GFA BASIC BOOK

An intermediate programming tutorial for the best selling BASIC... GFA BASIC.

for the Atari ST

MichTron

The GFA BASIC Book

An Intermediate Tutorial

For the GFA BASIC Interpreter

With Disk Enclosed

For the Atari ST Series of Personal Computers

Written by Frank Ostrowski,
GFA Systemtechnik

Distributed By MichTron, Inc.
576 South Telegraph
☎: (313) 334-5700
BBS: (313) 332-5452

YOUR RIGHTS AND OURS: This copy of ***THE GFA BASIC BOOK*** is licensed to you. You may make copies of the disk for your own use or for archival storage. You may also sell your copy without notifying us. However, we retain copyright and other property rights in the program code and documentation. We ask that ***THE GFA BASIC BOOK*** be used either by a single user on one or more computers or on a single computer by one or more users. If you expect several users of ***THE GFA BASIC BOOK*** on several computers, contact us for quantity discounts and site-licensing agreements. Also if you intend to rent this program, or place this program on a BBS, contact us for the appropriate license and fee.

We think this user policy is fair to both you and us; please abide by it. We will not tolerate use or distribution of all or part of ***THE GFA BASIC BOOK*** or its documentation by any other means.

LIMITED WARRANTY: In return for your understanding of our legal rights, we guarantee ***THE GFA BASIC BOOK*** will reliably perform as detailed in this documentation, subject to limitations here described, for a period of thirty days. If ***THE GFA BASIC BOOK*** fails to perform as specified, we will either correct the flaw(s) within 15 working days of notification or let you return ***THE GFA BASIC BOOK*** to the retailer for a full refund of your purchase price. If your retailer does not cooperate, return ***THE GFA BASIC BOOK*** to us. While we can't offer you more cash than we received for the program, we can give you this choice: 1) you may have a cash refund of the wholesale price, or 2) you may have a merchