Wert: Fenster-Kennzahl
                             5.Wert: was soll gemacht werden ?
                                     0= Seite hoch
                                     1= Seite runter
                                     2= Zeile hoch
                                     3= Zeile runter
                                     4= Seite links
                                     5= Seite rechts
                                     6= Spalte links
                                     7= Spalte rechts
WindowHorizSlided (25)     : Die waagerechte Rollbox wurde bewegt
                             4.Wert: Fenster-Kennzahl
                             5.Wert: neue Position der Box (1..1000)
WindowVertSlided (24)      : Die senkrechte Rollbox wurde bewegt
                             4.Wert: Fenster-Kennzahl
                             5.Wert: neue Position der Box (1..1000)
WindowSized (27)           : Die Fenstergröße wurde verändert
                             4.Wert: Fenster-Kennzahl
                             5.Wert: X-Koordinate
                             6.Wert: Y-Koordinate
                             7.Wert: Breite
                             8.Wert: Höhe
WindowMoved (28)           : Ein Fenster wurde bewegt
                             4.Wert: Fenster-Kennzahl
                             5.Wert: X-Koordinate
                             6.Wert: Y-Koordinate
                             7.Wert: Breite
                             8.Wert: Höhe
WindowNewTop (29)          : Ein Fenster wurde aktiviert
                             4.Wert: Fenster-Kennzahl
AccessoryOpen (40)         : Ein Accessory wurde ausgewählt
                             4.Wert: Kennzahl des Accessories
AccessoryClose (41)        : Ein Accessory wurde geschlossen
                             4.Wert: Kennzahl des Accessories
```

Die nächste Routine wartet darauf, daß eine bestimmte Zeitspanne vergeht:

```
EventTimer(LoCount,HiCount:INTEGER)
evnt_timer(ev_tlocount, ev_thicount)
```

Dabei soll der Wartezyklus LoCount+HiCount*65535 Millisekunden betragen.

Die letzte Funktion faßt alle Ereignisse zusammen und wartet, bis ein beliebiges eintritt.

```
EventMultiple (Flags, Clicks, Mask, State,
               M1Flags, M1X, M1Y, M1Width, M1Height,
               M2Flags, M2X, M2Y, M2Width, M2Height:INTEGER;
               PBuff:ADDRESS;
               LoCount,HiCount:INTEGER;
               VAR MoX, MoY, MoButton, MokState,
               KReturn, BReturn:INTEGER):INTEGER;
ev_mwhich=evnt_multi(flags,clicks,bmask,bstate,
                     m1flags,m1x,m1y,m1width,m1height,
                     m2flags,m2x,m2y,m2width,m2height,
                     mgpbuff,
                     tlocount,thicount,
                     &mox,&moy,&mobutton,&mokstate,
                     &kreturn,&breturn)
```

Mit *Flags* wird festgelegt, auf welche Ereignisse gewartet werden soll. Dabei bedeutet:

```
1 = Tastatur-Ereignis (entspr. EventKeyboard)
2 = Button-Ereignis   (entspr. EventButton)
4 = Mouse-Event       (entspr. EventMouse mit den Parametern
                       M1Flags-M1Height)
8 = Mouse-Event       (entspr. EventMouse mit den Parametern
                       M2Flags-M2Height)
16= Mitteilung        (entspr. EventMessage)
32= Timer             (entspr. EventTimer)
```

Soll auf mehrere Ereignisse gewartet werden, so sind die Kennzahlen einfach zu addieren.

Die Parameter sind eine Zusammenfassung der oben beschriebenen Funktionen. Das Funktionsergebnis gibt an, welche Ereignisse eingetreten sind. Die Bedeutung dieses Wertes entspricht *Flags*.

In den Beispielprogrammen werden einige dieser Routinen verwendet und ausführlich erklärt.

7.2 Der Graphics-Manager

Der Graphics-Manager stellt mehrere grafische Routinen zur Verfügung. Davon lassen sich einige auch beim Umgang mit Resources verwenden.

Die erste Routine ermöglicht dem Benutzer, mit gedrückten Mausknopf ein Rechteck auf dem Bildschirm zu verschieben.

```
GrafDragBox (Width, Height, StartX, StartY,
             BoundX, BoundY, BoundW, BoundH:INTEGER;
             VAR FinishX, FinishY:INTEGER);
graf_dragbox(width, height, startx, starty,
             boundx, boundy, boundw, boundh,
             &finishx, &finishy)
```

Das Rechteck wird durch *Width*, *Height*, *StartX* und *StartY* festgelegt. Der Bereich, in dem es bewegt werden kann, wird durch *BoundX* bis *BoundH* begrenzt.

In *FinishX* und *FinishY* finden sich die Koordinaten des Rechtecks, wenn der Benutzer den Mausknopf losgelassen hat.

Die nächsten beiden Routinen zeichnen eine sich vergrößernde bzw. verkleinernde Box auf den Bildschirm.

```
GrafGrowBox(StX, StY, StWidth, StHeight,
            FinX, FinY, FinWidth, FinHeight:INTEGER);
graf_growbox(stx,sty,stwidth,stheight,
             finx,finy,finwidth,finheight)
```

Das Rechteck beginnt an der Position *StX*, *StY* mit der Breite *StWidth* in der Höhe *StHeight* und wächst dann an, bis es die Position *FinX*, *FinY* sowie Breite und Höhe *FinWidth* bzw. *FinHeight* erreicht hat.

Die Routine

```
GrafShrinkBox(FinX, FinY, FinWidth, FinHeight,
              StX, StY, StWidth, StHeight:INTEGER)

graf_shrinkbox(finx,finy,finwidth,finheight,
               stx,sty,stwidth,stheight)
```

dient dazu genau das Gegenteil zu erreichen: Die Box beginnt bei den *Fin...*-Parametern und schrumpft auf die *St...*-Parameter zusammen.

Diese beiden Routinen werden übrigens bei einem Aufruf von *FormDialogue* aus dem Form-Manager mit den Parametern *FormGrow* beziehungsweise *FormShrink* benutzt.

Nun folgen zwei Routinen, die direkt mit Resources arbeiten. Die erste ist:

```
GrafWatchBox(Tree:ADDRESS; Object,
             InState, OutState:INTEGER):INTEGER;
gr_wreturn=graf_watchbox(ptree,object,instate,outstate)
```

Hiermit wird festgestellt, ob der Mauszeiger sich auf einem Objekt befindet (bei gedrücktem Knopf). Ist dies der Fall, nimmt *Object* im Baum *Tree* den Zustand *InState* an, ansonsten bekommt es den Zustand *OutState*.

Das Setzen des Zustandes geschieht durch eine Änderung des *state*-Feldes des Objekts. *InState* und *OutState* errechnen sich daher aus den gleichen Werten.

Das Funktionsergebnis gibt an, ob sich der Mauszeiger in dem Moment, als der Knopf losgelassen wurde innerhalb (1) oder außerhalb (0) des Objektes befand.

Mit der zweiten Routine lassen sich Regler (engl. Slider) darstellen.

```
GrafSlideBox(Tree:ADDRESS; Parent, Object, VH:INTEGER):INTEGER;
gr_slreturn=graf_slidebox(ptree,parent,object,vh)
```

Diese Routine ermöglicht dem Benutzer, bei gedrücktem Mausknopf das Object *Object* in dem Objekt *Parent* (beide aus dem Baum *Tree*) zu veschieben.

VH entscheidet darüber, ob das Objekt vertikal (1) oder horizontal(0) im *Parent*-Objekt verschoben werden kann.

Das Funktionsergebnis entspricht der Position des Objekts innerhalb *Parent* als der Mausknopf losgelassen wurde. Dabei werden Werte von 1 bis 1000 zurückgegeben, die Sie anschließend noch umrechnen müssen.

Abschließend noch zwei Routinen, die zwar nicht direkt mit Resources arbeiten, die aber doch häufig gebraucht werden.

Die erste Routine dient zum Verändern der Mausform:

```
GrafMouse(Number:INTEGER; Faadr:ADDRESS)
graf_mouse(number,faddr)
```

Number gibt dabei an, welche Form der Mauszeiger annehmen soll. Dabei gilt:

0 = Pfeil
1 = Cursor
2 = Biene ("busy bee")
3 = zeigende Hand
4 = flache Hand

5 = dünnes Fadenkreuz
6 = dickes Fadenkreuz
7 = umrissenenes Fadenkreuz
255 = von Benutzer definierte Mausform
256 = Mauszeiger ausschalten
257 = Mauszeiger einschalten

Bild 7.2.1: Die Maustypen

Faddr gibt bei benutzerdefinierten Mausformen deren Adresse an.

Wo sich der Mauszeiger gerade befindet, können Sie mit der folgenden Routine feststellen:

GrafMouseKeyboardState(VAR MX, MY, MState, KState:INTEGER)
graf_mkstate(&mx,&my,&mstate,&kstate)

In *MX* und *MY* werden die Koordinaten des Mauszeigers zurückgegeben. *MState* gibt an, welche Knöpfe gedrückt sind:
0 = kein Knopf gedrückt
1 = linker Knopf gedrückt
2 = rechter Knopf gedrückt
3 = beide Knöpfe gedrückt

KState enthält zusätzlich noch den Zustand der Tastatur. Dabei gilt:
0 = keine Taste gedrückt
1 = rechte Shift-Taste gedrückt
2 = linke Shift-Taste gedrückt
4 = Control-Taste gedrückt
8 = Alternate-Taste gedrückt

Sind mehrere Tasten gleichzeitig gedrückt, wird die Summe der entsprechenden Kennwerte zurückgegeben.

8. Allgemeines über Benutzerschnittstellen

Der Sinn von grafischen Oberflächen wie GEM liegt darin, die Bedienung des Computers zu vereinfachen. Die zugrundeliegenden Konzepte sind in vielen Arbeiten entwickelt und abgehandelt worden. An dieser Stelle soll deshalb nur eine kurze Zusammenfassung gegeben werden.

Da ist zunächst die Überlegung, den Bewegungsaufwand mit der Maus zu minimieren. Das heißt, es sollen möglichst wenige und kurze Bewegungen des Mauszeigers nötig sein, um eine Programmfunktion auszulösen. So macht es zum Beispiel wenig Sinn, eventuelle Icons weit entfernt von der Menüleiste zu plazieren.

Dieser Fehler wurde zum Beispiel bei dem Programm *1stWord* begangen. Die Icons am unteren Rand des Bildschirms werden sehr selten genutzt, da der Aufwand, sie zu erreichen, einfach zu groß ist.

Es gibt eine Art von "Muskelgedächtnis". Das heißt, die Erinnerung, welche Bewegung nötig ist, um ein bestimmtes Ziel zu erreichen, wird praktisch auf eine bestimmte Muskelbewegung reduziert, die sich dann recht schnell und fast unbewußt ausführen läßt. Ein Beispiel dafür ist das Zehnfinger-Blindschreiben: geübte Schreiber überlegen nicht mehr, wo sich welcher Buchstabe auf der Tastatur befindet.

Bei Benutzeroberflächen heißt das, daß bestimmte Funktionen an gleichen Stellen plaziert werden. So z.B. die Position des "Desk"-Menüs oder die eines "File"-Menüs direkt daneben. Steht der "Verlasse"-Befehl etwa bei jedem Programm unten im "File"- oder "Datei"-Menü, so ist er einfacher zu erreichen, als wenn seine Position (auf dem Bildschirm) bei jedem Programm wechselt.

Der nächste Punkt betrifft das visuelle Erlebnis. Die Wichtigkeit einzelner Objekte läßt sich auf dem Bildschirm optisch verdeutlichen. Hat der Anwender zum Beispiel eine Löschoption ausgewählt, kann seine Aufmerksamkeit durch eine

farbige Rückfrage erheblich gesteigert werden. Dies verhindert auch einen gewissen Gewöhnungseffekt, der sich nach einiger Zeit einstellt. Durch "außergewöhnliche" Ereignisse auf dem Bildschirm lassen sich viele Fehler verhindern.

Oft ist es sinnvoll, dem Anwender eines Programms ein visuelles Feedback zu geben. Deshalb läßt GEM den Titel eines Menüs, aus dem ein Kommando ausgewählt wurde, invers auf dem Bildschirm stehen. Dieser Effekt zeigt dem Anwender, daß der Rechner arbeitet. Sie sollten diesen Titel daher erst nach Ausführung des Kommandos wieder normal darstellen.

Eine andere Art von optischem Feedback stellen die wachsenden Rechtecke dar, die beispielsweise beim Programmstart erscheinen. Sie verdeutlichen, daß ein bestimmter Eintrag "geöffnet" wurde. Obwohl diese Effekte die Leistung eines Programmes nicht beeinflussen, steigern sie die Anwenderfreundlichkeit und die Sicherheit in der Bedienung.

Einige Grundregeln zur Konstruktion von Menüs will ich daher im einzelnen erläutern:

So sollte zum Beispiel der zweite Menü-Titel immer "File" oder "Datei" lauten. Hier finden sich Kommandos zum Laden und Speichern von Daten und ein Kommando zum Verlassen des Programms. Es gibt einige Programme, die ein gesondertes "Verlasse"-Menü verwenden, in dem sich nur der Eintrag "Verlasse Programm" findet. Dies erscheint mir ziemlich unsinnig.

In einem Menü dürfen nicht zu viele Einträge stehen, weil dadurch Unübersichtlichkeit entsteht. Kommandos, die logisch zusammengehören sollten räumlich nicht voneinander getrennt werden.

Falls Sie relativ viele Kommandos haben, überlegen Sie sich, ob sie nicht besser in einer eigenen Dialogbox aufgehoben sind.

Zum Beispiel gab es in den ersten Version von *1stWord* ein Menü, mit dem sich Markierungen im Text setzen ließen. Alle vier möglichen Marken bekamen einen eigenen Eintrag im Menü. In den neuesten Programmversionen ist nur ein "Setze Marke"-Eintrag vorhanden, der eine Dialogbox auslöst, in der die jeweilige Marke ausgewählt wird.

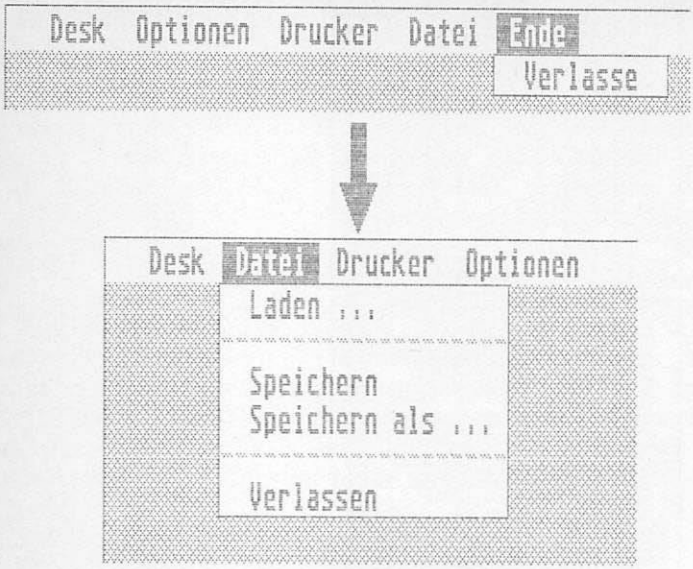


Bild 8.1: Sinnvollere Menüs

Folgt auf ein Kommando im Menü immer eine Dialogbox, macht es sich gut, an den Eintrag drei Punkte anzuhängen, um damit anzudeuten, daß noch eine weitere Auswahl getroffen werden muß. Im Beispiel auf der folgenden Seite müßte dann also "Setze Marke ..." im Menü stehen.

Es ist unbedingt nötig, in Menüs und Dialogboxen "gefährliche" von "ungefährlichen" Kommandos zu trennen. So empfiehlt sich das Löschkommando in einem Menü weit entfernt vom Ladekommando stehen. Nach Möglichkeit sind diese Optionen im Menü durch einen Strich zu trennen.

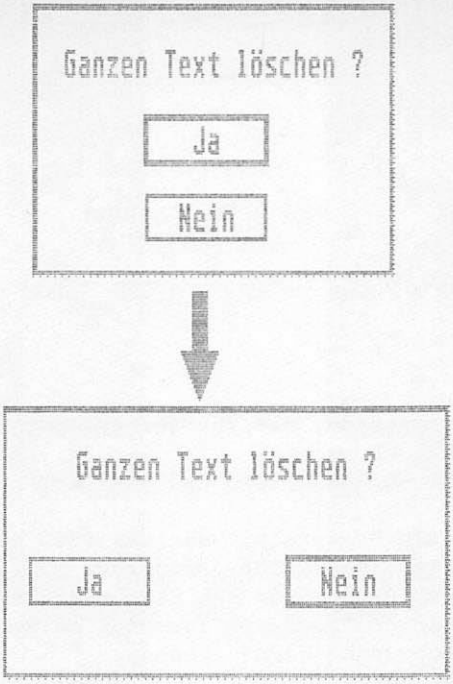


Bild 8.2: Gefährliche Auswahlen gut trennen

9. Beispielprogramme

Es folgt eine Reihe von Beispielprogrammen, die von Resources und den damit verbundenen Routinen Gebrauch machen. Ziel ist es, zu demonstrieren, welche Probleme auftauchen können und auf welche Feinheiten zu achten ist.

Die Programme dienen lediglich als Beispiele. das heißt, es handelt sich nicht um eigenständig lauffähige Programme, die besondere Aufgaben erfüllen könnten. Dies würde den Rahmen dieses Buches sprengen. Allerdings sind einige Programme durchaus weiter ausbaubar, so daß sie praktisch angewendet werden können.

Die Übertragung in andere Programmiersprachen erleichtern Ihnen die Kapitel 10 und 11 sowie der Anhang A. Für die Beispielprogramme habe ich, wie schon bemerkt, Modula-2 gewählt, da diese Sprache besonders gut lesbar ist. Auf spezielle Aspekte von Modula-2 - wie etwa Koroutinen oder das Modulkonzept - gehe ich nicht näher ein.

9.1 Die erste Dialogbox

Jetzt geht es darum, mit dem Resource-Editor eine erste Dialogbox zu konstruieren und in ein Programm einzubauen.
Zur Konstruktion der Resource *ERSTE* muß zunächst der Resource-Editor gestartet werden.

Dazu wird eine Dialogbox in die Resource hineingezogen und mit dem "NAME"-Kommandos *BAUM1* genannt. Auf dem Bildschirm sieht das so aus:

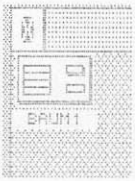


Bild 9.1.1: Der Resource-Baum

Öffnen Sie diesen Baum per Doppelklick mit der Maus, haben Sie zunächst ein leeres weißes Rechteck vor sich. Es dient als Hintergrund für die Box.

Ziehen Sie nun mit der Maus fünf String-Objekte in das Rechteck. Durch Öffnen dieser Objekte können Sie die Flags, den Status sowie den Text editieren. Bei den fünf Strings bleiben Flags und Status unbenutzt.

Schließlich sind noch zwei Button-Objekte nötig, die ebenfalls mit der Maus in die Box gesetzt werden können.

Durch Öffnen der Buttons kommen Sie wieder in den Editiermodus. Die Buttons sollen die Texte "JA" und "NEIN" enthalten. Beide sollen anwählbar sein und durch beide soll der Dialog verlassen werden können; also müssen Sie das *Selectable*- und das *Exit*-Flag setzen.

Der "JA"-Button soll mit der <Return>-Taste wählbar sein; also muß er zusätzlich das *Default*-Flag bekommen.

Die Box ist noch nicht ganz fertig - es bleiben die Namen für die Buttons zu vergeben. Den "JA"-Knopf nennen Sie *JABUTN*, den anderen *NEINBUTN*.

Wenn der Dialog folgendermaßen aussieht, kann die ganze Resource gespeichert werden.

Bild 9.1.2: Die Dialogbox

Das ganze Programm sieht folgendermaßen aus:

```
 1: MODULE ErsteBox ;
 2:
 3: (* erstes Programm zur Benutzung von Resources. Stellt eine
 4:    Ja-Nein-Frage und gibt darauf eine Antwort aus.            *)
 5:
 6: (* ADDRESS-Typ ist nicht "eingebaut"...                       *)
 7: FROM SYSTEM       IMPORT ADDRESS ;
 8:
 9: (* Alles Benötigte aus den Bibliotheken holen                 *)
10: FROM AESResources IMPORT ResourceLoad, ResourceGetAddr ;
11: FROM GEMAESbase   IMPORT AESCallResult, RTree, Object, Normal, Arrow,
12:                          FormStart, FormGrow, FormShrink, FormFinish ;
13: FROM AESForms     IMPORT FormAlert, FormCenter, FormDialogue, FormDo ;
14: FROM AESGraphics  IMPORT GrafMouse ;
15: FROM AESObjects   IMPORT ObjectDraw ;
16:
17: (* CONST-Definition wurden vom MMRCP erzeugt                  *)
18: CONST
19:      BAUM1 = 0 ;
20:     JABUTN = 5 ;
21:   NEINBUTN = 6 ;
```

```
22:
23: (* Texte für die Alerts                                          *)
24:   ErrorAlert = "[3][Fehler beim Laden][OK]" ;
25:   JaAlert    = "[0][Prima !][OK]" ;
26:   NeinAlert  = "[1][Das kann doch|nicht sein][OK]" ;
27:
28: VAR str : ARRAY [0..99] OF CHAR ;  (* entspricht einem String   *)
29:     Baum: ADDRESS;                 (* für die Resource-Addresse *)
30:     BaumFeld: POINTER TO           (* zum direkten Zugriff auf  *)
31:               ARRAY [0..10] OF Object; (* die Resource          *)
32:     result,                        (* für das Dialog-Ergebnis   *)
33:     x,y,w,h:INTEGER;               (* Clipping-Werte            *)
34:
35: BEGIN
36:   (* Mauszeiger als Pfeil *)
37:   GrafMouse(Arrow,NIL);
38:   (* Resource laden *)
39:   ResourceLoad('ERSTE.RSC');
40:   (* hat's geklappt ?? *)
41:   IF AESCallResult=0 THEN
42:     (* Nein -> anzeigen *)
43:     result:=FormAlert(1,ErrorAlert);
44:   ELSE
45:     (* Ja -> Adresse feststellen *)
46:     ResourceGetAddr(RTree,BAUM1,Baum);
47:     (* entspricht der Adresse des Feldes *)
48:     BaumFeld:=Baum;
49:     (* Box zentrieren *)
50:     FormCenter(Baum,x,y,w,h);
51:     (* Bildschirmbereich reservieren *)
52:     FormDialogue(FormStart,0,0,0,0,x,y,w,h);
53:     (* wachsendes Rechteck zeichnen *)
54:     FormDialogue(FormGrow,0,0,0,0,x,y,w,h);
55:     (* Box zeichnen *)
56:     ObjectDraw(Baum,BAUM1,5,x,y,w,h);
57:     (* Dialog durchführen *)
58:     result:=FormDo(Baum,-1);
59:     (* schrumpfendes Rechteck zeichnen *)
60:     FormDialogue(FormShrink,0,0,0,0,x,y,w,h);
61:     (* Bildschirmbereich freigeben *)
62:     FormDialogue(FormFinish,0,0,0,0,x,y,w,h);
63:     (* ausgewähltes Feld wieder normal setzen *)
64:     BaumFeld^[result].state:=Normal;
65:     (* auf Antwort reagieren *)
66:     CASE result OF
67:       (* Knopf "JA" -> "Prima"-Alert ausgeben *)
68:       JABUTN  : result:=FormAlert(1,JaAlert)      |
69:       (* Knopf "NEIN" -> "Kann nicht sein"-Alert ausgeben *)
```

```
70:       NEINBUTN: result:=FormAlert(1,NeinAlert)
71:     END;
72:   END;
73:   (* fertig ! *)
74: END ErsteBox.
```

Auf den Programmnamen folgt die Import-Liste. Bei Modula-2 Programmen wird dadurch der Zugriff auf externe Bibliotheken ermöglicht. "FROM AESResources IMPORT ResourceLoad" besagt, daß aus der Bibliothek *AESResources* die Prozedur *ResourceLoad* benutzt wird. Bei C-Programmen müssen Sie hier mit *extern* arbeiten und dann auch beim Link-Vorgang die entsprechende Bibliothek miteinbinden. (Modula-2 macht dies automatisch)

Der Resource-Editor erzeugt beim Abspeichern ein File *ERSTE.H*, in dem sich die Indizes für die Objekte befinden, die Namen erhalten haben. Sie sind direkt in den Programmtext zu übernehmen.

In den Zeilen 17-21 stehen die Indizes der benannten Objekte. Dieser Teil wurde direkt aus *ERSTE.H* (s.o.) eingelesen.

Dann werden drei String-Konstanten definiert (Zeilen 23-26). Das Programm wird einige *FormAlerts* benutzen und nimmt dafür diese Zeichenketten.

Die Variablen (Zeilen 28-33) sind im Listing beschrieben. Die wichtigsten davon sind *Baum* und *BaumFeld*. In Baum wird die Adresse der Resource gehalten und über *BaumFeld* kann der Benutzer auf die Resource direkt zugreifen.

Nun kann endlich das Programm anfangen. Nach einem Programmstart unter GEM hat der Mauszeiger die Form einer Biene. Da hier der normale Zeiger benutzt werden soll, muß dies GEM mit *GrafMouse* mitgeteilt werden (Zeile 37).

Mit *ResourceLoad* kann versucht werden, das *Resource*-File ERSTE.RSC einzulesen (Zeile 39). Ob dies geklappt hat, steht beim Modula-2-System in der Variablen *AESCallResult*. Bei C-Systemen ist dies das Ergebnis von *ResourceLoad*.

Ist dieser Wert Null, so trat ein Fehler auf, den das Programm in einer Alarmbox mitteilt (Zeile 43). Hier wird auch die Konstante *ErrorAlert* benutzt. Da das Laden nicht funktioniert hat und das Programm nicht ohne Resource arbeiten kann, werden keine weiteren Schritte mehr ausgeführt.

Wurde die Resource richtig geladen, läßt sich mit *ResourceGetAddr* die Adresse feststellen (Zeile 46). Diese Adresse ist gleichzeitig der Start der Feldes, über das auf die Resource zugegriffen werden soll, also bekommt auch *BaumFeld* diesen Wert.

Die Box soll in der Mitte des Bildschirms erscheinen, muß also von GEM zentriert werden (Zeile 50). In *x*, *y*, *w* und *h* steht nun der Bereich, der von der Resource belegt wird.

Jetzt wird der Dialog vorbereitet. Der erste Aufruf von *FormDialog* mit Parameter *FormStart* (Zeile 52) reserviert den in den letzten vier Parametern angegebenen Bereich auf dem Bildschirm.

Durch *FormDialog* und *FormGrow* als Parameter (Zeile 54) wird ein wachsendes Rechteck von der linken oberen Ecke (Koordinaten 0,0) bis auf die Größe und Position der Box gezeichnet.

Mit *ObjectDraw* wird die Dialogbox gezeichnet (Zeile 56). Dabei soll beim Objekt *BAUM1* gestartet werden.
Durch *FormDo* führt GEM den Dialog aus (Zeile 56). Da kein editierbares Objekt vorhanden ist, übergibt das Programm als zweiten Parameter -1. In *result* steht nun der Index des Objekts, über das der Dialog verlassen wurde.

FormDialogue (Zeile 60) mit *FormShrink* als Parameter zeichnet ein schrumpfendes Rechteck und der zweite Aufruf mit *FormFinish* (Zeile 62) gibt den reservierten Bildschirmbereich wieder frei. Der Dialog ist nun abgeschlossen.

Bei dem Objekt, über das der Dialog verlassen wurde, setzt GEM das *Selected*-Flag. Wollten Sie die Box ein zweites Mal verwenden, würde dieser Knopf invertiert gezeichnet. Deshalb wird dieses Flag gelöscht (Zeile 64), was über die Variable *BaumFeld* geschieht. Sie zeigt auf das Resource-Feld, deshalb kann ganz einfach *result* als Index verwendet werden.

Nun reagiert das Programm auf den ausgewählten Knopf. Wurde der "JA"-Knopf gedrückt (*result* ist *JABUTN*), erscheint eine Alarmbox mit dem Text aus *JaAlert*. Beim "NEIN"-Knopf zeigt es einen Alert mit *NeinAlert* an (Zeilen 68-70).

Damit ist das Programm beendet. Wie man sieht, ist die Programmierung mit Resources und Maussteuerung relativ einfach. In den nächsten Kapiteln werden Sie noch Gelegenheit erhalten, komplexere Beispiele zu programmieren.

9.2 Radio-Buttons

Das folgende Beispielprogramm demonstriert die Anwendung von Radio-Buttons in Dialogen. Diese Objekte sind immer dann zu verwenden, wenn aus mehreren Möglichkeiten immer nur genau eine ausgewählt werden soll.

Das Beispiel ist der Einstellung der seriellen RS232C-Schnittstelle mit dem "Emulator"-Accessory entlehnt. Ich beschränke mich aber auf die Auswahl der Baud-Rate in einer Dialogbox.

Die Resource soll wie folgt aussehen:

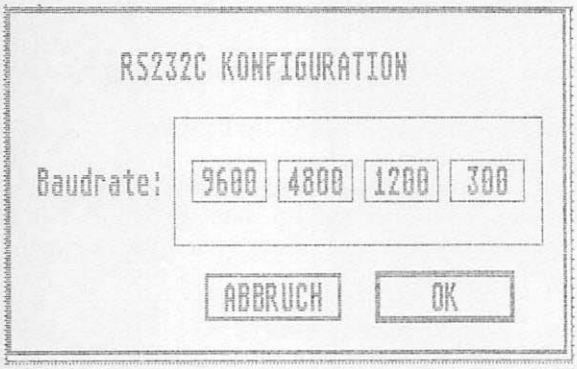


Bild 9.2.1: RS232C-Einstellung mit RadioButtons

Wichtig dabei ist, daß die vier Raten eine eigene Hintergrundbox haben. Bei allen vier muß das *Selectable*- und das *Radio-Button*-Flag gesetzt sein. Ihre Namen sind *BD300*, *BD1200*, *BD4800* und *BD9600*. Sie müssen beginnend mit dem "300"-Feld nacheinander in die Box gesetzt werden, damit die Indizes aufeinander folgen.

Die Buttons sind normale Knöpfe mit den Namen Abbruch und OK. Bei ihnen wird das *Exit*-Flag gesetzt, bei *OK* zusätzlich das *Default*-Flag.

Führt GEM mit dieser Box einen Dialog aus, so kann von den Radio-Buttons immer nur einer ausgewählt sein. Das "Abwählen" des alten Knopfes übernimmt GEM. Knöpfe, die sich gegenseitig ausschließen, müssen den gleichen Parent haben. Daher ist die Hintergrundbox notwendig.

Sind mehrere Einstellungen möglich, so müssen eben zwei Parent-Boxen vorhanden sein, in denen dann zwei Sätze von Radio-Buttons liegen.

Das kleine Programm sieht wie folgt aus (Die Zeilennummern dienen zur Orientierung und sind natürlich nicht Bestandteil des Programms!):

```
1: MODULE RadioButton;
2:
3: (* Benötigte Bibliotheksfunktionen anfordern *)
4: FROM    SYSTEM          IMPORT ADDRESS;
5: FROM    GEMAESbase      IMPORT Object, Arrow, RTree, AESCallResult,
6:                                FormStart, FormFinish, Normal, Selected ;
7: FROM    AESForms        IMPORT FormDo, FormCenter, FormDialogue, FormAlert ;
8: FROM    AESGraphics     IMPORT GrafMouse;
```

```
 9: FROM    AESObjects      IMPORT ObjectDraw, ObjectChange;
10: FROM    AESResources    IMPORT ResourceLoad, ResourceGetAddr ;
11:
12: (* Konstante vom MMRCP generiert *)
13: CONST
14:     RS232C = 0 ;
15:      BD300 = 4 ;
16:     BD1200 = 5 ;
17:     BD4800 = 6 ;
18:     BD9600 = 7 ;
19:         OK = 8 ;
20:    ABBRUCH = 9 ;
21:
22: VAR RSTree:ADDRESS;
23:     RSArray:POINTER TO ARRAY[0..100] OF Object;
24:     x,y,w,h,result:INTEGER;
25:     scan,BaudRate:INTEGER;
26:
27: BEGIN
28:   (* Resource laden *)
29:   ResourceLoad('RS232C.RSC');
30:   IF AESCallResult=0 THEN
31:     (* nicht gefunden *)
32:     result:=FormAlert(1,'[3][Kann RS232C.RSC nicht laden][OK]');
33:   ELSE
34:     (* Resourceadresse feststellen *)
35:     ResourceGetAddr(RTree,RS232C,RSTree);
36:     RSArray:=RSTree;
37:     (* Voreinstellung setzten *)
38:     BaudRate:=BD300;
39:     (* Button auswählen *)
40:     ObjectChange(RSTree,BaudRate,0,0,0,0,0,Selected,0);
41:     GrafMouse(Arrow,NIL);
42:     (* Box zentrieren *)
43:     FormCenter(RSTree,x,y,w,h);
44:     REPEAT
45:       (* Box zeichnen*)
46:       ObjectDraw(RSTree,RS232C,5,x,y,w,h);
47:       (* Dialog ausführen *)
48:       FormDialogue(FormStart,0,0,0,0,x,y,w,h);
49:       result:=FormDo(RSTree,0);
50:       FormDialogue(FormFinish,0,0,0,0,x,y,w,h);
51:       (* Ok oder ABBRUCH wieder normal setzen *)
52:       ObjectChange(RSTree,result,0,0,0,0,0,Normal,0);
53:       (* ausgewählte Rate suchen *)
54:       scan:=BD300;
55:       WHILE RSArray^[scan].state#Selected DO
56:         (* nächster Knopf *)
57:         INC(scan)
```

```
58:       END;
59:       (* Auswahl merken *)
60:       BaudRate:=scan;
61:     (* ... bis ABBRUCH-Knopf gedrückt *)
62:     UNTIL result=ABBRUCH;
63:   END;
64: END RadioButton.
```

Das schon bekannte Programmgerüst lädt und zentriert die Box in den Zeilen 29 bis 43. Anschließend wird in der REPEAT-Schleife ein Dialog mit der Box ausgeführt (Zeilen 46-50). Der Button, über den der Dialog verlassen wurde, wird wieder auf *Normal* gesetzt (Zeile 52).

Nun weiß das Programm aber noch nicht, welche Baudrate ausgewählt wurde (das Ergebnis von *FormDo* gibt lediglich an, über welches Objekt der Dialog verlassen wurde). Deshalb müssen die möglichen Einstellungen durchgesucht werden, bis sich das Objekt findet, das ausgewählt wurde (Zeilen 54-58). Damit diese Suche möglichst einfach ist, ist schon bei der Konstruktion der Resource darauf zu achten, daß die Objekte aufeinanderfolgende Indizes erhalten.

9.3 Einzelne Baumteile ausblenden

Dieses Programm demonstriert die Benutzung des *HideTree*-Flags. Ist dieses Flag bei einem Objekt gesetzt, so wird es bei einem *ObjectDraw* samt seinem Unterbaum nicht gezeichnet. Sie können damit zum Beispiel eine Box für mehrere Dialoge benutzen, in die sich - je nach Bedarf - Teile ein- und ausblenden lassen.

Die Beispiel-Resource könnte etwa in einem Malprogramm vorkommen. In einer Dialogbox sollen Farben für Flächen und Linien gesetzt werden können.

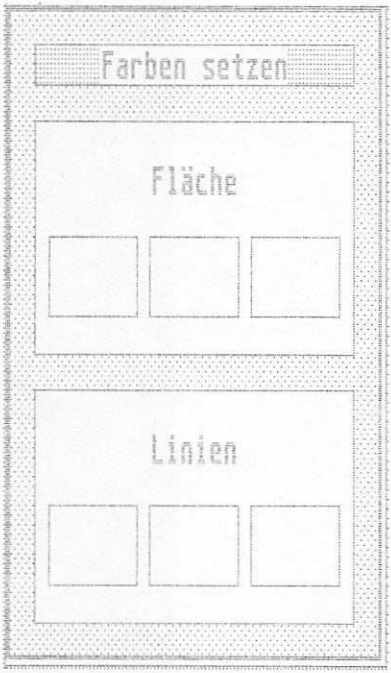


Bild 9.3.1: Die Dialogbox

Die jeweils drei Farbkästchen bekommen die Namen *LINIE1* bis *LINIE3* und *FLACHE1* bis *FLACHE3*. Die Hintergrundboxen der Farbfelder werden *LINIE* und *FLACHE* genannt. Bei allen sechs Farbboxen muß das *Exit*- und das *Selectable*-Flag gesetzt sein.

Die hypothetische Situation sei, daß einmal nur die Linienfarbe gesetzt werden soll und ein anderes Mal nur die Flächenfarbe. Damit nicht zwei Dialog-Boxen benötig werden, kommt das *HideTree*-Flag zum Einsatz. Das folgende Programm zeigt dies am Beispiel.

```
 1: MODULE HideTreeDemo;
 2:
 3: (* Benötigte Bibliotheksfunktionen anfordern *)
 4: FROM    SYSTEM          IMPORT ADDRESS;
 5: FROM    GEMAESbase      IMPORT Object, Arrow, RTree, AESCallResult,
 6:                                FormStart, FormFinish, HideTree, Normal ;
 7: FROM    AESForms        IMPORT FormDo, FormCenter, FormDialogue, FormAlert ;
 8: FROM    AESGraphics     IMPORT GrafMouse;
 9: FROM    AESObjects      IMPORT ObjectDraw, ObjectChange ;
10: FROM    AESResources    IMPORT ResourceLoad, ResourceGetAddr ;
11:
12: (* Konstante vom MMRCP generiert *)
13: CONST
14:     FARBEN = 0 ;
15:      LINIE = 2 ;
16:     LINIE1 = 3 ;
17:     LINIE2 = 4 ;
18:     LINIE3 = 5 ;
19:     FLACHE = 7 ;
20:    FLACHE1 = 9 ;
21:    FLACHE2 = 10 ;
22:    FLACHE3 = 11 ;
23:
24: VAR ColorTree:ADDRESS;
25:     ColorArray:POINTER TO ARRAY[0..100] OF Object;
26:     x,y,w,h,result:INTEGER;
27:
28: PROCEDURE DoColor;
29: BEGIN
30:   (* Box zeichnen*)
31:   ObjectDraw(ColorTree,FARBEN,5,x,y,w,h);
32:   (* Dialog ausführen *)
33:   FormDialogue(FormStart,0,0,0,0,x,y,w,h);
34:   result:=FormDo(ColorTree,0);
35:   FormDialogue(FormFinish,0,0,0,0,x,y,w,h);
36:   (* ausgewähltes Feld wieder normal *)
37:   ObjectChange(ColorTree,result,0,0,0,0,0,Normal,0);
38: END DoColor;
39:
40: BEGIN
41:   (* Resource laden *)
```

```
42:   ResourceLoad('COLOR.RSC');
43:   IF AESCallResult=0 THEN
44:     (* nicht gefunden *)
45:     result:=FormAlert(1,'[3][Kann COLOR.RSC nicht laden][OK]');
46:   ELSE
47:     (* Resourceadresse feststellen *)
48:     ResourceGetAddr(RTree,FARBEN,ColorTree);
49:     ColorArray:=ColorTree;
50:     (* Button auswählen *)
51:     GrafMouse(Arrow,NIL);
52:     (* Box zentrieren *)
53:     FormCenter(ColorTree,x,y,w,h);
54:     (* ganz zeichnen *)
55:     DoColor;
56:     (* Linienfarben ausblenden *)
57:     ColorArray^[.flags:=ColorArray^[LINIE].flags+HideTree;
58:       DoColor;
59:       (* Linienfarben einblenden *)
60:       ColorArray^[LINIE].flags:=ColorArray^[LINIE].flags-HideTree;
61:         (* Flächenfarben ausblenden *)
62:       Color-Array^[FLACHE].flags:=ColorArray^[FLACHE].flags+HideTree
63:       DoColor;
64:   END;
65: END HideTreeDemo.
```

Zuerst lädt das Programm die Resource und zentriert sie (Zeilen 41-53). Der erste Aufruf von *DoColor* führt einen Dialog durch. In Zeile 57 wird das *HideTree*-Flag für das Objekt LINIE gesetzt. Die Box mit den Linienfeldern wird dadurch ausgeblendet. Dies zeigt sich beim zweiten Aufruf von *DoColor*.

In Zeile 59 werden die Objekte wieder eingeblendet und die Box für Flächenfarben ausgeblendet. Der letzte Aufruf von *DoColor* (Zeile 63) läßt die Box entsprechend anders aussehen.

Es sieht natürlich unschön aus, daß jeweils ein großer Teil der Box leer bleibt. Dieses Problem ließe sich umgehen, wenn Linien- und Flächenbox die gleichen Koordinaten belegten. So gäbe es keine leeren Bereiche und die Box wäre mehrfach verwendbar.

Ausblenden einzelner Editierfelder oder Buttons wäre eine Alternative zum Disablen diesr Objekte.

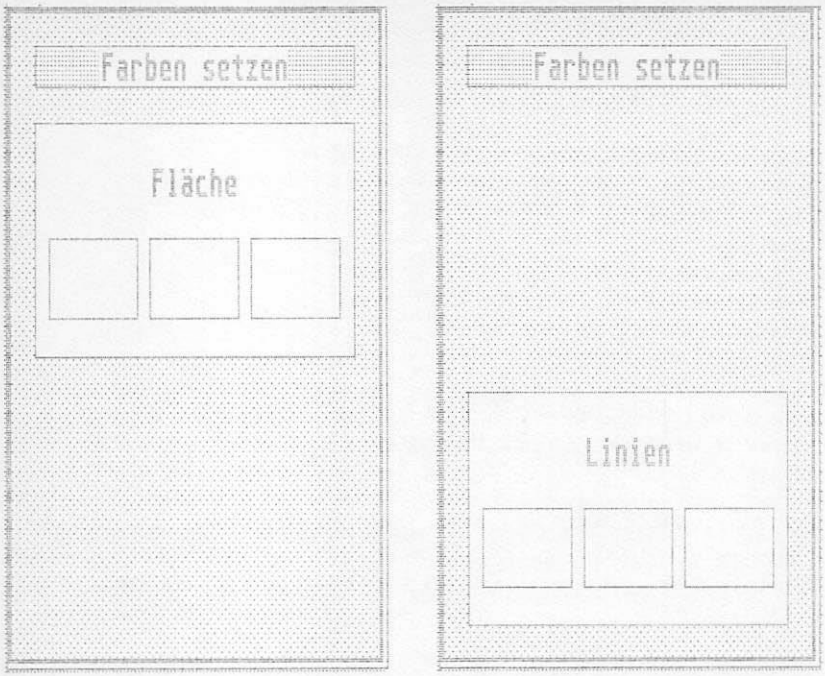


Bild 9.3.2: Die Boxen, die durch Ausblenden entstehen

9.4 Eigene Menüs

In diesem Kapitel geht es um Drop-Down-Menüs. Zunächst muß dazu die Resource gebaut werden. Sie besteht aus zwei Bäumen. Der erste Baum ist vom Menü-Typ und enthält das eigentliche Menü.

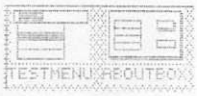


Bild 9.4.1: Die Bäume

Das Menü hat zwei Menü-Titel sowie den Standard-"Desk"-Eintrag. Die "Desk Accessory x"-Einträge im "Desk"-Menü werden vom Recource-Editor vorgegeben und sollten nicht verändert werden. Beim Laden der Resource fügt GEM an diese Stellen automatisch die Namen der Accessories ein.

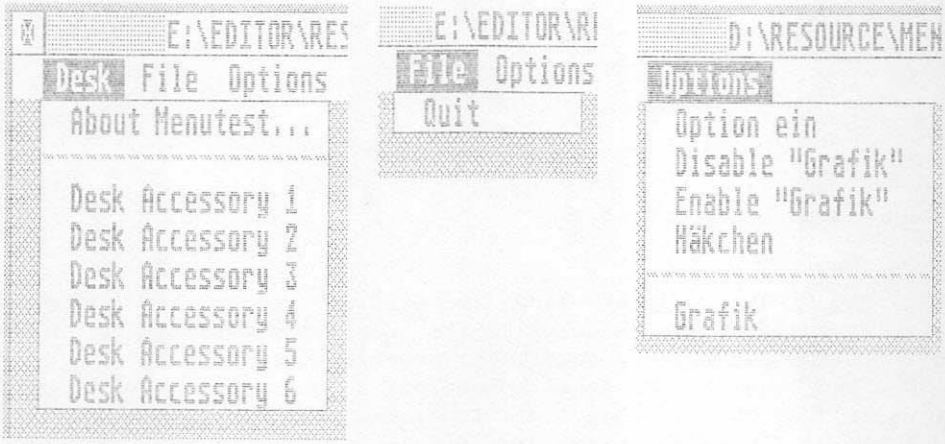


Bild 9.4.2: Die Menüs

Das "File"-Menü dient zum Verlassen des Programms und hat daher nur einen Eintrag. Im "Option"-Menü stehen einige Einträge, an denen sich verschiedene Routinen des Menü-Managers ausprobieren lassen.

Bei der Erstellung der Resource ist darauf zu achten, daß die Breite der Einträge der Hintergrundbox entspricht. Beachten Sie dies nicht, so wird beim Programmlauf nur ein Teil des Eintrags invertiert. Wird der weiße Hintergrundbereich angeklickt, führt dies zumeist zu einem Programmfehler.

Bild 9.4.3: Die Einträge müssen breit genug sein

Der zweite Baum enthält eine kleine Box, die bei Auswahl des "About Menutest ..."-Eintrags angezeigt werden soll:

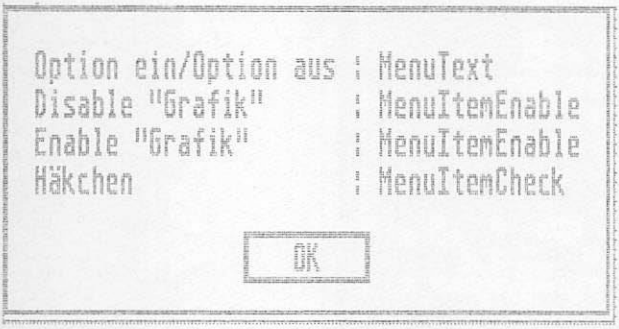


Bild 9.4.4: Die Info-Box

Die Namen der Objekte sind im Listing beschrieben. Dieser Listing-Teil muß vom Resource-Editor übernommen werden, da sich die Indizes der Objekte natürlich verschieben können.

Nun zum Programm: Zunächst kommen die schon bekannten Daten zum Benutzen der GEM-Routinen, die Konstanten für die Objekt-Indizes und eine Routine zum Laden der Resource, die auch die Adressen der beiden Bäume feststellt (Zeilen 40-62). Mit *MenuBar* schaltet die Routine die Menüzeile ein. GEM weiß nun, mit welchen Menüs es zu arbeiten hat.

Das Hauptprogramm (ab Zeile 120) setzt nach dem Laden der Resource einige Voreinstellungen (Zeilen 125-129), die per Menü verändert werden sollen.

Dann werden die Strings *ein* und *aus* gesetzt und anschließend *MenuText* aufgerufen (Zeilen 131-133). Damit ist der Inhalt des Menu-Eintrages *EINAUS* festgelegt. GEM übernimmt den Inhalt des angegebenen Strings in die Resource und zeigt ihn beim nächsten "herunterklappen" des Menüs an.

Nun muß die Kontrolle an GEM abgegeben werden. Dies geschieht mit *EventMessage* (Zeile 136). Das Programm erwartet daraufhin eine Mitteilung von GEM. Dies könnte eine Meldung zum Neuzeichnen des Bildschirms sein, oder die Mitteilung, daß ein Menü ausgewählt wurde.

GEM übernimmt bei Aufruf von *EventMessage* vollständig das Überwachen des Mauszeigers, das "Herunterklappen" der Menüs und das Invertieren der Menüeinträge. Wird ein Eintrag durch Anklicken ausgewählt, klappt das Menü wieder hoch und GEM schickt dem Programm die Mitteilung, daß etwas ausgewählt wurde.

Welcher Art die Mitteilung ist, steht im ersten Wert des bei *EventMessage* angegebenen Mitteilungs-Puffers. Ist es eine *MenuSelected* Mitteilung, so stehen im vierten und fünften Wert die Indizes des ausgewählten Menu-Titels und -Eintrags.

Diese beiden Werte werden an die Routine *DoMenu* weitergegeben (Zeile 139). Hier reagiert das Programm mit einer geschachtelten Fallunterscheidung auf die Auswahl.

Bei Auswahl von "About Menutest ...", also beim Menü-Titel *DESKMENU* und dem Eintrag *ABOUT* zeigt das Programm schon die About-Box an (Zeilen 69-78).

Wurde "Quit" angewählt, wird die Variable *done* gesetzt und die Schleife im Hauptprogramm verlassen (Zeilen 79-82).

Beim "Options"-Menü ist die Auswahl reichhaltiger. Der Eintrag *EINAUS* soll bei jeder Anwahl seinen Text ändern. Dazu wird die *MenuText*-Routine genutzt. Damit das Programm weiß, welcher Text gerade angezeigt wird, ist hier eine Hilfsvariable nötig, die sich hier im Record *info* befindet (Zeilen 84-89).

Durch *DISABLE* soll der Eintrag "Grafik" ausgeschaltet werden. Dazu gibt es die Routine *MenuItemEnable*, die den Baum und den Eintrag als Parameter erhält. Der letzte Parameter entscheidet darüber, ob der Eintrag an- oder ausgeschaltet werden soll. Schließlich wird der Zustand des "Grafik"-Eintrags wieder in einer Hilfsvariablen vermerkt (Zeilen 90-92).

ENABLE leistet das Gegenteil von *DISABLE*; es schaltet den "Grafik"-Eintrag wieder an. Dies geschieht analog zum Ausschalten mit *MenuItemEnable* (Zeilen 93-95).

Mit *HAEKCHEN* wird vor diesem Menü-Eintrag ein kleines Häkchen ein- und ausgeschaltet. Die verwendete Routine hierfür ist *MenuItemCheck*. Sie hat die gleichen Parameter wie *MenuItemEnable*. Der Zustand des Häkchens wird wieder in einer Hilfsvariablen vermerkt (Zeilen 96-102).

GRAFIK dient nur zur Demonstration des Ein- und Ausschaltens von Einträgen und löst keine Funktion aus.

Nachdem das Programm auf ein Menü reagiert hat, bleibt noch etwas zu tun: GEM bringt den Titel des Menüs nämlich invers auf den Bildschirm. Er muß also wieder normal gezeichnet werden. Dazu dient der Aufruf von *MenuTitleNormal* mit Angabe des entsprechenden Baumes und Titel. Die 1 als letzter Parameter besagt, daß der Titel normal gezeichnet werden soll (Zeile 108).

```
 1: MODULE MenuTest;
 2:
 3: (* Standard modules     *)
 4: FROM SYSTEM             IMPORT ADDRESS, ADR ;
 5: (* GEM modules          *)
 6: FROM AESApplications    IMPORT ApplInitialise ;
 7: FROM GEMAESbase         IMPORT RTree, AESCallResult,
 8:                                Arrow, MouseOn, MouseOff,
 9:                                MenuSelected,
10:                                FormStart, FormFinish,
11:                                Normal ;
12: FROM AESEvents          IMPORT EventMessage ;
13: FROM AESForms           IMPORT FormAlert, FormDialogue, FormDo, FormCenter ;
14: FROM AESMenus           IMPORT MenuBar, MenuItemCheck,
15:                                MenuItemEnable, MenuTitleNormal,
16:                                MenuText ;
17: FROM AESResources       IMPORT ResourceGetAddr, ResourceLoad ;
18: FROM AESGraphics        IMPORT GrafMouse ;
19: FROM AESObjects         IMPORT ObjectDraw, ObjectChange ;
20:
21: (* Konstante vom MMRCP geliefert *)
22: CONST
23:   ABOUTBOX = 1 ;    (* About-Dialog *)
24:    ABOUTOK = 8 ;    (* OK-Button dabei *)
25:   TESTMENU = 0 ;    (* Menübaum *)
26:   DESKMENU = 3 ;    (* "Desk"-Titel *)
27:   FILEMENU = 4 ;    (* "File"-Titel *)
28:    OPTMENU = 5 ;    (* "Options"-Titel *)
29:      ABOUT = 8 ;    (* "About Menutest"-Eintrag *)
30:       QUIT = 17 ;   (* "Quit"-Eintrag *)
31:     EINAUS = 19 ;   (* "Option ein/aus"-Eintrag *)
32:    DISABLE = 20 ;   (* "Disable 'Grafik'"-Eintrag *)
33:     ENABLE = 21 ;   (* "Enable 'Grafik'"-Eintrag *)
34:   HACKCHEN = 22 ;   (* "Häckchen"-Eintrag *)
35:     GRAFIK = 24 ;   (* "Grafik"-Eintrag *)
36:
37: VAR MenuTree, AboutTree :ADDRESS ;
38:     Appl : INTEGER ;
39:
```

```
40: (* Resource laden und anzeigen *)
41: PROCEDURE Init() : BOOLEAN ;
42: CONST RSCFileName = 'MENUTEST.RSC' ;
43:       AlertText  = "[3][Cannot load MENUTEST][OK]" ;
44:
45: VAR str: ARRAY [0..99] OF CHAR ;
46:     dummy: INTEGER ;
47:
48: BEGIN
49:   Appl:= ApplInitialise() ;
50:   str:=RSCFileName ;
51:   ResourceLoad(str) ;
52:   IF AESCallResult=0 THEN
53:     str:=AlertText ;
54:     dummy:=FormAlert(1,str) ;
55:     RETURN FALSE
56:   END ;
57:   ResourceGetAddr(RTree,TESTMENU,MenuTree) ;
58:   ResourceGetAddr(RTree,ABOUTBOX ,AboutTree) ;
59:   (* Neue Menüzeile anzeigen *)
60:   MenuBar(MenuTree,1) ;
61:   RETURN TRUE
62: END Init ;
63:
64: (* Eintrag "item" im Menü "title" verarbeiten *)
65: PROCEDURE DoMenu (title,item:INTEGER) ;
66: VAR x,y,w,h,dummy:INTEGER;
67: BEGIN
68:   CASE title OF
69:     DESKMENU : IF item = ABOUT THEN
70:                  (* "About"-Box anzeigen *)
71:                  FormCenter(AboutTree,x,y,w,h) ;
72:                  FormDialogue(FormStart,0,0,0,0,x,y,w,h) ;
73:                  ObjectDraw(AboutTree,0,5,x,y,w,h) ;
74:                  dummy:=FormDo(AboutTree,0) ;
75:                  FormDialogue(FormFinish,0,0,0,0,x,y,w,h) ;
76:                  ObjectChange(AboutTree,ABOUTOK,0,0,0,0,0,
77:                               Normal,0) ;
78:                END                                  |
79:     FILEMENU : IF item = QUIT THEN
80:                  (* Verlassen ausgewählt *)
81:                  done:=TRUE
82:                END                                  |
83:     OPTMENU  : CASE item OF
84:                  EINAUS  : (* Je nach Stellung Eintrag verändern *)
85:                            CASE info.ein OF
86:                              TRUE : MenuText(MenuTree,EINAUS,ADR(aus)) |
87:                              FALSE: MenuText(MenuTree,EINAUS,ADR(ein))
88:                            END;
```

```
 89:                          info.ein:=NOT info.ein;      |
 90:                 DISABLE : (* Eintrag "Grafik" disablen *)
 91:                          MenuItemEnable(MenuTree,GRAFIK,0);
 92:                          info.grafik:=FALSE;          |
 93:                 ENABLE  : (* Eintrag "Grafik" enablen *)
 94:                          MenuItemEnable(MenuTree,GRAFIK,1);
 95:                          info.grafik:=TRUE;           |
 96:                 HACKCHEN: (* Je nach Stellung Häkchen setzen oder
 97:                              löschen *)
 98:                          CASE info.hackchen OF
 99:                            TRUE : MenuItemCheck(MenuTree,HACKCHEN,0) |
100:                            FALSE: MenuItemCheck(MenuTree,HACKCHEN,1)
101:                          END;
102:                          info.hackchen:=NOT info.hackchen;    |
103:                 GRAFIK  : (* bewirkt nichts ... *)
104:               ELSE
105:               END
106:   END ;
107:   (* ausgewählten Menütitel wieder normal anzeigen *)
108:   MenuTitleNormal(MenuTree,title,1) ;
109: END DoMenu ;
110:
111: VAR buff: ARRAY[0..9] OF INTEGER ;
112:     done: BOOLEAN ;
113:     info: RECORD
114:             ein,
115:             hackchen,
116:             grafik      :BOOLEAN
117:           END;
118:     ein,aus:ARRAY [0..20] OF CHAR;
119:
120: BEGIN
121:   IF Init() THEN
122:     GrafMouse(Arrow,NIL);
123:     done:=FALSE;
124:     (* Voreinstellungen *)
125:     WITH info DO
126:       ein:=TRUE;
127:       hackchen:=FALSE;
128:       grafik:=TRUE;
129:     END;
130:     (* Text setzen *)
131:     ein:='  Option ein';
132:     aus:='  Option aus';
133:     MenuText(MenuTree,EINAUS,ADR(ein));
134:     REPEAT
135:       (* Auf Message warten ... *)
136:       EventMessage(ADR(buff));
137:       (* Bei einer Menu-Message auf Titel und Eintrag reagieren *)
```

```
138:      CASE buff[0] OF
139:        MenuSelected : DoMenu(buff[3],buff[4])          |
140:      ELSE
141:        (* auf andere Messages nicht reagieren *)
142:      END;
143:    UNTIL done;
144:    (* fertig *)
145:    END;
146: END MenuTest.
```

Dieses Programm kann als Grundgerüst für jedes andere dienen, das Menüs benutzt. Es setzt sich immer aus dem Warten auf eine Mitteilung, daß ein Menü ausgewählt wurde und einer geschachtelte Fallunterscheidung, in der auf das jeweilige Kommando reagiert wird, zusammen.

Es besteht natürlich die Möglichkeit, auf die Hilfsvariablen zu verzichten und den Status eines Eintrags (z.B. Häkchen oder nicht) direkt aus der Resource zu holen. Dieses Vorgehen ist jedoch erheblich langsamer. Und den Platz für ein paar boolsche Variablen hat man sicher.

9.5 Files komfortabel auswählen

Das Programm in diesem Kapitel soll als Beispiel für einen etwas komplexeren Dialog - eine Box zum Auswählen von Files - dienen. Die Idee bei dieser Box ist, daß im Gegensatz zur Standard-GEM-Box nur die Maus benutzt wird. Das Umschalten auf andere Laufwerke ist durch Buttons möglich. Solche Boxen werden auch in den Megamax-Produkten, zum Beispiel im MMRCP, angewendet.

Zunächst muß natürlich wieder die Resource konstruiert werden:

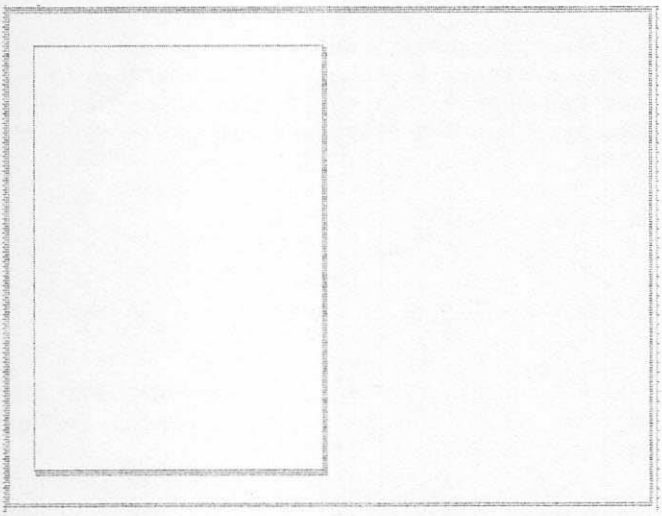


Bild 9.5.1: Die Fileselect-Resource am Anfang

In eine große Box wird eine kleinere plaziert, die aus optischen Gründen das *Shadowed*-Flag gesetzt hat.

In sie kommt eine kleinere Box hinein, in die wiederum zehn String-Objekte gesetzt werden, die später die Dateinamen aufnehmen sollen. Wenn das Programm eine neue Namensliste anzeigt, benutzt es diese Box als Startobjekt für den *ObjectDraw*-Aufruf.

Die kleinere Box erhält den Namen *BACKGRND*, und die String-Objekte heißen *LINE0* bis *LINE9*. Sie müssen nacheinander eingesetzt werden, damit ihre Indizes aufeinander folgen.

Das kleine Pseudo-Fenster wird (siehe Bild 9.5.3) noch durch eine Titelzeile, einen Rollbalken und ein "Pfeil hoch"- und "Pfeil runter"-Kästchen (Dies sind natürlich Objekte vom Type *GraphicBoxChar*.) vervollständigt.

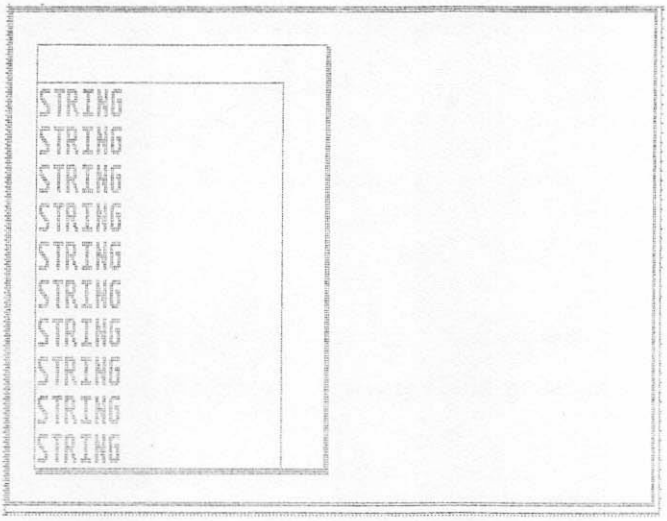


Bild 9.5.2: Die Resource mit Einträgen für die Dateinamen

Die Rollbox, die im Rollbalken plaziert wird, heißt *SCROLBOX*. Die Strichlinie (ein String-Objekt) dient zum Aufnehmen einer Informationszeile und heißt demzufolge *INFOLINE*.

Die Buttons "A:" bis "F:" (zum Umschalten auf ein anderes Laufwerk) werden *DRIVEA* bis *DRIVEF* genannt. Die beiden Buttons heißen *OK* und *CANCEL*.

Bei allen Objekten, die Namen bekommen, muß das *Selectable*- und das *Exit*-Flag gesetzt sein. Beim *OK*-Button kommt noch das *Default*-Flag hinzu.

Die Titelzeile erhält den Namen *NAMELINE*, der Rollbalken heißt *SCROLBAR* und die Kästchen *LINEUP* und *LINEDOWN*.

Es können jetzt die restlichen Objekte eingefügt werden - und wie Sie auf der nächsten Seite sehen, ist die Box damit komplett.

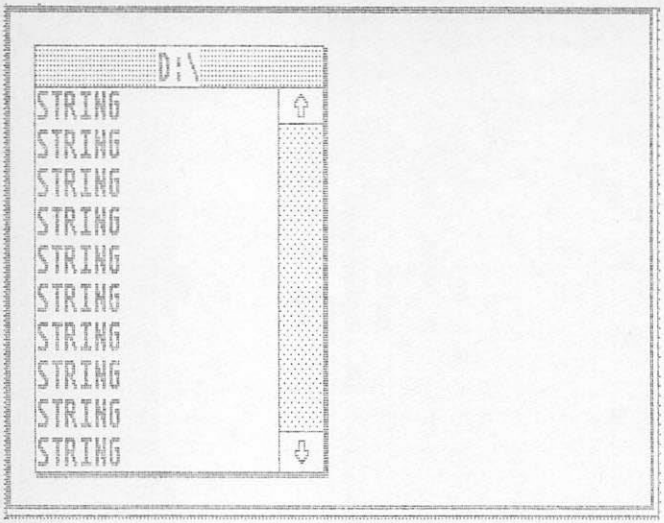


Bild 9.5.3: Jetzt sind auch Rollbalken, Titelzeile und Pfeile vorhanden.

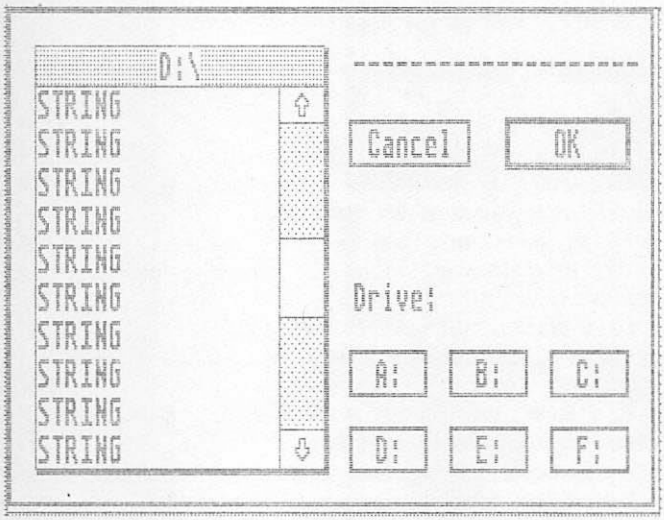


Bild 9.5.4: Die komplette Box

Nun zu einer Beschreibung der beabsichtigten Funktionen der einzelnen Elemente des Dialogs: Das simulierte "Fenster" dient zur Aufnahme von jeweils zehn Dateinamen aus dem aktuellen

Directory. Beim Anklicken eines Namens muß eventuell ein anderes Directory eingelesen werden. Dessen Name soll in der Namenszeile erscheinen.

Mit den Hoch- und Runterpfeilen soll jeweils eine Zeile im "Fenster" gescrollt werden. Klicken auf den Rollbalken bewirkt ein Scrollen um zehn Zeilen. Mit der Rollbox kann direkt positioniert werden.

Die sechs Laufwerksknöpfe dienen zum Umschalten auf ein anderes Laufwerk (Bei der Standard-GEM-Box muß man dazu zur Tastatur greifen!). Mit "OK" oder "ABBRUCH" wird die Auswahl beendet.

Zusätzlich kann in der Infozeile ein Text dargestellt werden, aus dem hervorgeht, ob die ausgewählte Datei geladen, gelöscht oder gespeichert werden soll.

Das Programm-Listing sieht wie folgt aus. Es benutzt zusätzlich zu den GEM-Funktionen einige GEMDOS- und VDI-Routinen.

```
 1: MODULE FileSelect ;
 2:
 3: (* Standard module     *)
 4: FROM SYSTEM            IMPORT ADDRESS, BYTE, ADR ;
 5: FROM Strings           IMPORT Concat, Compare, CompareResults, String,
 6:                               Length, Delete, Insert, InitStringModule,
 7:                               Assign ;
 8: (* GEM module         *)
 9: FROM GEMDOS            IMPORT GetDTA, SFirst, SNext;
10: FROM GEMAESbase        IMPORT RTree, AESCallResult, Object, TEdInfo,
11:                               Selected, Arrow, MouseOn, MouseOff,
12:                               FormStart, FormFinish ;
13: FROM AESApplications   IMPORT ApplInitialise, ApplExit ;
14: FROM AESForms          IMPORT FormAlert, FormCenter, FormDialogue, FormDo ;
15: FROM AESGraphics       IMPORT GrafMouse, GrafSlideBox,
16:                               GrafMouseKeyboardState ;
17: FROM AESMenus          IMPORT MenuText, MenuTitleNormal ;
18: FROM AESObjects        IMPORT ObjectDraw, ObjectOffset ;
19: FROM AESResources      IMPORT ResourceGetAddr, ResourceLoad ;
20: FROM GEMVDIbase        IMPORT BigPxyArrayType, VDIWorkInType,
21:                               VDIWorkOutType;
22: FROM VDIControls       IMPORT OpenVirtualWorkstation,
23:                               CloseVirtualWorkstation;
24: FROM VDIRasters        IMPORT MFDBType, CopyRasterOpaque;
25: FROM XBIOS             IMPORT ScreenPhysicalBase;
26:
27: (* Konstante vom MMRCP geliefert *)
28: CONST
29:   FILESLCT = 0 ;
30:   BACKGRND = 20 ;       (* Hintergrund der Namen *)
31:      LINE0 = 3 ;        (* die 10 Dateinamen *)
32:      LINE1 = 4 ;
```

```
33:      LINE2 = 5 ;
34:      LINE3 = 6 ;
35:      LINE4 = 7 ;
36:      LINE5 = 8 ;
37:      LINE6 = 9 ;
38:      LINE7 = 10 ;
39:      LINE8 = 11 ;
40:      LINE9 = 12 ;
41:     LINEUP = 2 ;        (* Pfeil hoch *)
42:   LINEDOWN = 13 ;       (* Pfeil runter *)
43:   NAMELINE = 16 ;       (* Titelzeile *)
44:   INFOLINE = 17 ;       (* Infozeile *)
45:   SCROLBOX = 15 ;       (* Rollbox *)
46:   SCROLBAR = 14 ;       (* Rollbalken *)
47:         OK = 19 ;       (* OK-Knopf *)
48:     CANCEL = 18 ;       (* ABBRUCH-Knopf *)
49:     DRIVEA = 21 ;       (* die Knöpfe zum Umschalten *)
50:     DRIVEB = 23 ;       (* des Laufwerks *)
51:     DRIVEC = 25 ;
52:     DRIVED = 22 ;
53:     DRIVEE = 24 ;
54:     DRIVEF = 26 ;
55:
56: TYPE FilenameType = ARRAY [0..13] OF CHAR;
57:      (* Informationen, die von SFirst und SNext geliefert werden *)
58:      DTABuffer    = RECORD
59:                       reserved: ARRAY [0..20] OF BYTE;
60:                       attrib  : BYTE;
61:                       time,date:CARDINAL;
62:                       size    : LONGCARD;
63:                       name    : FilenameType;
64:                     END;
65:
66: VAR FileTree          : ADDRESS ;
67:     Appl, Handle      : INTEGER ;
68:     TreeArray         : POINTER TO ARRAY [0..40] OF Object ;
69:     cancel            : BOOLEAN;
70:     thepath, themask,
71:     theinfo, filename : String;
72:
73: (* Resource laden und Adresse feststellen *)
74: PROCEDURE Init() : BOOLEAN ;
75: CONST RSCFileName = 'FILESLCT.RSC' ;
76:       AlertText  = "[3][Cannot load FILESLCT.RSC][OK]" ;
77:
78: VAR str      : ARRAY [0..99] OF CHAR ;
79:     dummy, i : INTEGER ;
80:     workin   : VDIWorkInType;
81:     workout  : VDIWorkOutType;
```

```
 82:
 83: BEGIN
 84:   InitStringModule;
 85:   (* Programm anmelden *)
 86:   Appl:= ApplInitialise() ;
 87:   str:=RSCFileName ;
 88:   ResourceLoad(str) ;
 89:   IF AESCallResult=0 THEN
 90:     str:=AlertText ;
 91:     dummy:=FormAlert(1,str) ;
 92:     RETURN FALSE
 93:   END ;
 94:   (* Adresse feststellen *)
 95:   ResourceGetAddr(RTree,FILESLCT,FileTree) ;
 96:   TreeArray:=FileTree;
 97:   (* kleine Korrekturen an der Resource, die mit dem MMRCP schlecht
 98:      machbar sind *)
 99:   DEC(TreeArray^[BACKGRND].width);
100:   FOR i:=LINE0 TO LINE9 DO
101:     TreeArray^[i].width:=TreeArray^[BACKGRND].width
102:   END;
103:   (* Bildschirm öffnen *)
104:   FOR i:=0 TO 9 DO
105:     workin[i]:=1;
106:   END;
107:   workin[10]:=2;
108:   OpenVirtualWorkstation(workin,Handle,workout);
109:   RETURN TRUE
110: END Init ;
111:
112: PROCEDURE FileSelect(VAR Path, mask, infoline, filename : String;
113:                      VAR cancel:BOOLEAN);
114: TYPE FileType = RECORD
115:                   name  :ARRAY[0..15] OF CHAR;
116:                   attrib:CARDINAL;
117:                 END;
118: VAR x, y, w, h, my, sy, dummy, scan, result,
119:     Fileno, SelectedFile, startline,
120:     fileselected                    : INTEGER;
121:     newpos                          : REAL;
122:     Files                           : ARRAY [0..200] OF FileType;
123:     dirname, empty                  : ARRAY [0..13] OF CHAR;
124:     path                            : String;
125:
126: (* Directory mit Namen "Path" in das Files-Feld einlesen und
127:    sortieren *)
128: PROCEDURE ReadDir;
129:
130: VAR  DTA                    : POINTER TO DTABuffer;
```

```
131:      name                    : ARRAY [0..80] OF CHAR;
132:      oldfileno, i, up, down : INTEGER;
133:      help                    : FileType;
134:      smaller                 : BOOLEAN;
135:      TedPointer              : POINTER TO TEdInfo;
136: BEGIN
137:   GrafMouse(MouseOff,NIL);
138:   (* Directory-Namen in der Kopfzeile anzeigen *)
139:   TedPointer:=TreeArray^[NAMELINE].spec;
140:   TedPointer^.ptext:=ADR(path);
141:   ObjectDraw(FileTree,NAMELINE,1,x,y,w,h);
142:   GrafMouse(MouseOn,NIL);
143:   oldfileno:=Fileno;
144:   Fileno:=0;
145:   (* Suchstring erstellen *)
146:   Concat(path,mask,name);
147:   (* Filenamen einlesen *)
148:   SFirst(name,12H,result);
149:   WHILE result=0 DO
150:     GetDTA(DTA);
151:     (* Attribut kopieren *)
152:     Files[Fileno].attrib:=CARDINAL(DTA^.attrib);
153:     (* bei Ordnern ist das erste Zeichen das "Ordner-Zeichen" *)
154:     IF ODD(Files[Fileno].attrib DIV 10H) THEN
155:       Files[Fileno].name[0]:=7C
156:     ELSE
157:       Files[Fileno].name[0]:=' '
158:     END;
159:     (* Leerstelle *)
160:     Files[Fileno].name[1]:=' ';
161:     (* Namen kopieren *)
162:     FOR i:=0 TO 13 DO
163:       Files[Fileno].name[i+2]:=DTA^.name[i]
164:     END;
165:     INC(Fileno);
166:     SNext(result)
167:   END;
168:   (* alle eventuell unbenutzten Einträge löschen *)
169:   FOR i:=Fileno TO oldfileno DO
170:     Files[i].name:='';
171:     Files[i].attrib:=0;
172:   END;
173:   (* Files aufsteigend nach dem Namen sortieren. Verwendet wird ein
174:      einfacher Bubblesort, der natürlich sehr langsam, dafür aber
175:      kurz ist ! *)
176:   FOR up:=1 TO Fileno-1 DO
177:     down:=up;
178:     smaller:=TRUE;
179:     WHILE (down>0) AND smaller DO
```

```
180:        IF Compare(Files[down].name,Files[down-1].name)=Less THEN
181:          help:=Files[down-1];
182:          Files[down-1]:=Files[down];
183:          Files[down]:=help;
184:          DEC(down);
185:        ELSE
186:          smaller:=FALSE
187:        END;
188:      END;
189:    END;
190:    (* noch kein File ausgewählt *)
191:    SelectedFile:=-1;
192: END ReadDir;
193:
194: (* File-Namen ab start anzeigen *)
195: PROCEDURE SetNames(start:INTEGER);
196: VAR line, lines : INTEGER;
197:     MFDB        : MFDBType;
198:     pxy         : BigPxyArrayType;
199: BEGIN
200:   (* alle Einträge durchgehen ... *)
201:   FOR line:=0 TO 9 DO
202:     (* ... und auf normal setzen, ... *)
203:     IF ODD(TreeArray^[LINEO+line].state DIV Selected) THEN
204:       TreeArray^[LINEO+line].state:=TreeArray^[LINEO+line].state-Selected;
205:     END;
206:     (* ... Filenamen in die Resource setzen, ... *)
207:     IF start+line<Fileno THEN
208:       TreeArray^[LINEO+line].spec:=ADR(Files[start+line].name);
209:       (* ... ausgewähltes File markieren, ... *)
210:       IF start+line=SelectedFile THEN
211:         TreeArray^[LINEO+line].state:=TreeArray^[LINEO+line].state+Selected;
212:       END;
213:     ELSE
214:       (* nicht benötigte Zeilen auf leer setzen *)
215:       TreeArray^[LINEO+line].spec:=ADR(empty);
216:     END;
217:   END;
218:   (* ist ein Scrollen möglich ? *)
219:   lines:=ABS(startline-start);
220:   GrafMouse(MouseOff,NIL);
221:   IF lines>9 THEN
222:     (* Nein, alles neuzeichnen *)
223:     ObjectDraw(FileTree,BACKGRND,1,x,y,w,h);
224:   ELSE
225:     (* Ja, Rastercopy vorbereiten *)
226:     WITH MFDB DO
227:       (* Adresse des Bildschirmspeichers ... *)
228:       pointer:=ScreenPhysicalBase();
```

```
229:        (* und die anderen Informationen setzen *)
230:        width:=640;
231:        height:=400;
232:        widthW:=40;
233:        format:=1;
234:        planes:=1;
235:      END;
236:      (* hoch- oder runterkopieren ? *)
237:      IF (startline-start)>0 THEN (* runter *)
238:        (* zu kopierenden Bereich feststellen *)
239:        ObjectOffset(FileTree,BACKGRND,pxy[0],pxy[1]);
240:        (* ... und vermerken *)
241:        pxy[2]:=pxy[0]+TreeArray^[BACKGRND].width;
242:        pxy[3]:=pxy[1]+(10-lines)*16;
243:        pxy[4]:=pxy[0];
244:        pxy[5]:=pxy[1]+(lines*16);
245:        pxy[6]:=pxy[2];
246:        pxy[7]:=pxy[1]+TreeArray^[BACKGRND].height;
247:        (* Bildschirmteil kopieren *)
248:        CopyRasterOpaque(Handle,3,pxy,ADR(MFDB),ADR(MFDB));
249:        (* neue Zeilen zeichnen *)
250:        ObjectDraw(FileTree,BACKGRND,1,pxy[0],pxy[1],
251:                           TreeArray^[BACKGRND].width,lines*16);
252:      ELSE (*hoch*)
253:        (* zu kopierenden Bereich feststellen *)
254:        ObjectOffset(FileTree,BACKGRND,pxy[4],pxy[5]);
255:        (* ... und vermerken *)
256:        pxy[0]:=pxy[4];
257:        pxy[1]:=pxy[5]+(lines*16);
258:        pxy[2]:=pxy[4]+TreeArray^[BACKGRND].width;
259:        pxy[3]:=pxy[5]+TreeArray^[BACKGRND].height-1;
260:        pxy[6]:=pxy[2];
261:        pxy[7]:=pxy[5]+(10-lines)*16;
262:        (* Bildschirmteil kopieren *)
263:        CopyRasterOpaque(Handle,3,pxy,ADR(MFDB),ADR(MFDB));
264:        (* neue Zeilen zeichnen *)
265:        ObjectDraw(FileTree,BACKGRND,1,pxy[4],pxy[5]+(10-lines)*16,
266:                           TreeArray^[BACKGRND].width,lines*16);
267:      END;
268:    END;
269:    (* Rollbalken und Rollbox setzen *)
270:    IF Fileno>10 THEN
271:      (* Höhe der Box *)
272:      TreeArray^[SCROLBOX].height:=
273:        (TreeArray^[SCROLBAR].height*10) DIV Fileno;
274:      (* Position der Box *)
275:      TreeArray^[SCROLBOX].y:=(start*(TreeArray^[SCROLBAR].height-
276:                             TreeArray^[SCROLBOX].height)) DIV (Fileno-10);
```

```
277:   ELSE
278:     (* weniger als 10 Namen -> Box hat die Größe des Balkens *)
279:     TreeArray^[SCROLBOX].y:=0;
280:     TreeArray^[SCROLBOX].height:=TreeArray^[SCROLBAR].height;
281:   END;
282:   (* Rollbalken und -box zeichnen *)
283:   ObjectDraw(FileTree,SCROLBAR,1,x,y,w,h);
284:   GrafMouse(MouseOn,NIL);
285: END SetNames;
286:
287: (* auf anderes Laufwerk wechseln *)
288: PROCEDURE SwitchTo(drive:CHAR; result:INTEGER);
289: BEGIN
290:   (* Knopf wieder normal darstellen *)
291:   TreeArray^[result].state:=TreeArray^[result].state-Selected;
292:   ObjectDraw(FileTree,result,1,x,y,w,h);
293:   (* neuen Pfad setzen *)
294:   path:='x:\';
295:   path[0]:=drive;
296:   (* Directory einlesen ... *)
297:   ReadDir;
298:   (* und Namen ausgeben *)
299:   startline:=0;
300:   SetNames(startline);
301: END SwitchTo;
302:
303: BEGIN
304:   (* damit der Pfad-Parameter nicht verändert wird, in lokale
305:      Variable kopieren *)
306:   path:=Path;
307:   (* Leerstring *)
308:   empty:='              ';
309:   GrafMouse(Arrow,NIL);
310:   (* Text der Infozeile setzen *)
311:   MenuText(FileTree,INFOLINE,ADR(infoline));
312:   (* Box zentrieren und zeichnen *)
313:   FormCenter(FileTree,x,y,w,h);
314:   FormDialogue(FormStart,0,0,0,0,x,y,w,h);
315:   ObjectDraw(FileTree,FILESLCT,10,x,y,w,h);
316:   (* Directory einlesen und ausgeben *)
317:   ReadDir;
318:   startline:=0;
319:   SetNames(startline);
320:   REPEAT
321:     (* Dialog ausführen *)
322:     result:=FormDo(FileTree,-1);
323:     CASE result OF
324:       LINEDOWN: (* eine Zeile runterbewegen *)
325:                 IF startline+10 < Fileno THEN
```

```
326:                SetNames(startline+1);
327:                INC(startline);
328:              END                          |
329:      LINEUP  : (* eine Zeile hochbewegen *)
330:              IF startline >0 THEN
331:                SetNames(startline-1);
332:                DEC(startline);
333:              END;                         |
334:      SCROLBOX: (* Rollbox bewegen *)
335:              newpos:=FLOAT(CARDINAL(GrafSlideBox(FileTree,
336:                                  SCROLBAR,SCROLBOX,1)));
337:              (* falls Bewegung möglich war ... *)
338:              IF TreeArray^[SCROLBAR].height#TreeArray^[SCROLBOX].height
339:                THEN
340:                  (* neue Position errechnen *)
341:                  newpos:=(newpos*FLOAT(CARDINAL(Fileno-10)))/1000.0;
342:              ELSE newpos:=0.0;
343:              END;
344:              (* anzeigen und merken *)
345:              SetNames(TRUNC(newpos));
346:              startline:=TRUNC(newpos)     |
347:      SCROLBAR: (* Mausposition feststellen *)
348:              GrafMouseKeyboardState(dummy,my,dummy,dummy);
349:              (* absolute Position der Rollbox feststellen *)
350:              ObjectOffset(FileTree,SCROLBAR,dummy,sy);
351:              my:=my-sy;
352:              (* wurde über oder unter die Box geklickt ? *)
353:              IF my<TreeArray^[SCROLBOX].y THEN (*hoch*)
354:                (* um 10 Files hochgehen *)
355:                IF startline>9 THEN
356:                  SetNames(startline-10);
357:                  DEC(startline,10);
358:                ELSE
359:                  SetNames(0);
360:                  startline:=0;
361:                END;
362:              ELSE
363:                (* um 10 Files runtergehen *)
364:                IF startline<Fileno-19 THEN
365:                  SetNames(startline+10);
366:                  INC(startline,10);
367:                ELSE
368:                  SetNames(Fileno-10);
369:                  startline:=Fileno-10;
370:                END;
371:              END;                         |
372:      LINE0..LINE9: (* ein Dateiname wurde ausgewählt *)
373:              fileselected:=startline+result-LINE0;
374:              (* ist es ein Ordner ? *)
```

```
375:            IF ODD(Files[fileselected].attrib DIV 10H) THEN
376:              (* Namen kopieren *)
377:              Assign(dirname,Files[fileselected].name);
378:              Delete(dirname,0,1);
379:              (* ist es das gleiche Directory ? *)
380:              IF Compare(dirname,' .')#Equal THEN
381:                (* soll eine Stufe zurückgegangen werden *)
382:                IF Compare(dirname,' ..')#Equal THEN
383:                  (* nein, Namen an Pfad anhängen *)
384:                  dirname[0]:='\';
385:                  Insert(dirname,path,Length(path)-1);
386:                ELSE
387:                  (* ja, letzten Directory-Namen löschen *)
388:                  scan:=Length(path)-1;
389:                  (* nur, wenn wir nicht schon ganz oben sind *)
390:                  IF path[scan-1]#':' THEN
391:                    path[scan]:=CHAR(0);
392:                    WHILE path[scan]#'\' DO
393:                      path[scan]:=CHAR(0);
394:                      DEC(scan);
395:                    END;
396:                  END;
397:                END;
398:                (* neues Directory einlesen *)
399:                ReadDir;
400:                (* aus anzeigen *)
401:                startline:=0;
402:                SetNames(startline)
403:              ELSE
404:                (* wieder normal *)
405:                MenuTitleNormal(FileTree,result,1);
406:              END;
407:            ELSE
408:              (* evenuell das alte ausgewählte File
409:                 normal anzeigen *)
410:              IF fileselected<Fileno THEN
411:               IF (SelectedFile>=startline) AND
412:                  (SelectedFile<startline+10) AND
413:                  (SelectedFile#fileselected) THEN
414:                  MenuTitleNormal(FileTree,
415:                                  LINEO+SelectedFile-startline,1);
416:               END;
417:               (* ausgewähltes File merken *)
418:               SelectedFile:=fileselected;
419:              ELSE
420:                (* wieder normal anzeigen *)
421:                IF fileselected#SelectedFile THEN
422:                  MenuTitleNormal(FileTree,result,1);
423:                END;
```

```
424:                END;
425:              END;                                  |
426:              (* jeweils auf das betreffende Laufwerk umschalten *)
427:      DRIVEA:  SwitchTo('A',result)                 |
428:      DRIVEB:  SwitchTo('B',result);                |
429:      DRIVEC:  SwitchTo('C',result);                |
430:      DRIVED:  SwitchTo('D',result);                |
431:      DRIVEE:  SwitchTo('E',result);                |
432:      DRIVEF:  SwitchTo('F',result);                |
433:    ELSE
434:    END;
435:  (* bis OK oder ABBRUCH gewählt wurde *)
436:  UNTIL (result=OK) OR (result=CANCEL);
437:  (* Button normal setzen *)
438:  TreeArray^[result].state:=TreeArray^[result].state-Selected;
439:  (* Dialog beenden *)
440:  FormDialogue(FormFinish,0,0,0,0,x,y,w,h);
441:  (* vermerken, ob ABBRUCH gedrückt wurde *)
442:  cancel:=(result=CANCEL);
443:  (* ausgewähltes File zusammensetzen *)
444:  IF (NOT cancel) AND (SelectedFile#-1) THEN
445:    Concat(path,Files[SelectedFile].name,filename);
446:  ELSE
447:    filename:='';
448:  END;
449: END FileSelect;
450:
451: BEGIN
452:  IF Init () THEN
453:    (* Beispielaufruf auf Laufwerk D *)
454:    thepath:='D:\';
455:    themask:='*.*';
456:    theinfo:='Choose File ...';
457:    FileSelect(thepath,themask,theinfo,filename,cancel);
458:  END;
459:  (* Bildschirm schließen und ... *)
460:    CloseVirtualWorkstation(Handle);
461:    (* Programm abmelden *)
462:    ApplExit;
463: END FileSelect.
```

Am Anfang des Programmtexts stehen wieder die vom Resource-Editor erzeugten Indizes für die Objekte. Außerdem müssen die GEM-Routinen aus den Bibliotheken angefordert werden.

Das Hauptprogramm ruft nach dem Initialisieren die Routine *FileSelect* mit Beispielparametern auf und wird dann nach dem Schließen des Bildschirms beendet.

Das Initialisieren in der Routine *Init* meldet zunächst das Programm mit *ApplInitialise* an und lädt dann auf schon bekannte Weise die Resource (Zeilen 86-96).

Aus optischen Gründen müssen an der Resource einige Korrekturen vorgenommen werden (Zeilen 99-102). Sie können mit dem MMRCP leider nicht ohne weiteres erreicht werden. Um die Auswirkungen dieser Korrekturen zu erkennen, sollte man das Programm einmal mit und einmal ohne Korrekturen laufen lassen und dabei auf optische Feinheiten achten!

Mit *OpenVirtualWorkstation* öffnet das Programm eine Bildschirmarbeitsstation (Zeilen 103-108). Diese wird benötigt, weil später mit einem Rastercopy direkt auf den Bildschirm zugegriffen werden soll.

Die erste lokale Routine in *FileSelect* ist *ReadDir*. Sie dient zum Einlesen des Directories. Dabei steht der Directory-Name in *path* und die Maske, die angibt, welche Files angezeigt werden sollen, in *mask*.

Das Feld *Files* nimmt die Dateinamen und ihre Attribute auf (z.B. Directory oder normales File).

Zunächst wird mit einem direkten Zugriff auf das TedInfo der Kopfzeile der Name des Directories angezeigt (Zeilen 139-141). Das Programm merkt sich die Anzahl der Files vor dem Einlesen (Zeile 143) und setzt die aktuelle Anzahl auf 0 (Zeile 144). Nun müssen noch der Pfad und die Maske zusammengesetzt werden - und das Einlesen kann beginnen.

Mit *SFirst* (Zeile 148) wird der erste Dateiname, auf den die Maske in *name* paßt, eingelesen. Der zweite Parameter gibt an, daß unter normalen Files und Directories gesucht werden soll. *result* gibt bei einem negativen Wert einen Fehler an; bei einer Null ist alles in Ordnung.

Wo steht nun aber die Information über dieses gefundene File? Dazu richtet GEMDOS einen Puffer ein, der Disk-Transfer-Address (DTA) genannt wird. *GetDTA* (Zeile 150) schreibt diese Adresse in den Parameter. Der Aufbau der DTA wurde in den Zeilen 58-64 definiert. Davon interessieren nur die Felder *attrib* und *name*.

Das Attribut-Feld (Zeile 152) sollte man sich merken. Handelt es sich um ein Inhaltsverzeichnis, dann soll es auf dem Bildschirm durch das "Ordner-Zeichen" angezeigt werden. Dies wird dann zum ersten Zeichen des Namens; bei normalen Files steht hier ein Blank (Zeilen 154-158). Aus optischen Gründen wird dem eigentlichen Namen ein weiteres Blank vorangestellt (Zeile 160).

Nun kann der File-Name aus dem DTA-Puffer ins *Files*-Feld kopiert werden. Schließlich wird der Zähler für die Anzahl der Files erhöht (Zeile 165).

SNext sucht das nächste File, wobei GEMDOS die Parameter übernimmt, die beim Aufruf von *SFirst* übergeben wurden.

Jetzt werden alle *Files*-Einträge, die das Programm nicht mehr braucht, gelöscht (Zeilen 169-172). Schließlich folgt ein kleiner Sortieralgorithmus, der die File-Namen in aufsteigender Folge sortiert (durch das "Ordner-Zeichen" werden die Subdirectories an den Anfang des Feldes gebracht!).

Der verwendete Algorithmus ist ein Bubblesort. Dieser ist zwar sehr langsam, dafür aber kurz. Die Langsamkeit macht sich ab etwa 50 Files durch eine kleine Verzögerung bemerkbar. Natürlich können Sie auch einen anderen Algorithmus (z.B. Quicksort) einsetzen.

Anschließend wird die Variable *SelectedFile* auf -1 gesetzt, was besagt, daß noch kein File ausgewählt wurde.

Ich überspringe vorerst die nächsten beiden Prozeduren und komme gleich zum Hauptteil von *FileSelect*.

Damit der Parameter *Path* nach *FileSelect* nicht verändert wird, kopiert das Programm ihn in die lokale Variable *path* (Zeile 306). Dann wird der leere String definiert (Zeile 308), der nicht benutzte Zeilen in der Dateienliste auffüllen soll.

Nun folgt die normale Vorbereitung eines Dialogs (Zeilen 209 bis 315). Es kann jetzt zum ersten Mal ein Directory eingelesen werden. In *startline* wird im Feld *Files* der Index der Datei vermerkt, die in der ersten Zeile der Dateiliste steht. Dies ist am Anfang natürlich die 0.

Dann erfolgt ein Aufruf von *SetNames*. Diese Prozedur dient zur Anzeige der Files in den Dateizeilen. Dabei soll auch berücksichtigt werden, daß einige eventuell schon geschrieben sind und nur an eine andere Bildschirmposition kopiert werden müssen.

Zunächst (Zeilen 201-217) wird bei allen Zeilen das *Selected*-Flag, falls vorhanden, gelöscht (Zeile 204). Dann können die Namen der Files in die Resource eingefügt werden (Zeile 208). Handelt es sich bei einem davon um ein vorher schon ausgewähltes File, muß wieder das *Selected*-Flag gesetzt werden (Zeile 210/211).

Nun wird festgestellt, ob sich schon Zeilen auf dem Schirm befinden (Zeile 219). Müssen mehr als neun Files neu geschrieben werden, so läßt das Programm mit einem *ObjectDraw* alles neu zeichnen.

Andernfalls kann gescrollt werden. Das heißt, ein Teil des Bildschirms wird kopiert, und nur die noch nicht vorhandene File-Namen müssen neu geschrieben werden.

Das Kopieren geschieht mit einem Rastercopy. Dies ist eine Funktion aus der VDI-Raster-Bibliothek. Sie erhält als Parameter unter anderem je einen sogenannten *Memory Form Definition Block*

(MFDB), worin Angaben über die Bildschirme stehen, aus denen und in die kopiert werden soll. Dies sind
- Die Adresse des Bildschirmspeichers (*pointer*)
- Die Breite des Schirms in Punkten (*width*)
- Die Höhe des Schirms in Punkten (*height*)
- Die Breite des Schirms in Worten (*widthW*)
- Das Format des Schirms 0=gerätespezifisch
  1=standard (*format*)
- Die Anzahl der Farbebenen (*planes*)

Diese Angaben werden für einen Mono-Bildschirm gesetzt (Zeilen 226-235), wobei die Bildschirmadresse mit der XBIOS-Funktion *ScreenPhysicalBase* festgestellt wird.

Weiterhin ist für das Rastercopy ein Feld nötig, in dem die Quell- und Zielkoordinaten des zu kopierenden Bildschirmbereiches stehen. Die acht Integer-Werte haben folgende Bedeutung:
1. Wert: X-Koordinate links oben (Quelle)
2. Wert: Y-Koordinate links oben (Quelle)
3. Wert: X-Koordinate rechts unten (Quelle)
4. Wert: Y-Koordinate rechts unten (Quelle)
5. Wert: X-Koordinate links oben (Ziel)
6. Wert: Y-Koordinate links oben (Ziel)
7. Wert: X-Koordinate rechts unten (Ziel)
8. Wert: Y-Koordinate rechts unten (Ziel)

Wie diese Werte gesetzt werden müssen, hängt davon ab, ob hoch- oder runtergescrollt werden soll (Zeile 237).

Beim Runterkopieren fängt der Quell-Bereich ganz links oben an. Die Koordinaten erhält das Programm durch einen Aufruf von *ObjectOffset*. Die Höhe des Bereichs hängt von der Anzahl der zu kopierenden Zeilen ab. (Zeilen 239-242).

Der Zielbereich beginnt entsprechend tiefer (Zeile 244) und endet ganz unten (Zeile 246). Nun kann der Bereich durch Aufruf von *CopyRasterOpaque* kopiert werden. Der Parameter "3" bewirkt, daß alles überschrieben wird, was sich vorher im Zielbereich befand. Die Kopierroutine ist äußerst schnell. Ist Ihr Rechner mit einem "Blitter"-Chip ausgerüstet, übernimmt die Hardware den Kopiervorgang.

Beim Kopieren nach oben sind Quell- und Zielbereich entsprechend vertauscht. Nach dem Kopieren müssen noch die neuen Zeilen gezeichnet werden. Dazu benutzt das Programm die Clip-Parameter bei *ObjectDraw* um den Bereich zu beschränken (Zeile 250 und 265).

Durch das Kopieren ist die Routine erheblich schneller geworden, als dies beim Neuzeichnen aller Zeilen möglich wäre. Das Einsparen von GEM-Aufrufen ist übrigens grundsätzlich eine Möglichkeit, ein Programm schneller zu machen. So ist beispielsweise das Verändern von Flags in den Resources mit

ObjectChange viel langsamer als ein direkter Zugriff auf die Resource mittels eines Zeigers (Pointer).

Das "Pseudo-Fenster" ist mit einer Rollbox ausgestattet, deren Position und Größe noch zu setzen sind. Befinden sich weniger als zehn Files in dem betreffenden Directory - wird anhand von *Fileno* festgestellt - so hat die Box die Größe des Rollbalkens (Zeilen 278-280).

Andernfalls entspricht das Verhältnis 'Höhe der Rollbox' zu 'Höhe des Rollbalkens' dem von 'Anzahl der angezeigten File-Namen' (=10) zu 'Anzahl alle Files'. Daraus ergibt sich die Formel in Zeile 272/273.

Die Position der Rollbox innerhalb des Rollbalkens entspricht der Position des ersten angezeigten File-Namens innerhalb aller vorhandenen Files. Die entsprechende Formel steht in Zeile 275/276.

Mit einem abschließendem *ObjectDraw* werden Rollbox und -balken gezeichnet und mit *SetNames* abgeschlossen.

Jetzt geht es darum, wie auf die einzelnen Objekte aus der Box reagiert werden soll. Bei den Pfeil-hoch und -runter Elementen wird *SetNames* jeweils mit einer Zeile mehr oder weniger aufgerufen und die neue Startzeile vermerkt (Zeilen 324-333).

Bei der Rollbox ermöglicht *GrafSlideBox* eine Bewegung der Box über den Rollbalken. Die Funktion liefert als Ergebnis einen Wert von 1 bis 1000, der die Stellung der Box im Balken bezeichnet.

Mit diesem Wert wird dann in Zeile 341 - wenn möglich - errechnet, welcher Dateiname als erster auf dem Bildschirm erscheinen soll. Die Berechnungsvorschrift dazu ergibt sich aus der Umkehrung der Formel zum Setzen der Rollbox. Bei *newpos* handelt es sich um einen Fließkommawert, da die Zwischenergebnisse beim Umrechnen die Grenzen für Integers überschreiten können.

Ein Klick in den Rollbalken bewirkt zunächst ein Auslesen der vertikalen Mausposition durch *GrafMouseKeyboardState* (Zeile 348). Dann stellt das Programm durch Verrechnung mit der Position der Rollbox (Zeilen 350-351) fest, ob über oder unter diese Box geklickt wurde.

Nach diesem Ergebnis gehe ich in der Liste der vorhandenen Dateien zehn Zeilen hoch oder runter (Zeilen 356-357 und 365-367), wobei das "obere" und "untere" Ende dieser Liste berücksichtigt wird (Zeilen 355/359-360 und 364/368-369).
Wurde ein Dateiname angeklickt, ist zu unterscheiden, ob ein Directory oder ein normales File gewählt wurde.

Im ersten Fall wird zunächst der Name aus dem *Files*-Feld geholt und das Ordner-Zeichen gelöscht (Zeile 377/378). Anschließend ist zu prüfen, ob es sich um eins der beiden Sonder-Directories handelt. Diese haben die Namen "." und ".." und werden bei *ReadDir* miteingelesen. (Das Desktop zeigt beide nicht an!)

"." führt in das Ausgangs-Directory zurück und wird deshalb nicht berücksichtigt (Zeile 405). ".." steigt im Directory-Baum eine Stufe hoch. Ein Auswählen dieses Directories entspricht also der Close-Box in der Standard-Box zum Auswählen von Files.

Für den Fall daß man sich wieder im Ausgangs-Directory befindet, ist der letzte Directory-Name im Pfad zu löschen (Zeilen 387-396). Handelt es sich um ein normales Directory, so wird der Name einfach an den Pfad angehängt (Zeilen 384/385). Mit *ReadDir* (Zeile 399) liest das Programm das neue Directory ein und gibt es aus (Zeilen 401/402).

Bei einem normalen File-Namen wird zunächst geprüft, ob sich noch ein anderer - vorher ausgewählter - Dateiname auf dem Schirm befindet (Zeilen 411-413), dessen Markierung zu löschen wäre (Zeile 414). Danach wird vermerkt, welches File ausgewählt wurde (Zeile 418). GEM setzt die Markierung dieser Files automatisch.

Handelt es sich um einen leeren File-Namen, so wird er einfach wieder normal gesetzt (Zeile 422) und nicht weiter darauf reagiert.

Zum Löschen des *Selected*-Flag und zum Neuzeichnen wird hier die Routine *MenuTitelNormal* mißbraucht. Man kann sie nämlich auf alle String-Objecte anwenden - nicht nur auf Menütitel.

Das Umschalten auf andere Laufwerke durch die Knöpfe *DRIVEA* bis *DRIVEF* erreicht das Programm durch Aufruf von *SwitchTo* mit den entsprechenden Parametern.

Wurde der "OK"- oder "ABBRUCH"-Knopf gedrückt, kann der Dialog beendet (Zeile 436) und zunächst wieder normal gesetzt werden (Zeile 438).

Schließlich erhält GEM die Mitteilung, daß der Dialog beendet ist und es wird vermerkt, ob der "ABBRUCH"-Knopf gedrückt wurde (Zeilen 440-442). Zuletzt muß noch der ausgewählt File-Name aus dem markierten File und dem Pfad zusammengesetzt werden (Zeilen 444-448). Damit ist *FileSelect* beendet.

Diese Routine können Sie leicht in eigene Programme übernehmen. Damit sich ein neues File erzeugen läßt, muß zusätzlich ein "New"- oder "Neu"-Button vorhanden sein, der zu einer einzelnen Box führt, in der ein neuer Dateiname eingegeben werden kann. Gegenüber der von GEM zur Verfügung gestellten File-Select-Box bietet diese Routine den Vorteil, daß eine vollständige Auswahl mit der Maus erfolgen kann. Ein Laufwerkwechsel ist mit der normalen Box etwas umständlich.

9.6 Wie wurde das Kontrollfeld programmiert?

In diesem Kapitel wird das Kontrollfeld "nachprogrammiert". Allerdings nicht vollständig - als Accessory - sondern als eigenständiges Programm. Dabei werden Probleme behandelt, die sich beim Umgang mit Resources einstellen. Gleichzeitig wird der Umgang mit Fenstern angerissen.

Das Kontrollfeld dient dazu, einige Systemparameter zu setzen. Dabei sind alle Funktionen mit der Maus steuerbar. Das Kontrollfeld enthält so unterschiedliche Objekte wie Icons (z.B. Tastastaturklick) , Boxen (z.B. die Farbregler), farbige Felder (die sechzehn Farbanzeigen) oder editierbare Textfelder (Uhrzeit und Datum). Ich gehe davon aus, daß Ihnen die Funktionen des Kontrollfeldes bekannt sind.

Der erste Schritt ist das Aufbauen der Resource mit dem Resource-Editor. Die Beschreibung dieses Vorgangs wird leider umfangreich und kompliziert. Dabei wird klar, wie mit einzelnen Resource-Elementen nicht nur funktionale Objekte zu realisieren sind, sondern auch visuelle Effekte, wie die optische Gliederung von zusammenhängenden Funktionen, erreicht werden können. Die zuerstellende Box soll folgendermaßen aussehen:

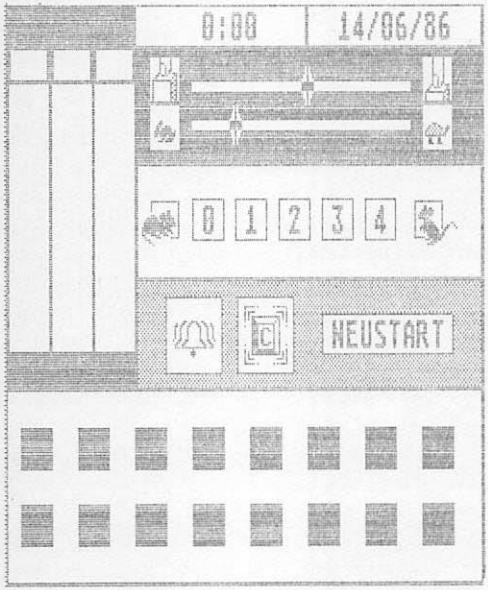


Bild 9.6.1: Die Kontrollfeld-Resource

Angefangen wird der Resource-Aufbau mit den Grundflächen. Sie dienen dazu, das Feld optisch zu gliedern. Alle diese Boxen brauchen keinen Namen.

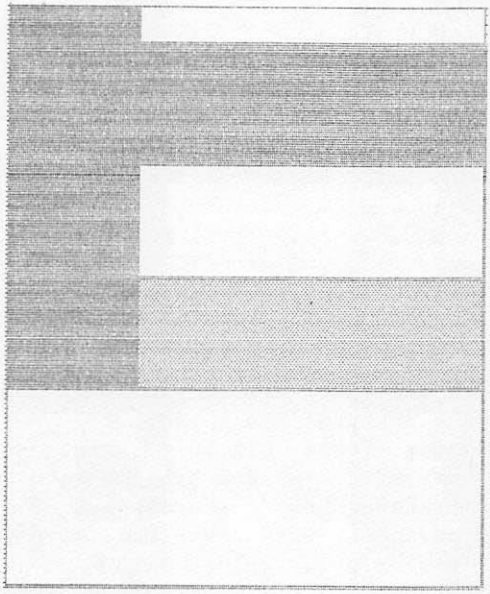


Bild 9.6.2: Die Hintergrundfelder des Kontrollfeldes

Fangen Sie nun mit dem Einsetzen der einzelnen Funktionsobjekte an. Dabei beginnen Sie mit den Farbreglern, die aus jeweils zwei ineinanderliegenden Boxen bestehen. Die größeren werden (von links nach rechts) *PARENTR*, *PARENTG* und *PARENTB* genannt. Die kleineren bekommen die Namen *REGLERR*, *REGLERG* und *REGLERB*.

Bei den *REGLER*-Boxen muß das *TOUCHEXIT*-Flag gesetzt werden.

Die sechzehn Farbfelder werden in der unteren Box plaziert. Sie müssen nacheinander in die Resource eingefügt werden, damit die Indizes aufeinander folgen. Ihre Namen sind *FARBE00* bis *FARBE15*.

Ebenso müssen die fünf Boxen für die Geschwindigkeit der Doppelklicks nacheinander eingefügt werden. Sie erhalten die Namen *MAUS0* bis *MAUS4*.

Die Icons zum Ein- und Ausschalten von Tastaturklick und Warnton brauchen nicht in einer bestimmten Reihenfolge eingefügt werden. Sieerhalten die Namen *KLICKTON*, *WARNTON* und *NEUSTART*.

Die Flags sind bei allen diesen Feldern ohne Bedeutung, da sie nicht in einem Dialog verwendet werden.

Bild 9.6.3: Die Farbregler

Nun zu den Reglern zum Einstellen der Tastaturwiederholrate und deren Verzögerung. Sie bestehen aus mehreren Elementen:

Jeweils rechts und links werden Icons plaziert, die die Funktion der Regler andeuten. Dann wird eine Box hineingesetzt, ohne Rand und in der Farbe der Hintergrundbox, also schwarz. Sie soll nicht auf dem Bildschirm erscheinen. Wozu diese Box dient, wird klar, wenn die Bewegung der Regler programmiert wird.

Dann werden zwei "Striche" eingefügt, die Sie dadurch erhalten, daß Sie jeweils eine Box mit der Maus auf eine "Höhe" von null Zeichen bringen. So bleibt von der Box nur noch der Rand übrig, der somit einen Strich ergibt. Seine Farbe muß auf weiß gesetzt werden, damit er vor dem schwarzen Hintergrund sichtbar ist.

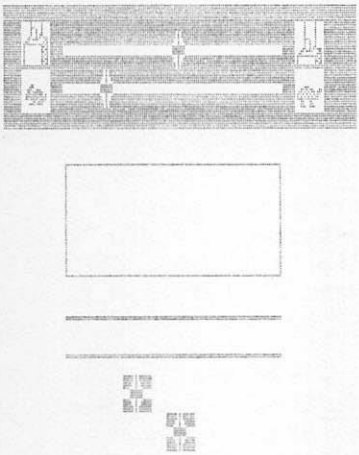


Bild 9.6.4: Die Regler für die Wiederholrate

Schließlich werden die zwei Regler-Icons gesetzt - damit ist die Teilbox fertig. Das Bild zeigt die Box und die Elemente, aus denen sie zusammengesetzt wurde.

Die Felder für Uhrzeit und Datum sind *TEXT*-Objekte mit den Namen *UHRZEIT* und *DATUM*. Bei ihnen muß angegeben werden, welche Maske mit welchem Eingangstext verwendet wird, und welche Zeichen zur Eingabe erlaubt sind. Das sieht dann wie folgt aus:

```
PTMPLT>~~:~~____     PTMPLT>~~/~~/~~____
PVALID>99~99____     PVALID>99~99~99____
 PTEXT> 0~00____      PTEXT>14~06~86____
```

Bild 9.6.5: Die Infos für Uhrzeit und Datum

Bei beiden Feldern müssen die *DEFAULT*-, *EXIT*- und *EDITABLE*-Flags gesetzt werden.

Das Erstellen dieser Resource ist nicht einfach, wenn man sie optisch perfekt gestalten will. Man lernt dadurch einiges über die Arbeit mit dem Resource-Editor und sieht, welche Effekte sich durch geschickte Benutzung der Objekte realisieren lassen.

Im folgenden Programm kommen auch einige Routinen zur Fenstersteuerung vor:

```
 1: MODULE Control;
 2:
 3: FROM SYSTEM            IMPORT  ADDRESS, ADR ;
 4: (* GEM-Module sollen benutzt werden ... *)
 5: FROM GEMAESbase        IMPORT  AESCallResult, RTree, Object,
 6:                                Name, Closer, Mover, WindowName,
 7:                                MouseOn, MouseOff, Arrow,
 8:                                ButtonEvent, MesageEvent,
 9:                                WindowRedraw, WindowClosed,WindowMoved,
10:                                WorkXYWH, CurrXYWH, FirstXYWH, NextXYWH,
11:                                Selected, Outlined ;
12: FROM AESApplications   IMPORT  ApplInitialise, ApplExit ;
13: FROM AESEvents         IMPORT  EventMultiple ;
14: FROM AESForms          IMPORT  FormAlert, FormCenter, FormDialogue,
15:                                FormDo ;
16: FROM AESGraphics       IMPORT  GrafMouse, GrafGrowBox, GrafShrinkBox,
17:                                GrafSlideBox ;
18: FROM AESObjects        IMPORT  ObjectDraw, ObjectFind, ObjectEdit ;
19: FROM AESResources      IMPORT  ResourceLoad, ResourceGetAddr ;
20: FROM AESWindows        IMPORT  WindowCreate, WindowOpen, WindowClose,
```

```
21:                                 WindowDelete, WindowGet, WindowCalc,
22:                                 WindowSet, WindowUpdate ;
23:
24: TYPE GRect = RECORD
25:                x,y,w,h:INTEGER
26:              END;
27:
28: (* Konstante vom MMRCP geliefert *)
29: CONST
30:    CONTROL = 0 ;
31:   NEUSTART = 10 ;
32:   KLICKTON = 11 ;
33:    WARNTON = 12 ;
34:      MAUS0 = 4 ;        (* MAUS0 bis MAUS4 müssen aufeinander- *)
35:      MAUS1 = 5 ;        (* folgende Indizes haben !            *)
36:      MAUS2 = 6 ;
37:      MAUS3 = 7 ;
38:      MAUS4 = 8 ;
39:   PARENTVZ = 15 ;
40:   VERZGRNG = 18 ;
41:   GESCHWDK = 17 ;
42:    PARENTG = 25 ;
43:    PARENTR = 27 ;
44:    REGLERG = 26 ;
45:    REGLERR = 28 ;
46:    PARENTB = 29 ;
47:   PARENTFA = 48 ;
48:    REGLERB = 30 ;
49:    FARBE00 = 32 ;        (* FARBE00 bis FARBE15 müssen aufeinander- *)
50:    FARBE01 = 33 ;        (* folgende Indizes haben !                *)
51:    FARBE02 = 34 ;
52:    FARBE03 = 35 ;
53:    FARBE04 = 36 ;
54:    FARBE05 = 37 ;
55:    FARBE06 = 38 ;
56:    FARBE07 = 39 ;
57:    FARBE08 = 40 ;
58:    FARBE09 = 41 ;
59:    FARBE10 = 42 ;
60:    FARBE11 = 43 ;
61:    FARBE12 = 44 ;
62:    FARBE13 = 45 ;
63:    FARBE14 = 46 ;
64:    FARBE15 = 47 ;
65:    UHRZEIT = 22 ;
66:      DATUM = 31 ;
67:
68: (* MAX und MIN können auch als Standard-Routinen vorhanden sein ! *)
69: PROCEDURE Max(a,b:INTEGER):INTEGER;
```

```
 70: BEGIN
 71:   IF a>b THEN RETURN a
 72:          ELSE RETURN b
 73:   END
 74: END Max;
 75:
 76: PROCEDURE Min(a,b:INTEGER):INTEGER;
 77: BEGIN
 78:   IF a<b THEN RETURN a
 79:          ELSE RETURN b
 80:   END
 81: END Min;
 82:
 83: (* Stellt fest, ob sich die Rechtecke r1 und r2 überlappen.
 84:    Die Überlappungsfläche wird in r2 zurückgegeben *)
 85: PROCEDURE RcIntersect(VAR r1,r2:GRect):BOOLEAN;
 86: VAR x,y,w,h:INTEGER;
 87: BEGIN
 88:   x:=Max(r2.x,r1.x);
 89:   y:=Max(r2.y,r1.y);
 90:   w:=Min(r2.x+r2.w,r1.x+r1.w)-x;
 91:   h:=Min(r2.y+r2.h,r1.y+r1.h)-y;
 92:   r2.x:=x;      r2.y:=y;
 93:   r2.w:=w;      r2.h:=h;
 94:   RETURN ( (w>0) AND (h>0) )
 95: END RcIntersect;
 96:
 97: (* Fenster öffnen *)
 98: PROCEDURE OpenWindow;
 99: BEGIN
100:   GrafMouse(MouseOff,NIL);
101:   name:=' KONTROLLFELD ';
102:   (* Kontrollfeld soll in die Mitte des Bildschirms *)
103:   FormCenter(ControlTree,x,y,w,h);
104:   (* Ausmaße eines Fensters errechnen, in das das Kontrollfeld
105:      hineinpaßt *)
106:   WindowCalc(0,Mover+Closer+Name,x,y,w,h,x,y,w,h);
107:   (* Fenster öffnen *)
108:   Window:=WindowCreate(Mover+Closer+Name,x,y,w,h);
109:   WindowSet(Window,WindowName, INTEGER(ADR(name) DIV 10000H),
110:                                INTEGER(ADR(name) MOD 10000H),0,0);
111:   GrafGrowBox(x+w DIV 2, y+h DIV 2,1,1,x,y,w,h);
112:   WindowOpen(Window,x,y,w,h);
113:   (* hierdurch wird gleichzeitig eine Mitteilung zum Neuzeichnen des
114:      Fensterinhalts ausgelöst, ein ObjectDraw ist hier also nicht
115:      nötig ! *)
116:   GrafMouse(MouseOn,NIL);
117: END OpenWindow;
118:
```

```
119: (* Fenster wieder schließen *)
120: PROCEDURE CloseWindow;
121: BEGIN
122:   GrafMouse(MouseOff,NIL);
123:   (* Fenster schließen *)
124:   WindowClose(Window);
125:   GrafShrinkBox(x+w DIV 2, y+h DIV 2,1,1,x,y,w,h);
126:   WindowDelete(Window);
127:   GrafMouse(MouseOn,NIL);
128: END CloseWindow;
129:
130: (* Zeichnet den Teil des Kontrollfeldes neu, der überschrieben wurde *)
131: PROCEDURE RedrawField(x,y,w,h:INTEGER);
132: VAR r1,r2:GRect;
133: BEGIN
134:   GrafMouse(MouseOff,NIL);
135:   (* GEM-Fenster- und Menukontroll ausschalten *)
136:   WindowUpdate(1);
137:   r2.x:=x;      r2.y:=y;
138:   r2.w:=w;      r2.h:=h;
139:   (* erstes Rechteck der Liste holen *)
140:   WindowGet(Window,FirstXYWH,r1.x,r1.y,r1.w,r1.h);
141:   WHILE ( (r1.w#0) AND (r1.h#0) ) DO
142:     (* Falls ein Teil des Kontrollfeldes zerstört wurde -> neuzeichnen *)
143:     IF RcIntersect(r2,r1) THEN
144:       ObjectDraw(ControlTree,CONTROL,50,r1.x,r1.y,r1.w,r1.h)
145:     END;
146:     (* nächstes Rechteck der Liste holen *)
147:     WindowGet(Window,NextXYWH,r1.x,r1.y,r1.w,r1.h);
148:   END;
149:   (* GEM darf wieder reagieren *)
150:   WindowUpdate(0);
151:   GrafMouse(MouseOn,NIL);
152: END RedrawField;
153:
154: (* Farbregler entsprechend Farbe setzen *)
155: PROCEDURE SetColors(farbe:INTEGER);
156: BEGIN
157:   (* Positionen der Regler in der Resource festlegen *)
158:   TreeArray^[REGLERR].y:=controlinfo.farben[farbe,0]*
159:     TreeArray^[PARENTR].height DIV 8;
160:   TreeArray^[REGLERG].y:=controlinfo.farben[farbe,1]*
161:     TreeArray^[PARENTG].height DIV 8;
162:   TreeArray^[REGLERB].y:=controlinfo.farben[farbe,2]*
163:     TreeArray^[PARENTB].height DIV 8;
164:   (* alle Regler neuzeichnen *)
165:   ObjectDraw(ControlTree,PARENTR,1,x,y,w,h);
166:   ObjectDraw(ControlTree,PARENTG,1,x,y,w,h);
167:   ObjectDraw(ControlTree,PARENTB,1,x,y,w,h);
```

```
168: END SetColors;
169:
170: (* Auf ausgewähltes Objekt reagieren *)
171: PROCEDURE HandleObject(which:INTEGER);
172: VAR dummy:INTEGER;
173: BEGIN
174:   CASE which OF
175:     MAUS0..MAUS4 : (* Mausgeschwindigkeit verändern ? *)
176:                    IF which-MAUS0 # controlinfo.maus THEN
177:                      (* alte Anzeige normal setzen *)
178:                      TreeArray^[controlinfo.maus+MAUS0].state:=
179:                        TreeArray^[controlinfo.maus+MAUS0].state-Selected;
180:                      ObjectDraw(ControlTree,controlinfo.maus+
181:                        MAUS0,0,x,y,w,h);
182:                      (* neuen Zustand merken ... *)
183:                      controlinfo.maus:=which-MAUS0;
184:                      (* ... in der Resource setzen und zeichnen *)
185:                      TreeArray^[controlinfo.maus+MAUS0].state:=
186:                        TreeArray^[controlinfo.maus+MAUS0].state+Selected;
187:                      ObjectDraw(ControlTree,controlinfo.maus+
188:                        MAUS0,0,x,y,w,h);
189:                    END  |
190:     NEUSTART : (* müßte eine Neustartfunktion eingefügt werden, die
191:                  alle Werte wie beim Start setzt und das Feld ent-
192:                  sprechend neu zeichnet. Aus Platzgründen verzichten
193:                  wir hier darauf *)
194:                        |
195:     KLICKTON : (* ein- oder ausschalten ? *)
196:                CASE controlinfo.klickton OF
197:                  TRUE : TreeArray^[KLICKTON].state:=
198:                         TreeArray^[KLICKTON].state+Selected  |
199:                  FALSE: TreeArray^[KLICKTON].state:=
200:                         TreeArray^[KLICKTON].state-Selected
201:                END;
202:                (* neuen Zustand merken *)
203:                controlinfo.klickton:=NOT controlinfo.klickton;
204:                (* und zeichnen *)
205:                ObjectDraw(ControlTree,KLICKTON,0,x,y,w,h);    |
206:     WARNTON  : (* ein- oder ausschalten ? *)
207:                CASE controlinfo.warnton OF
208:                  TRUE : TreeArray^[WARNTON].state:=
209:                         TreeArray^[WARNTON].state+Selected  |
210:                  FALSE: TreeArray^[WARNTON].state:=
211:                         TreeArray^[WARNTON].state-Selected
212:                END;
213:                (* neuen Zustand merken *)
214:                controlinfo.warnton:=NOT controlinfo.warnton;
215:                (* und zeichnen *)
216:                ObjectDraw(ControlTree,WARNTON,0,x,y,w,h);    |
```

```
217:    VERZGRNG : (* Den Regler mit GrafSlideBox durch der Benutzer
218:                   bewegen lassen und neuen Zustand merken *)
219:               controlinfo.verzgrng:=GrafSlideBox(ControlTree,
220:                 PARENTVZ,VERZGRNG,0) DIV 45;
221:               (* neue Position des Reglers setzen und neu zeichnen *)
222:               TreeArray^[VERZGRNG].x:=controlinfo.verzgrng*
223:                 TreeArray^[PARENTVZ].width DIV 25;
224:               ObjectDraw(ControlTree,PARENTVZ,2,x,y,w,h);   |
225:    GESCHWDK : (* Ebenfalls durch GrafSlideBox bewegen lassen *)
226:               controlinfo.geschwdk:=GrafSlideBox(ControlTree,
227:                 PARENTVZ,GESCHWDK,0) DIV 45;
228:               (* neue Position setzen und neu zeichnen *)
229:               TreeArray^[GESCHWDK].x:=controlinfo.geschwdk*
230:                 TreeArray^[PARENTVZ].width DIV 25;
231:               ObjectDraw(ControlTree,PARENTVZ,2,x,y,w,h);   |
232:    UHRZEIT  : (* Uhrzeit-Feld invertieren *)
233:               TreeArray^[UHRZEIT].state:=
234:                 TreeArray^[UHRZEIT].state+Selected;
235:               ObjectDraw(ControlTree,UHRZEIT,0,x,y,w,h);
236:               (* Eingabe *)
237:               dummy:=FormDo(ControlTree,UHRZEIT);
238:               (* hier müßte eine Prüfung erfolgen, ob die Uhrzeit
239:                  gültig ist (z.B. 99.78 Uhr) *)
240:               (* unsichere GEM-Ergebnisse abfangen *)
241:               IF (dummy=UHRZEIT) OR (dummy=DATUM) THEN
242:                 (* Das Objekt, durch das der Dialog verlassen wurde
243:                    wieder normal setzen  *)
244:                 TreeArray^[dummy].state:=
245:                   TreeArray^[dummy].state-Selected;
246:                 ObjectDraw(ControlTree,dummy,0,x,y,w,h);
247:                 (* eventuell anderes Default-Feld normal setzen *)
248:                 IF dummy#UHRZEIT THEN
249:                   TreeArray^[UHRZEIT].state:=
250:                     TreeArray^[UHRZEIT].state-Selected;
251:                   ObjectDraw(ControlTree,UHRZEIT,0,x,y,w,h);
252:                 END;
253:               END                                           |
254:    DATUM    : (* Datums-Feld invertieren *)
255:               TreeArray^[DATUM].state:=
256:                 TreeArray^[DATUM].state+Selected;
257:               ObjectDraw(ControlTree,DATUM,0,x,y,w,h);
258:               (* Eingabe *)
259:               dummy:=FormDo(ControlTree,DATUM);
260:               (* hier müßte eine Prüfung erfolgen, ob das Datum
261:                  gültig ist (z.B. 31.2.86) *)
262:               (* unsichere GEM-Ergebnisse abfangen *)
263:               IF (dummy=UHRZEIT) OR (dummy=DATUM) THEN
264:                 (* Das Objekt, durch das der Dialog verlassen wurde
265:                    wieder normal setzen  *)
```

```
266:                TreeArray^[dummy].state:=
267:                  TreeArray^[dummy].state-Selected;
268:                ObjectDraw(ControlTree,dummy,0,x,y,w,h);
269:                (* eventuell anderes Default-Feld normal setzen *)
270:                IF dummy#DATUM THEN
271:                  TreeArray^[DATUM].state:=
272:                    TreeArray^[DATUM].state-Selected;
273:                  ObjectDraw(ControlTree,DATUM,0,x,y,w,h);
274:                END;
275:              END                                        |
276:    REGLERR  : (* Regler durch GrafSlideBox bewegen lassen und neue
277:                  Position merken *)
278:              controlinfo.farben[aktfarbe,0]:=GrafSlideBox(ControlTree,
279:                PARENTR,REGLERR,1) DIV 142;
280:              (* neue Position in der Resource vermerken und neu zeichnen *)
281:              TreeArray^[REGLERR].y:=controlinfo.farben[aktfarbe,0]*
282:                TreeArray^[PARENTR].height DIV 8;
283:              ObjectDraw(ControlTree,PARENTR,1,x,y,w,h);   |
284:    REGLERG  : (* wie bei REGLERR *)
285:              controlinfo.farben[aktfarbe,1]:=GrafSlideBox(ControlTree,
286:                PARENTG,REGLERG,1) DIV 142;
287:              TreeArray^[REGLERG].y:=controlinfo.farben[aktfarbe,1]*
288:                TreeArray^[PARENTG].height DIV 8;
289:              ObjectDraw(ControlTree,PARENTG,1,x,y,w,h);   |
290:    REGLERB  : (* wie bei REGLERR *)
291:              controlinfo.farben[aktfarbe,2]:=GrafSlideBox(ControlTree,
292:                PARENTB,REGLERB,1) DIV 142;
293:              TreeArray^[REGLERB].y:=controlinfo.farben[aktfarbe,2]*
294:                TreeArray^[PARENTB].height DIV 8;
295:              ObjectDraw(ControlTree,PARENTB,1,x,y,w,h);   |
296:    FARBE00..FARBE15: (* Soll eine neue Farbe ausgewählt werden ? *)
297:                      IF which-FARBE00#aktfarbe THEN
298:                        (* altes Farbfeld auf normal setzen *)
299:                        TreeArray^[aktfarbe+FARBE00].state:=
300:                          TreeArray^[aktfarbe+FARBE00].state-Outlined;
301:                        (* neue Fabe merken *)
302:                        aktfarbe:=which-FARBE00;
303:                        (* neues Farbfeld umranden *)
304:                        TreeArray^[aktfarbe+FARBE00].state:=
305:                          TreeArray^[aktfarbe+FARBE00].state+Outlined;
306:                        ObjectDraw(ControlTree,PARENTFA,1,x,y,w,h);
307:                        (* R-,G- und B-Werte neu setzen und zeichnen *)
308:                        SetColors(aktfarbe);
309:                      END    |
310:  ELSE
311:  END;
312: END HandleObject;
313:
```

```
314: (* Überwachung der Benutzeraktionen *)
315: PROCEDURE HandleField;
316: VAR mox,moy,TheObject:INTEGER;
317: VAR buff: ARRAY [0..9] OF INTEGER;
318: BEGIN
319:   (* Auf ein Ereignis warten *)
320:   CASE EventMultiple(ButtonEvent+MesageEvent,
321:                      1,1,1,
322:                      0,0,0,0,0,0,0,0,0,0,
323:                      ADR(buff),
324:                      0,0,
325:                      mox,moy,dummy,dummy,
326:                      dummy,dummy)              OF
327:     (* Ein Knopf wurde gedrückt *)
328:     ButtonEvent:  (* Objekt unter dem Mauszeiger feststellen *)
329:                   TheObject:=ObjectFind(ControlTree,CONTROL,
330:                                         50,mox,moy);
331:                   (* Falls eins gefunden wurde -> darauf reagieren *)
332:                   IF TheObject#-1 THEN HandleObject(TheObject) END      |
333:     (* Wir haben eine Mitteilung erhalten *)
334:     MesageEvent: CASE buff[0] OF
335:                    (* Teile des Fensters neu zeichnen *)
336:                    WindowRedraw: RedrawField(buff[4],buff[5],
337:                                              buff[6],buff[7])      |
338:                    (* Das Fenster wurde geschlossen -> Ende *)
339:                    WindowClosed: done:=TRUE                        |
340:                    (* Das Fenster wurde bewegt *)
341:                    WindowMoved : (* Neue Fensterposition setzen *)
342:                                  WindowSet(Window,CurrXYWH,buff[4],
343:                                            buff[5],buff[6],buff[7]);
344:                                  WindowGet(Window,WorkXYWH,x,y,w,h);
345:                                  (* Position der Resource setzen *)
346:                                  TreeArray^[CONTROL].x:=x;
347:                                  TreeArray^[CONTROL].y:=y;
348:                  ELSE (* Auf andere Mitteilungen nicht reagieren *)
349:                  END;
350:   ELSE (* Auf andere Ereignisse wird nicht reagiert *)
351:   END;
352: END HandleField;
353:
354: (* Setzen der Werte, die mit dem Kontrollfeld geändert werden können.
355:    Hier sind eigentlich Systemaufrufe nötig, die allerdings das
356:    Programm erheblich in die Länge ziehen würden *)
357: PROCEDURE InitControlInfo;
358: VAR farbe,rgb:INTEGER;
359: BEGIN
360:   WITH controlinfo DO
361:     maus:=3;
362:     klickton:=TRUE;
```

```
363:     warnton:=FALSE;
364:     verzgrng:=8;
365:     geschwdk:=15;
366:     uhrzeit:='1346';
367:     datum:='300886';
368:     FOR farbe:=0 TO 15 DO
369:       FOR rgb:=0 TO 2 DO
370:         farben[farbe,rgb]:=0
371:       END
372:     END;
373:     FOR rgb:=0 TO 2 DO
374:       farben[1,rgb]:=7
375:     END;
376:     aktfarbe:=0;
377:     (* Resource entsprechend der Einstellungen setzen *)
378:     TreeArray^[maus+MAUS0].state:=TreeArray^[maus+MAUS0].state+Selected;
379:     TreeArray^[GESCHWDK].x:=geschwdk*TreeArray^[PARENTVZ].width DIV 25;
380:     TreeArray^[VERZGRNG].x:=verzgrng*TreeArray^[PARENTVZ].width DIV 25;
381:     IF NOT warnton THEN TreeArray^[WARNTON].state:=
382:                           TreeArray^[WARNTON].state+Selected
383:     END;
384:     IF NOT klickton THEN TreeArray^[KLICKTON].state:=
385:                            TreeArray^[KLICKTON].state+Selected
386:     END;
387:     TreeArray^[aktfarbe+FARBE00].state:=
388:       TreeArray^[aktfarbe+FARBE00].state+Outlined;
389:   END;
390: END InitControlInfo;
391:
392: VAR ApplID:INTEGER;
393:     Window:INTEGER;
394:     dummy:INTEGER;
395:     done:BOOLEAN;
396:     ControlTree:ADDRESS;
397:     x,y,w,h:INTEGER;
398:     TreeArray:POINTER TO ARRAY [0..50] OF Object;
399:     name:ARRAY[0..80] OF CHAR;
400:     controlinfo:RECORD
401:                 maus     :INTEGER;
402:                 klickton,
403:                 warnton  :BOOLEAN;
404:                 verzgrng,
405:                 geschwdk :INTEGER;
406:                 uhrzeit,
407:                 datum    :ARRAY[0..10] OF CHAR;
408:                 farben   :ARRAY[0..15],[0..2] OF INTEGER;
409:               END;
410:     aktfarbe: INTEGER;
411:
```

```
412: BEGIN
413:   (* Programm bei GEM anmelden *)
414:   ApplID:=ApplInitialise();
415:   (* Resource laden *)
416:   ResourceLoad('CONTROL.RSC');
417:   IF AESCallResult=0 THEN
418:     dummy:=FormAlert(1,'[3][Kann CONTROL.RSC nicht laden][OK]');
419:   ELSE
420:     (* Resource-Adresse holen *)
421:     GrafMouse(Arrow,NIL);
422:     ResourceGetAddr(RTree,CONTROL,ControlTree);
423:     TreeArray:=ControlTree;
424:     (* Initialisieren und Fenster öffnen *)
425:     InitControlInfo;
426:     OpenWindow;
427:     (* Farbregler setzen *)
428:     SetColors(aktfarbe);
429:     done:=FALSE;
430:     (* Bis zum bitteren Ende durchhalten ... *)
431:     REPEAT
432:       HandleField
433:     UNTIL done;
434:     (* Fenster schließen *)
435:     CloseWindow;
436:   END;
437:   (* Program abmelden *)
438:   ApplExit;
439: END Control.
```

Nachdem alle benötigten Routinen aus den Bibliotheken angefordert sind, folgen ab Zeile 28 die Konstanten der Objekte, die von MMRCP erzeugt werden. Diese können sich bei Ihnen natürlich von den abgedruckten unterscheiden.

Die Funktionen *Min* und *Max* werden später noch benutzt und können eventuell auch als Standardroutinen vorhanden sein. Sie ergeben das Minimum beziehungsweise das Maximum zweier Werte.

RcIntersect stellt fest, ob sich zwei Rechtecke überlappen. Dazu wird der Typ *GRect* benutzt, der ab Zeile 24 definiert wurde. Er enthält die Position und Größe eines Rechteckes. Die Überlappungsfläche wird in *r2* zurückgegeben; zugleich zeigt der Funktionswert an, ob eine Überlappung überhaupt vorhanden ist.

Darauf folgt die erste Fensterroutine zum Öffnen eines Fensters, in das die Kontrollfeld-Resource hineinpaßt. Nach dem Ausschalten des Mauszeigers (Zeile 100) und dem Festlegen des Fensternamens (Zeile 101) ermittelt das Programm mit *FormCenter* den Platz, den die Resource belegen wird. Dies sind die inneren Maße des Fensters.

Da GEM die äußeren Maße (einschließlich der Titelzeile) benötigt, lasse ich diese Werte mit *WindowCalc* berechnen.

Das Fenster läßt sich jetzt öffnen. Mit *WindowCreate* wird es angemeldet und erhält eine Kennziffer, die später bei allen Fenster-Befehlen gebraucht wird.

Mit *WindowSet* (Zeile 109) wird GEM mitgeteilt, wo die Titelzeile zu finden ist. Der anschließende Aufruf von *GrafGrowBox* hat nur optische Funktion und könnte auch wegfallen.

Durch *WindowOpen* wird das Fenster nun auf den Bildschirm gezeichnet. GEM kann dessen Inhalt natürlich nicht voraussehen und schickt eine Mitteilung ab, daß der Fensterinhalt neu gezeichnet werden muß.

CloseWindow schließt das Fenster wieder (ab Zeile 120). *WindowClose* ist für das Löschen des Bildschirmfensters verantwortlich. *GrafShrinkBox* wurde wieder nur zur optischen Verschönerung eingefügt. Mit *WindowDelete* teilt das Programm GEM mit, daß das Fenster nicht mehr gebraucht wird und die Kennziffer für ein anderes Fenster frei ist.

Es folgt die Routine zum Neuzeichnen des Kontrollfeldes, falls Teile von ihm zerstört (überschrieben) wurden. Dazu kann es kommen, wenn das Fenster teilweise aus dem Bildschirm "herausgeschoben" wurde und dann wieder in die Mitte des Schirms gesetzt werden soll. Die Teile, die außerhalb des Bildschirms lagen, sind nicht mehr verfügbar und GEM kann sie nicht selbständig zeichnen. Auch ist direkt nach dem Öffnen des Fensters logischerweise ein Neuzeichnen nötig.

Die Routine erhält als Parameter die Ausmaße des Kontrollfelds. Mit *WindowUpdate* (Zeile 136) und dem Parameter 1 wird GEM mitgeteilt, daß keine Fenster mehr bewegt werden dürfen und auch keine Menüs heruntergeklappt werden sollen. Während das Programm am Zeichnen ist käme es sonst zu einem Durcheinander auf dem Bildschirm.

GEM stellt eine Liste der zerstörten Flächen zur Verfügung. Dies ist die sogenannte Rechteckliste. Alle Rechtecke zusammen ergeben die gesamte neu zu zeichnende Fläche.

Mit einem *WindowGet*-Aufruf mit Parameter *FirstXYWH* (Zeile 140) ergibt sich das erste Rechteck dieser Liste, das im Record *r1* abgelegt wird.

In der *WHILE*-Schleife (Zeile 141) prüft das Programm, ob sich das Rechteck mit dem Kontrollfeld überschneidet (dazu braucht man *RcIntersect*). Ist dies der Fall, so wird die Überlappungsfläche neu gezeichnet. Hier werden die letzten vier Parameter des *ObjectDraw*-Aufrufs (Zeile 144), nämlich die Clipping-Variablen, benutzt, um das Neuzeichnen auf die kleinstmögliche Fläche zu beschränken.

WindowGet mit dem Parameter *NextXYWH* (Zeile 147) liefert das nächste Rechteck aus der Liste, das dann wieder neu gezeichnet wird. Hat das Rechteck eine Höhe oder Breite von Null, so ist das Ende der Rechteckliste erreicht, und die Prozedur ist fertig.

Ein 0-Parameter bei *WindowUpdate* erlaubt GEM, wieder auf Fensterbewegungen und Menüs zu reagieren (Zeile 150).

Nun erfolgt die eigentliche Resource-Programmierung für das Kontrollfeld: In den Zeilen 400 bis 409 wird eine Struktur definiert, in der alle Informationen gehalten werden sollen, die für das Feld wichtig sind - also die Farbeinstellungen, Mausgeschwindigkeit und so weiter.

Die Farbeinstellungen benötigt auch die Routine *SetColors* (ab Zeile 155). Sie soll die drei Farbregler entsprechend der jeweiligen Farbvoreinstellung setzen.

Im *controlinfo* stehen die Farbwerte im Feld *farbe*, in dem drei Angaben für den Rot-, Grün- und Blauanteil gehalten werden. Diese werden gebraucht, um die Position der Regler zu setzen. Da es jeweils acht Stufen gibt, bedeutet jede Stufe eine "Verschiebung" des Reglers um ein Achtel der Höhe der Hintergrundboxen. Dieser Wert wird direkt in die Resource eingetragen (Zeilen 158 bis 163).

Mit drei *ObjectDraw*-Aufrufen (Zeilen 165 bis 167) werden die Regler neu gezeichnet. Es ist wichtig, zuerst die Hintergrundboxen (*PARENTx*) zu zeichnen, da sonst die "alten" Regler nicht gelöscht würden.

Ich überspringe vorerst die Routine *HandleObject* (ab Zeile 171) und gehe zum Hauptprogramm über (ab Zeile 412).

Zunächst wird das Programm mit *ApplInitialise* bei GEM angemeldet und die Resource geladen. Der Aufruf von *InitControlInfo* (Zeile 425) bewirkt das Setzen des *controlinfo*-Records.

Diese Routine (ab Zeile 357) müßte eigentlich einige Systemaufrufe durchführen, um die benötigten Informationen zu erhalten. Das erspare ich mir (genauso wie das Setzen dieser Werte beim Verlassen des Kontrollfeldes) in diesem Beispielprogramm und nehme einfach konstante Werte.

Ab Zeile 378 werden einige Resource-Objekte gesetzt. Dies betrifft die Doppelklick-Geschwindigkeit (Zeile 378), die Regler für die Tastaturwiederholung (Zeilen 379-380), die Zustände des Tastaturklicks und des Warntons (Zeilen 381-386) sowie die Markierung der gerade ausgewählten Farbe. Diese Routinen kommen später nochmals bei *HandleObject* vor und werden dort besprochen.

Nun wird das Fenster geöffnet und damit auch das Kontrollfeld gezeichnet. Das Programm setzt die Regler der aktuellen Farbe - und dann kann mit dem Feld gearbeitet werden. Das übernimmt die Routine *HandleField*. Soll das Kontrollfeld verlassen werden, ist das Fenster zu schließen und mit *ApplExit* das Programm bei GEM abzumelden.

HandleField wartet mit *EventMultiple* (Zeilen 320-326) auf ein *ButtonEvent* (wenn ein Knopf gedrückt wurde) und ein *MessageEvent* (z.B. wenn das Fenster bewegt wird). Eine eventuelle Mitteilung soll im Feld *buff* abgelegt werden und die Mauskoordinaten bei einem Klick in *mox* und *moy*.

Trat ein Knopf-Ereignis auf (Zeilen 328-332), so wird mit *ObjectFind* festgestellt, welches Objekt sich unter der Maus befand. Falls die Routine das Objekt gefunden hat, reicht das Programm seinen Index an die Routine *HandleObject* weiter.

Handelte es sich um ein Mitteilungs-Ereignis, wird auf die verschiedenen Fenster-Mitteilungen reagiert. Bei *WindowRedraw* (Zeile 336) soll ein Teil des Fensters neu gezeichnet werden. Die Ausmaße dieses Bereichs stehen im vierten bis siebten Wort des Mitteilungspuffers. Diese Aufgabe übernimmt die Routine *RedrawField*.

War es eine *WindowClosed*-Mitteilung, hatte der Benutzer die Close-Box links oben am Fenster gedrückt und will das Programm verlassen. Durch Setzen von *done* wird dies erreicht (Zeile 339).

WindowMoved bedeutet, daß das Fenster verschoben wurde. Die neuen Koordinatenwerte finden sich wieder im vierten bis siebten Wort des Mitteilungspuffers.

Mit *WindowSet* und dem Parameter *CurrXYWH* (Zeile 342) bekommt GEM die neuen Koordinaten mitgeteilt. Eine Mitteilung zum Neuzeichnen von Fensterteilen erzeugt GEM bei Bedarf automatisch.

Mit *WindowGet* (Zeile 344) wird die neue Position des inneren Fensterraums ermittelt, in dem das Kontrollfeld steht. Damit alle folgenden *ObjectFind* korrekt arbeiten, müssen diese Werte in die Resource eingetragen werden (Zeilen 346/347). Ein Neuzeichnen ist hier nicht nötig - dies geschieht über die schon genannte Mitteilung von GEM.

Auf andere Mitteilungen braucht das Programm nicht zu reagieren. Die Routine *HandleObject* verarbeitet nun die einzelnen angeklickten Objekte.

Bei den *MAUSx*-Objekten muß das neu ausgewählte Feld invertiert, und das alte wieder normal gezeichnet werden. Dazu wird beim "alten" einfach das *Selected*-Flag gelöscht und das Objekt neu gezeichnet (Zeilen 178-181).

Das Programm merkt sich den neuen Wert, und wenn er auftritt setzt es das *Selected*-Flag mit anschließendem Neuzeichnen.

Die *Neustart*-Funktion ist hier nicht implementiert (Zeile 190). An dieser Stelle müßte beispielsweise ein erneuter Aufruf von *InitControlInfo* stehen.

Beim Tastaturklick- (Zeilen 195-205) und Warnton (Zeilen 206-216) kann analog vorgegangen werden. Je nach aktuellem Zustand wird das *Selected*-Flag gesetzt oder gelöscht. Das Programm merkt sich wieder den neuen Zustand und zeichnet das jeweilige Objekt neu.

Die Regler für die Tastaturwiederholrate werden genauso behandelt (ab Zeile 217). Mit *GrafSlideBox* kann das Regler-Icon bewegt werden. Die Funktion liefert als Ergebnis die neue Position. Sie merken sie sich im Record *controlinfo*, setzen sie in die Resource und zeichnen die Regler neu.

Wurde die Uhrzeit angewählt, so invertiert das Programm zunächst dieses Feld (Zeile 233-235). Dann wird ein *FormDo* aufgerufen (wobei der Cursor auf die Uhrzeit gesetzt wird) und der Benutzer kann die Uhrzeit setzen. Da bei der Uhrzeit das *Default*-Flag gesetzt wurde, kann die Eingabe mit der Return-Taste abgeschlossen werden.

Nun müßte eine Prüfung der eingegebenen Uhrzeit erfolgen. Die will ich Ihnen ersparen und Sie statt dessen mit einem anderen Problem konfrontieren: Da es zwei editierbare Objekte gibt, nämlich die Uhrzeit und das Datum, die beide das *Editable*-Flag gesetzt haben, kann nach den Regeln für einen GEM-Dialog mit den Cursor-Tasten zum Datum gewechselt werden. Außerdem ist für GEM nicht entscheidbar, welches Objekt mit der Return-Taste ausgewählt werden soll, da zwei Objekte mit dem "*Default*"-Flag im Dialog vorhanden sind.

Daher ist das Ergebnis von *FormDo* (der Index des Objekts, über das der Dialog verlassen wurde) nicht immer korrekt und muß zunächst überprüft werden (Zeile 241).

Trat kein Fehler auf, so wird bei dem betreffenden Objekt das *Selected*-Flag gelöscht (es wurde ja durch die Return-Taste gesetzt) und das Objekt neu gezeichnet.
Falls durch die Cursor-Tasten in das Datumsfeld übergewechselt wurde, so ist beim Uhrzeit-Objekt dieses Flag noch gesetzt und wird gelöscht. Damit sind beide Objekte wieder im normalen Zustand.

Die Behandlung des Datums funktioniert entsprechend, nur wird eben das Wechseln auf das Uhrzeit-Feld berücksichtigt.

Bei den drei Reglern ruft das Programm wieder *GrafSlideBox* auf, errechnet aus dessen Ergebnis die neue Stellung und setzt sie.

Die Auswahl eines Farbfeldes ähnelt der Behandlung der *MAUSx*-Objekte. Beim "alten" Farbfeld wird das *Outlined*-Flag gelöscht, beim "neuen" gesetzt und beide neu gezeichnet. Zusätzlich ruft das Programm noch *SetColors* auf, womit die einzelnen Regler entsprechend gesetzt werden.

Damit ist das Kontrollfeld fertig. Wenn Sie wollen, können Sie natürlich noch die letzten Feinheiten ausprogrammieren. Das Beispiel zeigt, wie ein Bedien- oder Kontrollfeld programmiert werden kann, das den Konzepten einer grafischen Benutzeroberfläche voll entspricht und in dem alle möglichen Aktionen mit der Maus angesteuert werden.

9.7 Icons bewegen

Viele Programme verwenden ein eigenes Desktop, auf dem Icons "liegen", die verschoben, geöffnet oder anders verwendet werden können. Ein Beispiel dafür ist das GEM-Desktop mit den Icons für die Diskettenstationen und den Papierkorb.

Ich stelle Ihnen jetzt ein kleines Programm vor, mit dem man ein Icon auf einem kleinen Desktop verschieben kann.

Zuerst brauchen Sie eine Resource. Sie besteht einfach aus einer großen Box, in die ein Icon gesetzt wird. Das hier verwendete Icon entspricht den "Karteikästen" des GEM-Desktops. Dabei lautet der Icon-Text "DISKSTATION". Zusätzlich bekommt das Icon den Buchstaben "A", der innerhalb des Icons plaziert werden muß. Dies soll natürlich alles im Resource-Editor geschehen.

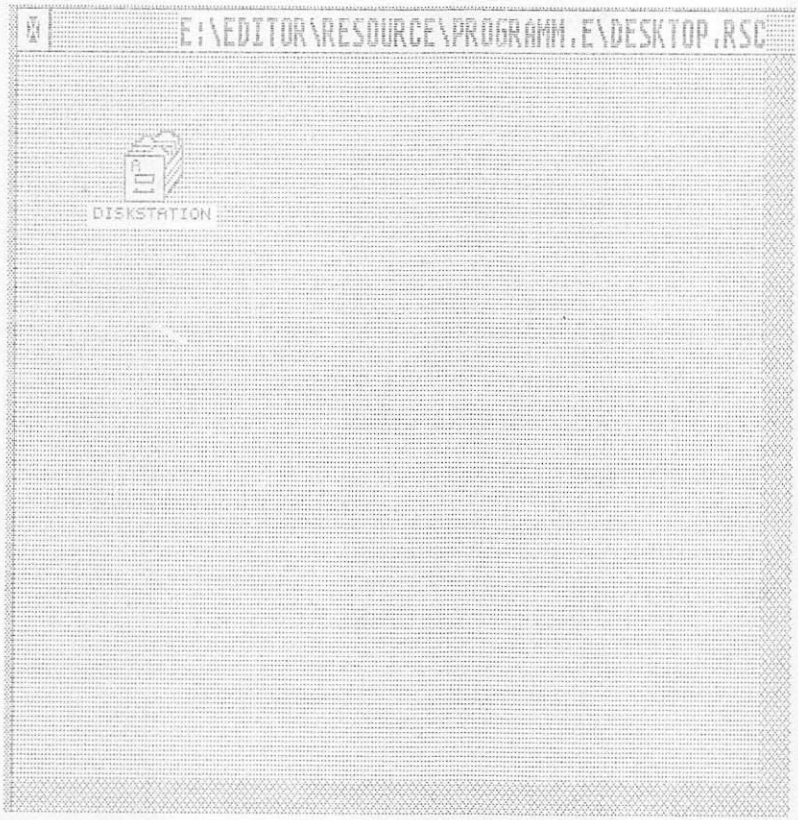


Bild 9.7.1: Die Desktop-Resource

Vor dem Abspeichern erhalten die Objekte Namen. Der Hintergrund heißt hier *DESKTOP* und das Icon sinnigerweise *ICON*.

Es ergibt sich das folgende Listing:

```
  1: MODULE Desktop;
  2:
  3: (* Standard modules      *)
  4: FROM SYSTEM              IMPORT ADDRESS, ADR ;
  5: (* GEM modules           *)
  6: FROM AESApplications     IMPORT ApplInitialise ;
  7: FROM GEMAESbase          IMPORT RTree, AESCallResult, Object,
  8:                                 Arrow, FlatHand, MouseOn, MouseOff;
  9: FROM AESEvents           IMPORT EventButton ;
 10: FROM AESForms            IMPORT FormAlert, FormCenter ;
 11: FROM AESResources        IMPORT ResourceGetAddr, ResourceLoad ;
 12: FROM AESGraphics         IMPORT GrafMouse, GrafDragBox ;
 13: FROM AESObjects          IMPORT ObjectDraw, ObjectFind, ObjectOffset ;
 14:
 15: (* Konstante vom MMRCP geliefert *)
 16: CONST
 17:       ICON = 1 ;
 18:    DESKTOP = 0 ;
 19:
 20: VAR DeskTree :ADDRESS ;
 21:     Appl : INTEGER ;
 22:
 23:
 24: (* Resource laden und Adresse feststellen *)
 25: PROCEDURE Init() : BOOLEAN ;
 26: CONST RSCFileName = 'DESKTOP.RSC' ;
 27:       AlertText  = "[3][Cannot load DESKTOP.RSC][OK]" ;
 28:
 29: VAR str: ARRAY [0..99] OF CHAR ;
 30:     dummy: INTEGER ;
 31:
 32: BEGIN
 33:   Appl:= ApplInitialise() ;
 34:   str:=RSCFileName ;
 35:   ResourceLoad(str) ;
 36:   IF AESCallResult=0 THEN
 37:     str:=AlertText ;
 38:     dummy:=FormAlert(1,str) ;
 39:     RETURN FALSE
 40:   END ;
 41:   ResourceGetAddr(RTree,DESKTOP,DeskTree) ;
 42:   TreeArray:=DeskTree;
 43:   RETURN TRUE
```

```
44: END Init ;
45:
46: VAR buff: ARRAY[0..9] OF INTEGER ;
47:     done: BOOLEAN ;
48:     dummy,x,y,w,h:INTEGER;
49:     mx,my,newx,newy,truex,truey:INTEGER;
50:     TreeArray:POINTER TO ARRAY [0..10] OF Object;
51:
52: BEGIN
53:   IF Init() THEN
54:     (* "Desktop" zentrieren und ausgeben *)
55:     FormCenter(DeskTree,x,y,w,h);
56:     ObjectDraw(DeskTree,DESKTOP,1,x,y,w,h);
57:     GrafMouse(Arrow,NIL);
58:     done:=FALSE;
59:     REPEAT
60:       (* auf ein Button-Ereignis warten *)
61:       IF EventButton(2,1,1,mx,my,dummy,dummy)=1 THEN
62:         (* einfacher Klick -> war Icon unter dem Zeiger ? *)
63:         IF ObjectFind(DeskTree,DESKTOP,1,mx,my)=ICON THEN
64:           (* Hand als Zeiger *)
65:           GrafMouse(FlatHand,NIL);
66:           (* absolute Position des Icons errechnen *)
67:           ObjectOffset(DeskTree,ICON,truex,truey);
68:           (* Box in der Größe des Icons innerhalb des
69:              "Desktop" bewegen *)
70:           GrafDragBox(TreeArray^[ICON].width,TreeArray^[ICON].height,
71:                       truex,truey,
72:                       TreeArray^[DESKTOP].x,TreeArray^[DESKTOP].y,
73:                       TreeArray^[DESKTOP].width,TreeArray^[DESKTOP].height,
74:                       newx,newy);
75:           (* neue Koordinaten setzen (Offset verrechnen !) *)
76:           TreeArray^[ICON].x:=newx-TreeArray^[DESKTOP].x;
77:           TreeArray^[ICON].y:=newy-TreeArray^[DESKTOP].y;
78:           (* "Desktop" neu ausgeben *)
79:           ObjectDraw(DeskTree,DESKTOP,1,x,y,w,h);
80:           (* Pfeil als Zeiger *)
81:           GrafMouse(Arrow,NIL);
82:         END;
83:       ELSE
84:         (* Doppelklick -> Ende *)
85:         done:=TRUE
86:       END;
87:     UNTIL done;
88:     (* fertig *)
89:   END;
90: END Desktop.
```

Die Zeilen 16 bis 18 werden vom Resource-Editor geliefert und enthalten die entsprechenden Konstanten der beiden Objekte.

In den Zeilen 24 bis 44 wird die Resource geladen und ihre Adresse festgestellt. Der Zeiger *TreeArray* (deklariert in Zeile 50) wird noch gebraucht, da das Programm direkt auf die Objekte zugreift.

Nachdem die Resource geladen wurde (Zeile 53), können Sie das kleine Desktop ausgeben und den Mauszeiger wieder als Pfeil setzen (Zeilen 55 bis 57). Mit *done* läßt sich das Programm abbrechen.

Was hat das Programm zu leisten? Es soll darauf warten, daß die linke Maustaste gedrückt wird. Falls die Maus auf das Icon zeigt, ist ein bewegliches Rechteck zu zeichnen, mit dem die neue Position des Icons festgelegt werden kann. Ist der Mausknopf wieder losgelassen, können Sie das Icon neu zeichnen. Mit einem Doppelklick wird das Programm verlassen.

Das Warten auf den Mausklick geschieht mit der *EventButton*-Funktion. Der erste Parameter gibt an, daß auf bis zu zwei Klicks zu warten ist. Die folgende 1 besagt, daß der linke Knopf verlangt wird. (Für den rechten müßte eine 2 stehen; für beide eine 3.) Die zweite 1 bedeutet, daß darauf gewartet werden soll, daß der linke Knopf gedrückt wird. (Bei einer 0 würde aufs Loslassen des Knopfs gewartet werden.)

Nun kommen die zwei wichtigsten Parameter, in die die Routine die Mauskoordinaten beim Erreichen des gewünschten Zustands der Mausknöpfe einschreibt. Die zwei *dummys* enthalten die Knopfzustände analog zum zweiten und dritten Parameter. Die Funktion liefert als Ergebnis die Anzahl der Mausklicks.

Bei einem Mausklick (dabei soll der Knopf übrigens gedrückt bleiben) schaut das Programm mit *ObjectFind* nach, ob sich das Icon unter dem Mauszeiger befindet (Zeile 63). Ist dies der Fall, schaltet es zunächst den Mauszeiger mit *GrafMouse* um. Auf dem Bildschirm erscheint eine ausgestreckte Hand (Zeile 65).

Nun wird mit *ObjectOffset* die absolute Position des *ICON*-Objekts festgestellt. *GrafDragBox* erlaubt es dem Benutzer, ein Rechteck innerhalb eines anderen mit der Maus zu bewegen. Mit dieser Aktion soll die neue Position des Icons festgelegt werden (Zeile 70).

Die ersten beiden Parameter geben die Breite und Höhe des beweglichen Rechtecks an. Ich hole diese direkt vom Objekt *ICON*, wobei ich mit dem *TreeArray*-Zeiger auf die Resource-Daten zugreife.

truex und *truey* geben die Startkoordinaten des besagten Rechtecks an. Hier sind die zuvor ermittelten Koordinaten des Icons erforderlich.

Die nächsten vier Parameter definieren einen Rahmen, in dem das Rechteck frei bewegt werden kann. Dies sind Position und Größe des Desktops. Die Größe hole ich direkt vom Objekt *DESKTOP*.

In *newx* und *newy* steht nach Beendigung von *GrafDragBox* die neue Position des Rechtecks, in dem das Icon erscheinen soll.

Da es sich hier um absolute Koordinaten handelt, aber die Koordinaten von Objekten bekanntlich relativ zum Parent-Objekt sind, können *newx* und *newy* nicht ohne Verrechnung der Position des *DESKTOP*-Objekts übernommen werden (Zeilen 76/77).

Sind die Koordinaten richtig gesetzt, braucht nur noch die gesamte Resource neu gezeichnet werden. Dabei beginne ich mit dem *DESKTOP*-Objekt, damit das alte Icon gelöscht (=überschrieben) wird (Zeile 79).

Durch einen Doppelklick soll das Programm verlassen werden. Das Programm setzt bei einem *EventButton*-Ergebnis von 2 die Variable *done* und verläßt somit die *REPEAT-UNTIL* Schleife.

Man muß natürlich bedenken, daß bei einem richtigen Programm auch auf andere Ereignisse gewartet werden muß. Dies betrifft vor allem die Menüs oder Fensterereignisse.

Ein Icon muß auch nicht unbedingt auf einem eigenen Hintergrund stehen. Schreibt man es direkt auf den Bildschirmhintergrund, so muß man den Rahmen bei *GrafDragBox* direkt angeben. Außerdem muß man dafür sorgen, daß das "alte" Icon wieder gelöscht wird.

9.8 Reagierende Objekte

Dieses kleine Beispiel demonstriert die Anwendung der Routine *GrafWatchBox*.

Die dazu benötigte Resource ist recht einfach. Sie besteht nur aus einer Dialogbox, in die ein *BoxText*-Objekt gesetzt wird.

Bild 9.8.1: Kleine Dialogbox

Die Dialogbox wird *BOX* genannt, und die Textbox erhält den Namen *HIER*.

Auf der folgenden Seite finden Sie ein kleines Programm, das zeigt, wie die *GrafWatchMouse*-Routine den Zustand eines Objekts verändert.

```
 1: MODULE Box;
 2:
 3: FROM SYSTEM       IMPORT ADDRESS;
 4:
 5: (* Alle benötigten Bibliotheksroutinen anfordern *)
 6: FROM GEMAESbase   IMPORT AESCallResult, RTree, Selected, Normal, Arrow ;
 7: FROM AESForms     IMPORT FormCenter ;
 8: FROM AESGraphics  IMPORT GrafWatchBox, GrafMouse ;
 9: FROM AESObjects   IMPORT ObjectDraw ;
10: FROM AESResources IMPORT ResourceLoad, ResourceGetAddr ;
11:
12: (* Konstante von MMRCP generiert *)
13: CONST
14:        BOX = 0 ;
15:       HIER = 1 ;
16:
17: VAR BoxTree :ADDRESS;
18:     inout,
19:     x,y,w,h,
20:     mx,my   :INTEGER;
21:
22: BEGIN
23:   (* Pfeil als Mauszeiger *)
24:   GrafMouse(Arrow,NIL);
25:   (* Resource laden *)
26:   ResourceLoad('BOX.RSC');
27:   IF AESCallResult#0 THEN
28:     (* Adresse feststellen *)
29:     ResourceGetAddr(RTree, BOX, BoxTree);
30:     (* zentrieren *)
31:     FormCenter(BoxTree,x,y,w,h);
32:     (* zeichnen *)
33:     ObjectDraw(BoxTree,BOX,1,x,y,w,h);
34:     (* GrafWatchBox ausführen *)
35:     inout:=GrafWatchBox(BoxTree, HIER, Selected, Normal);
36:   END;
37: END Box.
```

Die Resource wird geladen (Zeilen 26-27), zentriert (Zeile 31) und gezeichnet (Zeile 33).

Dann ruft das Programm *GrafWatchBox* auf (Zeile 35). Die Routine soll die Mausbewegungen daraufhin überwachen, ob sich der Mauszeiger in das Objekt *HIER* im Baum *BoxTree* bewegt oder nicht.

Befindet sich der Mauszeiger über dem Objekt, soll es den Zustand *Selected* annehmen, ansonsten soll es *Normal* sein. Dies funktioniert solange, bis der Mausknopf losgelassen wird. *inout* enthält je nachdem, ob der Zeiger über dem Objekt war oder nicht, eine Eins beziehungsweise eine Null.

Beim Programmstart muß der Mausknopf schon gedrückt sein! Sie können dann beobachten, wie das *HIER*-Objekt seinen Zustand automatisch ändert.

10. Resources in GFA-BASIC

Das recht beliebte und verbreitete GFA-BASIC bietet aufgrund seines interpretativen Charakters nur einen Bruchteil der Möglichkeiten von Resources. Trotzdem sind einige Befehle vorhanden, mit denen man eine GEM-Oberfläche schaffen kann.

10.1 Alerts

Für Alertboxen gibt es den Befehl

Alert zeichen,text$,default,button$,retbutton

zeichen enthält die Nummer des Icons, das in der Box dargestellt werden soll. Im String *text* wird der Text übergeben, der erscheinen soll. Mehrere Zeilen werden durch "|" getrennt. Es sind vier Zeilen mit maximal je 30 Zeichen erlaubt.

In *default* steht, welcher Button durch <Return> ausgewählt werden kann. Die vorhandenen Buttons werden in *button* angegeben und sind ebenfalls durch "|" getrennt. Es können drei Buttons mit jeweils bis zu 8 Zeichen verwendet werden.

retbutton enthält nach Durchführung des Alerts die Nummer des ausgewählten Buttons.

Da GFA-BASIC keine Dialog-Boxen zur Verfügung stellt, muß man an diesen Stellen auf Alertboxen ausweichen.

10.2 Menüs

Stärker zeigt sich GFA-BASIC bei den Möglichkeiten zur Menüdarstellung. Dabei wurde ein eigenes Konzept entworfen, das gut mit dem Interpreter zusammenpaßt.

Ein Menü wird vom Programm in einem Feld aus Zeichenketten dargestellt. Dieses Feld übernimmt GFA-BASIC und wandelt es in einen Menübaum um. Dazu dient der Befehl

Menu feld$()

Dadurch wird ein Menü übernommen, umgewandelt, dargestellt und kann sofort benutzt werden.

In *feld* stehen die Menütitel und -einträge als Zeichenketten. Dabei wird das Feld in mehrere Bereiche aufgeteilt, die jeweils ein Menü darstellen. Der erste Eintrag entspricht der Titelzeile, worauf die Einträge folgen. Abgeschlossen wird ein solcher Bereich durch einen leeren String.

Das gesamte Feld wird nochmals durch zwei Leer-Strings abgeschlossen, so daß die letzten beiden Feldeinträge leer bleiben (*feld$(n-1)=""* und *feld$(n)=""*).

Das erste Menü muß sechs Platzhalter-Strings für die Accessories bereitstellen. Ein Beispiel für diese Aufteilung gibt das zweite Beispielprogramm im nächsten Kapitel.

Ausschalten läßt sich das Menü mit

Menu Kill

Damit reagiert das Programm nicht mehr aufs Anwählen der Menüleiste. Allerdings bleibt die Titelzeile auf dem Bildschirm stehen und muß noch gelöscht werden.

Nach der Auswahl eines Menüpunkts bleibt der Menütitel invers auf dem Bildschirm stehen.

Mit dem Befehl

Menu Off

werden alle Menütitel wieder normal gezeichnet.

Nun zu den Befehlen, mit denen Menüeinträge verändert werden können:

Menu eintrag,0

Bei dem betreffenden Menüeintrag wird ein eventuell vorhandenes Häkchen gelöscht.

Setzen läßt sich das Häkchen mit

Menu eintrag,1

Zum Ausschalten von Menüeinträgen (Disablen) dient

Menu eintrag,2

Der Menüeintrag ist dann nicht mehr auswählbar.

Wieder eingeschaltet wird er mit

Menu eintrag,3

Leider gibt es keine Funktion, mit der sich der Text eines Menüeintrags verändern ließe. Um dieses Problem zu umgehen, muß man den Text im Menü-Feld ändern und dieses mit *Menu feld$()* wieder übernehmen.

Auf eine Menüauswahl wird reagiert, indem zunächst mit

On Menu Gosub menuehandler

festgelegt wird, daß bei Erkennung einer Menüauswahl zur Prozedur *menuehandler* gesprungen werden soll. Diese Aktion führt GFA-BASIC automatisch aus.

Damit eine Auswahl erkannt wird, muß GFA-BASIC zu einer Nachprüfung veranlaßt werden. Dazu dient

On Menu

Die Funktion prüft, ob ein Menüeintrag ausgewählt wurde. Wenn ja, wird zu der vorher angemeldeten Routine gesprungen, die auf die Auswahl reagiert. *On Menu* wartet nicht auf eine Menüauswahl; sie überprüft nur, ob eine vorlag. Also muß dieser Befehl in einer Schleife - möglichst der Hauptschleife eines Programms - so oft wie möglich ausgeführt werden.

Die Routine, die auf eine Menüauswahl reagieren soll, muß außerdem wissen, welcher Eintrag gewählt wurde. Dies erfährt sie mit

Menu(0)

Die Funktion liefert den Feldindex des ausgewählten Menüeintrags. Man kann dann auf die einzelnen Menüpunkte reagieren.

Mit

Menu(-1)

erhält man die Adresse des eigentlichen Menübaums. Man kann auch direkt darauf zugreifen, muß dabei aber sehr vorsichtig sein. Der Baum ist, in der am Anfang des Buches besprochenen Darstellung, im Speicher vorhanden.

10.3 Beispielprogramme

Die folgenden zwei Beispielprogramme sollen den Umgang mit den Resource-Möglichkeiten von GFA-BASIC demonstrieren.

Das erste Programm befaßt sich mit Alertboxen:

```
 1: ' erster Alert mit STOP-Zeichen
 2: Alert 3,"STOP !|Bitte auswählen",1,"Weiter|Schluß",Button
 3: ' wurde "Weiter" ausgewählt ??
 4: If Button=1 Then
 5:   ' zweiter Alert mit ?-Zeichen
 6:   Alert 2,"Wie bitte ?|Nochmal auswählen",2,"Weiter|Schluß",Button
 7:   ' wurde "Weiter" ausgewählt ??
 8:   If Button=1 Then
 9:     ' dritter Alert mit !-Zeichen
10:     Alert 1,"Achtung !|Gleich ist Schluß",1,"Weiter|Schluß",Button
11:     ' wurde "Weiter" ausgewählt ??
12:     If Button=1 Then
13:       ' letzter Alert ohne Zeichen
14:       Alert 0,"Das ist jetzt aber wirklich|die letzte Alertbox",1,
              "Schluß",Button
15:     Endif
16:   Endif
17: Endif
```

Dabei werden vier Alerts dargestellt, jeweils mit anderen Icons. In *Button* steht das Ergebnis des Alert-Dialogs, auf den dann reagiert wird.

Das zweite Programm entspricht dem schon bekannten Programm *MenuTest*, - nur diesmal in GFA-BASIC. Alle Funktionen dieses Programms können problemlos in GFA-BASIC programmiert werden.

```
 1: ' Feld für das Menü
 2: Dim Menu$(50)
 3: ' Einträge einlesen
 4: For I=0 To 50
 5:   Read Menu$(I)
 6:   Exit If Menu$(I)="."
 7: Next I
 8: ' Schluß des Feldes kennzeichnen
 9: Let Menu$(I)=""
10: Let Menu$(I+1)=""
11: '
12: ' DATAs für die Einträge
13: ' Desk Menü
14: Data Desk,  About Menutest...
15: Data ———————————————
16: ' Platz für die Accessories
17: Data 1,2,3,4,5,6,""
18: ' File Menü
19: Data File,  Quit  ,""
20: ' Options Menü
21: Data Options,  Option ein,  Disable Grafik,  Enable Grafik,  Häkchen
22: Data ———————————————,  Grafik,""
23: Data .
24: '
25: ' Menü anmelden
26: Menu Menu$()
27: Openw 0
28: ' Bei Menüauswahl nach "Menue" gehen
29: On Menu  Gosub Menue
30: ' Merker
31: Finish=False
32: Hakchen=False
33: Optein=True
34: Diable=False
35: '
36: ' immer wieder abfragen ...
37: Repeat
38:   On Menu
39:   ' ... bis "Quit" angewählt wurde
40: Until Finish
41: End
42: '
```

```
43: ' Hier wird auf eine Menüauswahl reagiert
44: '
45: Procedure Menue
46:   If Menu(0)=1 Then
47:     ' "About Menutest"
48:     ' Alert anzeigen
49:     Alert 1,"Disable Grafik : Menu x,2|Enable Grafik  : Menu x,3
        |Häkchen       : Menu x,0|                  Menu x,1",1,"OK",D
50:   Else
51:     If Menu(0)=11 Then
52:       ' "Quit"
53:       ' Programmende auslösen
54:       Finish=True
55:     Else
56:       If Menu(0)=14 Then
57:         ' "Option ein"/"Option aus"
58:         ' Test neu setzen
59:         If Optein Then
60:           Let Menu$(14)="  Option aus"
61:         Else
62:           Let Menu$(14)="  Option ein"
63:         Endif
64:         ' merken
65:         Optein=Not Optein
66:         ' anmelden
67:         Menu Menu$()
68:         ' eventuell Häkchen setzen
69:         If Hakchen Then
70:           Menu 17,1
71:         Endif
72:         ' eventuell disablen
73:         If Disable Then
74:           Menu 19,2
75:         Endif
76:       Else
77:         If Menu(0)=15 Then
78:           ' "Disable Grafik"
79:           ' Eintrag "Grafik" ausschalten
80:           Menu 19,2
81:           ' merken
82:           Disable=True
83:         Else
84:           If Menu(0)=16 Then
85:             ' "Enable Grafik"
86:             ' Eintrag "Grafik" einschalten
87:             Menu 19,3
88:             ' merken
89:             Disable=False
```

```
 90:          Else
 91:            If Menu(0)=17 Then
 92:              ' "Häkchen"
 93:              If Hakchen Then
 94:                ' Häkchen ausschalten
 95:                Menu 17,0
 96:              Else
 97:                ' Häkchen einschalten
 98:                Menu 17,1
 99:              Endif
100:              ' Zustand merken
101:              Hakchen=Not Hakchen
102:            Endif
103:          Endif
104:        Endif
105:      Endif
106:    Endif
107:  Endif
108:  ' Titel wieder normal
109:  Menu Off
110:  ' fertig
111: Return
```

Zunächst gilt es, das Feld für die Menüeinträge vorzubereiten. Nach dem Einlesen (Zeilen 4-7) werden die letzten beiden Felder als Leer-Strings gesetzt, um das Ende der Menüeinträge zu kennzeichnen (Zeilen 9/10).

Diese stehen in den *Data*-Zeilen (Zeilen 14-23). Die einzelnen Bereiche für das Desk-, File- und Options-Menü werden jeweils durch einen Leer-String abgeschlossen. Zeile 17 definiert die sechs Platzhalter für die Accessories.

Alle Einträge, die mit einem "-"-Zeichen beginnen, werden von GFA automatisch als Trennzeilen erkannt und sind im Menü nicht anwählbar (Zeilen 15 und 22).

Nun kann das Menü angemeldet werden (Zeile 26). Bei einer Auswahl soll die Routine *Menue* angesprungen werden (Zeile 29).

Anschließend erfolgen einige Voreinstellungen, die später noch gebraucht werden (Zeilen 31-34). Jetzt kann in einer Repeat-Until Schleife abgefragt werden, ob es eine Menüauswahl gab (Zeilen 37-40).

Menue wird bei einer Menüauswahl angesprungen und reagiert auf die einzelnen Menükommandos. Welcher Menüeintrag ausgewählt wurde, fragt das Programm mit *Menu(0)* ab.

War es "About Menutest ...", so wird einen kleine Alertbox ausgegeben (Zeilen 47-49). Bei "Quit" wird mit *Finished* die Schleife und damit das Programm verlassen (Zeile 54).

Die Reaktion auf "Option ein"/"Option aus" ist etwas kompizierter: Bei Anwahl dieses Eintrags soll sich der Text verändern. Dazu muß je nach aktuellem Zustand das betreffende Element im *Menu$*-Feld verändert werden (Zeilen 59-63).

Jetzt wird das Menü neu angemeldet, und der veränderte Text erscheint bei der Auswahl (Zeile 67). Allerdings befinden sich jetzt die Zustände der anderen Einträge wieder im Anfangszustand und sind zu korrigieren (Zeilen 69-75).

Bei "Disable Grafik" wird der "Grafik"-Eintrag ausgeschaltet. Dies geschieht einfach mit *Menu 19,2* (Zeile 80). Entsprechend wirkt beim Einschalten "Enable Grafik" (Zeile 87).

"Häkchen" soll, falls noch nicht vorhanden, einen Haken vorangestellt bekommen; anderenfalls wird er gelöscht. Dies geschieht mit *Menu 17,0* und *Menu 17,1* (Zeilen 93-101).

Nach dem Reagieren auf die jeweilige Auswahl muß noch der Menütitel normal dargestellt werden. Dazu verwendet das Programm *Menu Off* (Zeile 109).

Wie man sieht, läßt sich in GFA-BASIC recht einfach mit Resources umgehen, auch wenn hier nur wenige der vorhandenen Möglichkeiten zugänglich sind.

Wer mehr mit Resources arbeiten will (für Icons oder Dialogboxen) kommt nicht umhin, die Programmiersprache zu wechseln.

11. Resource-Programmierung mit ST Pascal Plus

Der Pascal-Compiler von CCD erfreut sich großer Beliebtheit. Neben den normalen GEM-Libraries stehen einige zusätzliche Routinen zur Verfügung, die von CCD als "Pascal Plus GEM-Bibliothek" zusammengefaßt werden.

11.1 Vorhandene Routinen

Die Routinen aus der GEM-Bibliothek bestehen aus zusammengesetzten Aufrufen der normalen GEM-Funktionen. Daher wird in dieser Beschreibung meist nur auf die schon bekannten GEM-Routinen verwiesen.

Init_Gem : INTEGER;
Exit_Gem;

Die Routinen entsprechen den Aufrufen *ApplInitialise* und *ApplExit*.

Do_Alert (Prompt:String; Default:Integer):Integer;

Die Funktion ist identisch mit *FormAlert*. Der einzige Unterschied besteht in der vertauschten Reihenfolge der Parameter.

Die Dialog-Funktionen dienen dazu, auf einfache Weise eine Resource im Speicher aufzubauen. Dadurch kann ein Resource-File entfallen.

New_Dialog(item_count,links,oben,breite,höhe:Integer):Dialog_Ptr;

Übergeben werden die Anzahl der Elemente des Dialogs sowie deren Ausmaße. Das Programm erhält einen Zeiger, der auf den internen Resource-Baum zurückweist. Bei den folgenden Routinen dient er zur Identifizierung des Dialogs.

```
Add_Ditem(box:Dialog_Ptr; itemtyp,flags,itemlinks,itemoben,
          itembreite,itemhoehe,Rahmen,Farbe:Integer):Integer;
```

Die Routine ergänzt den Dialog *box* um ein neues Element. Die Parameter geben den Objekttyp (*itemtyp*), sein Flags-Feld (*flags*), seine Größe, die Rahmendicke (*Rahmen*) und schließlich seine Farbe an. Diese Werte entsprechen den beim Objektaufbau beschriebenen Feldern. Es handelt sich also um einen geschickt parametrisierten Aufruf von *ObjectAdd*.

Das Funktionsergebnis ist der Index des Objekts im Baum. Dies entspricht den normalerweise vom Resource-Editor generierten Konstanten.

```
Set_Dtext(box:Dialog_Ptr; item:TreeIndex; text:Str255;
          font:Integer; justify:TE_Just);
```

Damit wird in einem Text-Objekt (Objekt *item* im Baum *box*) der Inhalt festgelegt. Dabei greift die Routine auf das dazugehörige TedInfo zu. Demnach finden sich auch alle Parameter im TedInfo-Record wieder (*text* = *ptext*, *font* = *font* und *justify* = *just*).

```
Set_Dedit(box:Dialog_Ptr; item:TreeIndex; maske:Str255;
          prüfung:Str255; init:Str255; font:Integer;
          justify:TE_Just);
```

Hiermit wird bei einem Edit-Objekt das TedInfo gesetzt. Daher sind auch die Parameter *maske* (=*ptmplt*), *prüfung* (=*pvalid*) und *init* (=*ptext*) nötig.

```
Get_DEdit(box:Dialog_Ptr; item:TreeIndex;
          VAR edited:Str255);
```

Nach einem Dialog kann mit der Funktion der Wert von *ptext* geholt werden, in dem der eingegebene Text steht.

```
Obj_SetState(box:Dialog_Ptr; item:TreeIndex; state:Integer;
             redraw:Boolean);
```

Diese Routine verändert das *State*-Feld eines Objekts. Mit *redraw* gleich *TRUE* wird das Objekt neu gezeichnet. Es handelt sich eigentlich um einen Aufruf von *ObjectChange*.

```
Obj_State(box:Dialog_Ptr; item:TreeIndex):Integer;
```

Die Funktion ergibt den Wert des *state*-Feldes beim angegebenen Objekt.

Obj_SetFlags(box:Dialog_Ptr; item:TreeIndex; Flags:Integer);

Die Routine setzt das *flags*-Feld eines Objekts entsprechend dem Parameter *Flags*.

Obj_Flags(box:Dialog_Ptr; item:TreeIndex):Integer;

Hiermit erhält man den Wert des *flags*-Feldes als Funktionswert zurück.

Center_Dialog(box:Dialog:Ptr);

Entspricht einem Aufruf von *FormCenter*, mit dem Unterschied, daß die Koordinaten der zentrierten Box nicht von der Prozedur zurückgegeben werden.

Do_Dialog(box:Dialog_Ptr; start_item:TreeIndex):Tree_Index;

Die Funktion faßt die Aufrufe von *FormDialogue*, *ObjectDraw* und *FormDo* zusammen, so daß ein Dialog ausgeführt wird. Es kann ebenfalls der Index des ersten editierbaren Feldes übergeben werden. Das Funktionsergebnis ist der Index des Objekts, über den der Dialog verlassen wurde.

Redo_Dialog(box:Dialog_Ptr; start_item:TreeIndex):Tree_Index;

Hiermit wird *FormDo* nochmals aufgerufen. Dabei muß der Dialog natürlich schon auf dem Bildschirm gezeichnet sein.

End_Dialog(box:Dialog_Ptr);

Dies ist ein Aufruf von *FormDialogue* mit dem Parameter *FormFinish*, woduch ein Dialog beendet wird.

Delete_Dialog(box:Dialog_Ptr);

Hierbei handelt es sich um eine Verwaltungsfunktion, die den Objektbaum wieder entfernt. Der belegte Speicher wird freigegeben und der Dialog kann nicht mehr benutzt werden.

Get_In_File (VAR path:Path_Name; VAR name:Path_Name):Boolean;

Hierdurch wird die GEM-FileSelectBox aufgerufen. Danach wird der ausgewählte File-Name konstruiert, so wie dies auch in der

Beschreibung von *FileSelectBox* als kleines Programm vorkam. Das Funktionsergebnis gibt an, ob der "ABBRUCH"-Button gedrückt wurde.

Get_Out_File(prompt:String; VAR name:Path_Name):Boolean;

Dadurch erscheint ein kleiner Dialog auf dem Bildschirm, in dem ein File-Name eingegeben werden kann. *prompt* ist ein kleiner Info-Text, der zusätzlich erscheint. Das Funktionsergebnis liefert die Angabe, ob der "ABBRUCH"-Button gedrückt wurde.

Es folgen die Menüfunktionen:

New_Menu(count:Integer; about:Str255):Menu_Ptr;

Hiermit wird Speicher für einen Menübaum reserviert. *count* gibt an, wie viele Menüeinträge benutzt werden sollen. In *about* steht ein Menüeintrag, der automatisch an die erste Stelle im Desk-Menü eingefügt wird. Das Funktionsergebnis dient zur Identifikation des Menübaumes in den folgenden Routinen.

Add_MTitle(menu:Menu_Ptr; title:Str255):Integer;

Die Routine fügt einen neuen Menütitel in den Baum *menu* ein. Das Funktionsergebnis ist der Index des Titels in der Resource.

Add_MItem(menu:Menu_Ptr; title_index:Integer;
item:Str255):Integer;

Hiermit wird ein Menüeintrag (*item*) im Menü *menu* unter dem Menütitel *title_index* eingefügt. Als Ergebnis erhält man wieder den Index des Eintrags.

Menu_Check(menu:Menu_Ptr; item_index:Integer; check:Boolean);

Entspricht einem Aufruf von *MenuItemCheck*.

Menu_Enable (Menu:Menu_Ptr; item_index:Integer);
Menu_Disable(Menu:Menu_Ptr; item_index:Integer);

Sind gleichwertig zu Aufrufen von *MenuItemEnable* mit den entsprechenden Parametern.

Menu_Highlight(menu:Menu_Ptr; title_index:Integer);
Menu_Normal (menu:Menu_Ptr; title_index:Integer);

Sind identisch mit Aufrufen von *MenuTitleNormal* mit gleichen Parametern.

Menu_Text(menu:Menu_Ptr; item_index:Integer; new_text:Str255);

Stimmt mit dem Aufruf von *MenuText* überein.

Draw_Menu(menu:Menu_Ptr);
Erase_Menu(menu:Menu_Ptr);

Erreicht ein Zeichnen beziehungsweise Löschen der Menüzeile durch einen Aufruf von *MenuBar* mit dem Parameter 1 oder 0.

Delete_Menu(menu:Menu_Ptr);

Entfernt einen Menübaum und gibt den belegten Speicher wieder frei. Das Menü kann danach nicht mehr benutzt werden.

Die Routinen für die Event- und Message-Verwaltung werden in ähnlicher Weise einfach mit den entsprechenden GEM-Routinen durchgeführt, wobei die Parameter die gleichen Bedeutungen haben.

Load_Resource(file:Path_Name):Boolean;

Ist identisch mit einem Aufruf von *LoadResource*.

Free_Resource;

Entspricht einem Aufruf von *ResourceFree*:

Find_Menu (index:Integer; VAR r_menu:Menu_Ptr);
Find_Dialog(index:Integer; VAR r_dial:Dialog_Ptr);
Find_Alert (index:Integer; VAR alert:Str255);

Diese drei Routinen sind identisch mit Aufrufen von *ResourceGetAddr* mit den entsprechenden Parametern. Man kann so auch direkt auf die Objekte zugreifen. Sie beziehen sich nur auf vorher geladene Resource-Dateien und nicht auf die mit obigen Routinen erzeugten internen Objekt-Bäume!

Wie man sieht, bietet diese GEM-Bibliothek nichts wesentlich Neues. Hier wurden nur Routinen zusammengefaßt und mit neuen Parameterstrukturen versehen. Alle diese Prozeduren lassen sich mit minimalen Aufwand auch in anderen Sprachen erreichen (dies ist auch mit dem normalen Pascal-System von CCD möglich!).

11.2 Beispielprogramme

In den ersten Beispielprogrammenen unter ST Pascal-Plus wird ein kleiner Dialog konstruiert und ausgeführt. Die Dialogbox soll wie folgt aussehen:

Bild 11.2.1: Die Dialogbox

In dem Programm müssen zunächst mit der *{$I <filename>}* Compiler-Anweisung die GEM-Bibliothek mitkompiliert werden (Zeilen 4-8 und 16).

In der Prozedur *Make_Dialog* wird die kleine Dialogbox mit den Routinen der GEM-Bibliothek konstruiert. Dazu muß der Dialog zunächst mit *New_Dialog* angemeldet werden (Zeile 23). Dabei übergeben werden die Ausmaße der Hintergrundbox übergeben und als Rückgabewert erhält man einen Kennwert für diesen Dialog. In dieser Box wird Platz für zehn Objekte reserviert.

Das erste Objekt ist eine einfache Textzeile vom Typ *GraphicText*. Es wird zunächst in die Box eingefügt (Zeile 25) und dann sein Text definiert (Zeile 28). Die Parameter bei *Add_DItem* und *Set_DText* ergeben sich aus der Größe des Texts, seiner Plazierung und seinen Farben.

Ebenso ist das zweite Textobjekt zu behandeln, nur lassen sich hier die Möglichkeiten des *GraphicBoxText*-Objekts besser ausnutzen (Zeilen 28-33).

Bei den beiden Buttons erhält der *Flags*-Parameter weitere Bedeutung, da die Buttons auswählbar sein sollen, und einer von ihnen das *Default*-Objekt ist (Zeilen 35-43).

Die Ergebnisse der *Add_DItem*-Funktionen werden später noch für die Ausführung des Dialogs gebraucht. Wie Sie sehen, läßt sich so eine Resource auch ohne Resource-Editor erstellen.

Handle_Dialog soll den Dialog durchführen. Dazu wird er zunächst mit *Center_Dialog* zentriert (Zeile 51) und dann mit *Do_Dialog* ausgeführt (Zeile 53). Der Kennwert des ausgewählten Buttons wird in *result* geschrieben.

Wurde "Weiter" ausgewählt, so soll das Programm den Dialog nochmals ausführen. Davor muß aber erst der "Weiter"-Button normal gesetzt und neu gezeichnet werden. Dazu dient *Obj_SetState* mit den entsprechenden Parametern (Zeile 59).

Nun kann mit *Redo_Dialog* die Box erneut in Aktion treten (Zeile 61). Dies geschieht solange, bis "Ende" ausgewählt wird. Schließlich wird der Dialog mit *End_Dialog* beendet.

Forget_Dialog dient dazu, die Box wieder aus dem Speicher zu entfernen. Das geschieht einfach mit *Delete_Dialog* unter Angabe der Dialog-Kennzahl (Zeile 72).

Das Hauptprogramm ruft die drei Prozeduren nach der Anmeldung bei GEM (Zeile 77) auf und meldet sich anschließend wieder ab (Zeile 82).

Das Listing des Programms:

```
  1: PROGRAM Dialog;
  2:
  3: { GEM-Bibliothek mitcompilieren }
  4: CONST
  5:   {$I GEMCONST.PAS}
  6:
  7: TYPE
  8:   {$I GEMTYPE.PAS}
  9:
 10: VAR
 11:   dummy:INTEGER;                  { allg. Verwendung }
 12:   mydialog:Dialog_Ptr;            { Zeiger für unseren Dialog }
 13:   weiter_btn, ende_btn:INTEGER;   { Kennzahlen für die Buttons }
 14:
 15: { GEM-Bibliothek mitcompilieren }
 16: {$I GEMSUBS.PAS}
 17:
 18: PROCEDURE Make_Dialog;
 19: { Dialogbox erstellen }
 20: VAR str1,str2:INTEGER;
 21: BEGIN
 22:   { Dialog anmelden }
 23:   mydialog:=New_Dialog(10,0,0,25,10);
 24:   { ersten Textstring installieren ... }
 25:   str1:=Add_DItem(mydialog,G_Text,None,5,1,14,1,0,
```

```
26:                    white*4096+Black*256);
27:   { ... und setzen }
28:   Set_DText(mydialog,str1,'Dies ist eine',System_Font,TE_Center);
29:   { zweiten Textstring installieren ... }
30:   str2:=Add_DItem(mydialog,G_BoxText,None,3,3,19,4,1,
31:                    Black*4096+Black*256+1*128+2*16+Black);
32:   { ... und setzen }
33:   Set_DText(mydialog,str2,'Dialogbox',Small_Font,TE_Center);
34:   { ersten Button installieren ... }
35:   weiter_btn:=Add_DItem(mydialog,G_Button,Selectable+Exit_Btn,
36:                         3,8,8,1,0,Black*4096+Black*256);
37:   { ... und Text setzen }
38:   Set_DText(mydialog,weiter_btn,'Weiter',System_Font,TE_Center);
39:   { zweiten Button installieren ... }
40:   ende_btn:=Add_DItem(mydialog,G_Button,Selectable+Exit_Btn+Default,
41:                       14,8,8,1,0,Black*4096+Black*256);
42:   { ... und Text setzen }
43:   Set_DText(mydialog,ende_btn,'Ende',System_Font,TE_Center);
44: END;
45:
46: PROCEDURE Handle_Dialog;
47: { Dialog durchführen }
48: VAR result:INTEGER;
49: BEGIN
50:   { Dialog zentrieren }
51:   Center_Dialog(mydialog);
52:   { Dialog ausführen }
53:   result:=Do_Dialog(mydialog,0);
54:   IF result=weiter_btn THEN
55:   { Falls 'Weiter' gewählt wurde ... }
56:   BEGIN
57:     REPEAT
58:       { 'Weiter'-Button wieder normal setzen }
59:       Obj_SetState(mydialog,weiter_btn,Normal,TRUE);
60:       { Dialog nochmal ausführen }
61:       result:=Redo_Dialog(mydialog,0);
62:     { ... bis 'Ende' gewählt wurde }
63:     UNTIL result=ende_btn;
64:   END;
65:   { Dialog beenden }
66:   End_Dialog(mydialog);
67: END;
68:
69: PROCEDURE Forget_Dialog;
70: BEGIN
71:   { Dialog aus dem Speicher entfernen }
72:   Delete_Dialog(mydialog);
73: END;
74:
```

```
 75: BEGIN
 76:   { Programm bei GEM anmelden }
 77:   dummy:=Init_Gem;
 78:   Make_Dialog;
 79:   Handle_Dialog;
 80:   Forget_Dialog;
 81:   { Programm bei GEM abmelden }
 82:   Exit_Gem;
 83: END.
```

Das zweite Beispielprogramm demonstriert die Menübehandlung mit der GEM-Bibliothek.

```
  1: PROGRAM Hakchen;
  2:
  3: { GEM-Bibliothek einlesen }
  4: CONST
  5:   {$I GEMCONST.PAS}
  6:
  7: TYPE
  8:   {$I GEMTYPE.PAS}
  9:
 10: VAR
 11:   mymenu:Menu_Ptr;              { Zeiger auf unser Menü }
 12:   deskmenu, filemenu, optmenu,  { Indizes der Menüeinträge }
 13:   quit,
 14:   disable,enable,dummy,
 15:   Hakchen,grafik               : INTEGER;
 16:   info  : RECORD               { Zum Merken }
 17:             hakchen:BOOLEAN;
 18:           END;
 19:
 20:   {$I GEMSUBS.PAS}
 21:
 22: PROCEDURE Make_Menu;
 23: { Menü aufbauen }
 24: BEGIN
 25:   { neues Menü, Infozeile 'About...' }
 26:   mymenu :=New_Menu(25,'  About Menutest ...');
 27:
 28:   { Filemenü erstellen }
 29:   filemenu:=Add_MTitle(mymenu,' File ');
 30:   { Menüeintrag einfügen }
 31:   quit    :=Add_MItem(mymenu,filemenu,'  Quit ');
 32:
 33:   { Optionsmenü erstellen }
 34:   optmenu :=Add_MTitle(mymenu,' Options ');
 35:   { Menüeinträge einfügen }
```

```
36:   disable :=Add_MItem(mymenu,optmenu,'  Disable Grafik ');
37:   enable  :=Add_MItem(mymenu,optmenu,'  Enable Grafik  ');
38:   hakchen :=Add_MItem(mymenu,optmenu,'  Häkchen        ');
39:   dummy   :=Add_MItem(mymenu,optmenu,'----------------');
40:   grafik  :=Add_MItem(mymenu,optmenu,'  Grafik         ');
41:
42:   { Trennlinie ausschalten }
43:   Menu_Disable(mymenu,dummy);
44:   { Häkchen nicht gesetzt }
45:   info.hakchen:=FALSE;
46: END;
47:
48: PROCEDURE Handle_Menu;
49: VAR finished: BOOLEAN;
50:     event   : INTEGER;
51:     buffer  : Message_Buffer;
52: BEGIN
53:   finished:=FALSE;
54:   { Menüzeile anzeigen }
55:   Draw_Menu(mymenu);
56:   REPEAT
57:     { auf Mitteilung warten }
58:     event:=Get_Event(E_MESSAGE, 0, 0, 0, 0,
59:                      FALSE, 0, 0, 0, 0, FALSE, 0, 0, 0, 0,
60:                      buffer, dummy, dummy, dummy, dummy, dummy, dummy);
61:     IF ((event & E_MESSAGE)<>0) AND (buffer[0]=MN_Selected) THEN
62:     { Menüauswahl verarbeiten }
63:     BEGIN
64:       { 'About ...' ausgewählt }
65:       IF buffer[3]=3 THEN
66:         { Alert anzeigen }
67:         dummy:=Do_Alert('[1][Über Menütest][OK]',1)
68:       ELSE
69:         IF buffer[3]=filemenu THEN
70:         { Filemenü ausgewählt }
71:         BEGIN
72:           { Eintrag 'Quit'. '+1' wegen GEM-Fehler ! }
73:           IF buffer[4]=quit+1 THEN
74:             { Programm beenden }
75:             finished:=TRUE;
76:         END
77:         ELSE
78:           IF buffer[3]=optmenu THEN
79:           { Optionsmenü ausgewählt }
80:           BEGIN
81:             IF buffer[4]=disable THEN
82:               { 'Grafik' ausschalten }
83:               Menu_Disable(mymenu,grafik);
84:             IF buffer[4]=enable THEN
```

```
 85:              { 'Grafik' einschalten }
 86:              Menu_Enable(mymenu,grafik);
 87:            IF buffer[4]=hakchen THEN
 88:            BEGIN
 89:              IF info.hakchen THEN
 90:                { Häkchen löschen }
 91:                Menu_Check(mymenu,hakchen,FALSE)
 92:              ELSE
 93:                { Häkchen setzen }
 94:                Menu_Check(mymenu,hakchen,TRUE);
 95:              { Zustand merken }
 96:              info.hakchen:=NOT info.hakchen
 97:            END;
 98:          END;
 99:       { Menütitel wieder normal }
100:       Menu_Normal(mymenu,buffer[3]);
101:     END;
102:   UNTIL finished;
103: END;
104:
105: PROCEDURE Forget_Menu;
106: BEGIN
107:   { Menüzeile löschen }
108:   Erase_Menu(mymenu);
109:   { Menü entfernen }
110:   Delete_Menu(mymenu);
111: END;
112:
113: BEGIN
114:   { Programm anmelden }
115:   dummy:=Init_Gem;
116:   Make_Menu;
117:   Handle_Menu;
118:   Forget_Menu;
119:   { Programm abmelden }
120:   Exit_Gem;
121: END.
```

Zunächst muß wieder mit der *$I*-Compiler-Anweisung die Bibliothek mitkompiliert werden (Zeilen 4-8 und 20). In der Routine *Make_Menu* wird das Menü definiert. Zunächst wird mit *New_Menu* (Zeile 26) Platz für das Menü reserviert. Die erste Zeile im "Desk"-Menü soll dabei "About Menutest ..." lauten.

Nun lassen sich die beiden Menüs installieren. Den Titel setzt das Programm jeweils mit *Add_MTitle* (Zeilen 29 und 34). Die Einträge werden mittels *Add_MItem* eingetragen (Zeilen 31 und 36 bis 40). Die Ergebniswerte dieser Funktionen vermerkt das Programm.

Der Trennstrich im "Options"-Menü soll natürlich nicht anwählbar sein und ist deshalb mit *Menu_Disable* auszuschalten (Zeile 43). Schließlich merkt sich das Programm, ob beim Eintrag "Häkchen" im Menü "Options" ein Häkchen gesetzt ist (Zeile 45).

Die Prozedur *Handle_Menu* ist das eigentliche Hauptprogramm und es reagiert auf die Menü-Auswahl. Zuerst muß jedoch das Menü mit *Draw_Menu* angezeigt werden (Zeile 55).

Es folgt das Warten auf ein Mitteilungsereignis (Zeile 58). Bei der Mitteilung "Ein Menüeintrag wurde ausgewählt", reagiert das Programm auf die unterschiedlichen Auswahlmöglichkeiten. *buffer[3]* enthält dabei den Menütitel und *buffer[4]* den Eintrag.

Die erste Zeile im "Desk"-Menü erhält immer die Nummer 3. Dort steht der "About ..."-Eintrag. Das Programm reagiert auf seine Auswahl mit einer kleinen Alarmbox (Zeilen 65-67).

Bei der Auswahl von "Quit" im "File"-Menü soll das Programm verlassen werden. Dies geschieht durch Setzen der *finished*-Variablen.

In diesem Fall stimmt die von *Add_MItem* in *Make_Menu* zurückgegebene Kennzahl für den "Quit"-Eintrag allerdings nicht. Der wirkliche Wert liegt genau um eins höher. Warum dieser Fehler auftritt, ist nicht verständlich. Ich kann nur aus dem Handbuch zu ST Pascal-Plus, Seite 79 zitieren:

> "In der derzeit aktuellen GEM Version existiert ein Fehler, der zu unvorhersehbaren Resultaten führen kann. Man sollte zur Umgehung dieses Fehlers Menüeinträge grundsätzlich in Gruppen hinzufügen, und zwar in der Reihenfolge, in der die Menütitel eingetragen wurden."

Man kann sich also nicht darauf verlassen, daß die von *Add_MItem* gelieferten Werte stimmen. Eine Umgehung dieses Fehlers ist nur durch Ausprobieren möglich. Ärgerlich, aber leider nicht einfach zu beheben!

Jetzt komme ich zum "Options"-Menü (bei dem übrigens alle Werte stimmen!). Der erste Eintrag "Disable Grafik" soll bewirken, daß der Eintrag "Grafik" nicht mehr anwählbar ist. Die geschieht per *Menu_Disable* mit Angabe der entsprechenden Parameter (Zeile 83).

Bei Auswahl von "Enable Grafik" passiert genau das Gegenteil; "Grafik" wird wieder anwählbar. Dazu verwendet das Programm *Menu_Enable* (Zeile 86).

Wurde "Häkchen" ausgewählt, soll das Häkchen von diesem Eintrag gesetzt oder gelöscht werden. Das läßt sich mit *Menu_Check* erreichen (Zeilen 89 bis 96).

Nach erfolgter Auswahl muß mit *Menu_Normal* der Titel der ausgewählten Menüs wieder normal dargestellt werden. Damit ist diese Routine auch schon beendet. Sie kann - wie das Modula-2-Programm - als Grundgerüst für eine Menüauswahl verwendet werden.

Die dritte Routine - *Forget_Menu* - des Programms sorgt dafür, daß das vom Programm definierte Menü wieder entfernt wird.

Durch *Erase_Menu* wird die Menüzeile gelöscht und GEM mitgeteilt, daß dieses Menü nicht mehr gültig ist. *Delete_Menu* entfernt den Menübaum aus dem Speicher (Zeilen 108-110). Er kann nun nicht mehr angesprochen werden.

Nachdem sich das Hauptprogramm mit *Init_Gem* bei GEM angemeldet hat (Zeile 115), kann es Mitteilungen erhalten und Menüs benutzen. Dann werden die drei Prozeduren zum Erzeugen, Benutzen und Löschen des Menüs aufgerufen. Nach Abschluß dieser Prozedur meldet sich das Programm mit *Exit_Gem* ab (Zeile 120).

12. Anhang

A Modula-2- und C-Bibliotheksroutinen

Modula-2	C- bzw. Pascal-Bibliotheken
ApplInitialise	appl_init
ApplRead	appl_read
ApplWrite	appl_write
ApplFind	appl_find
ApplTPlayback	appl_tplay
ApplTRecord	appl_trecord
ApplExit	appl_exit
EventKeyboard	evnt_keybd
EventButton	evnt_button
EventMouse	evnt_mouse
EventMessage	evnt_mesag
EventTimer	evnt_timer
EventMultiple	evnt_multi
EventDoubleClick	evnt_dclick
FormDo	form_do
FormDialogue	form_dial
FormAlert	form_alert
FormError	form_error
FormCenter	form_center
FileSelectorInput	fsel_input
GrafRubberBox	graf_rubberbox
GrafDragBox	graf_dragbox
GrafMoveBox	graf_movebox
GrafGrowBox	graf_growbox
GrafShrinkBox	graf_shrinkbox
GrafWatchBox	graf_watchbox

Modula-2	C- bzw. Pascal-Bibliotheken
GrafSlideBox	graf_slidebox
GrafHandle	graf_handle
GrafMouse	graf_mouse
GrafMouseKeyboardState	graf_mkstate
MenuBar	menu_bar
MenuItemCheck	menu_icheck
MenuItemEnable	menu_ienable
MenuTitleNormal	menu_tnormal
MenuText	menu_text
MenuRegister	menu_register
ObjectAdd	objc_add
ObjectDelete	objc_delete
ObjectDraw	objc_draw
ObjectFind	objc_find
ObjectOffset	objc_offset
ObjectOrder	objc_order
ObjectEdit	objc_edit
ObjectChange	objc_change
ResourceLoad	rsrc_load
ResourceFree	rsrc_free
ResourceGetAddr	rsrc_gaddr
ResourceSetAddr	rsrc_saddr
ResourceObjectFix	rsrc_obfix
WindowCreate	wind_create
WIndowOpen	wind_open
WindowClose	wind_close
WindowDelete	wind_delete
WindowGet	wind_get
WindowSet	wind_set
WindowFind	wind_find
WindowUpdate	wind_update
WindowCalc	wind_calc

B Literaturhinweise

Abraham/Englisch u.a.: Atari ST GEM, Düsseldorf 1986, DATA BECKER VERlAG

Oren, Tim: Professional GEM, Antic Publishing
(Diese Artikelserie besteht aus mehreren Textdateien, die auf Diskette als Public-Domain verfügbar sind. Wenn Sie also jemand kennen, der diese Dateien hat, können sie ohne Copyright-Verstoß kopiert werden. Die Lektüre der Artikelserie ist sehr empfehlenswert! Teile davon sind auf der Diskette zum Buch vorhanden.)

Szczepanowski/Günther: Das große GEM-Buch, Düsseldorf 1985, DATA BECKER Verlag
(Dieses Buch ist im Endeffekt eine Übersetzung der Original-GEM-Handbücher. Leider wird nicht sehr weit auf die GEM-Programmierung eingegangen. Die Autoren beschränken sich auf eine Auflistung der einzelnen GEM-Funktionen.)

TDI Modula-2/ST User's Manual, Bristol 1986, TDI Software Ltd.
(Dies ist das Handbuch zum Modula-2 System von TDI, mit dem auch die Programme in diesem Buch entwickelt wurden. Interessant ist das Beispielprogramm "GEMDEM". Hier werden viele GEM-Funktionen benutzt und das Listing gibt somit eine allererste Anleitung zur GEM-Programmierung.)

Dal Cin/Lutz/Risse: Programmierung in Modula-2, Stuttgart 1984, Teubner-Verlag
(Dies ist ein Lehrbuch zur Programmierung in Modula-2. Gegenüber dem Original-Buch von N.Wirth bietet es in etwa den gleichen Inhalt zu einem erheblich niedrigerem Preis!)

Weiterhin sind viele Public-Domain-Programme (also Programme, die ohne Copyright-Verletzung beliebig kopiert und weitergegeben werden können) auch mit Programm-Listings erhältlich, in denen viele anschauliche Tips und Tricks zur GEM-Programmierung zu finden sind.

C Sachwortregister

O

R

S

T

W

X

Y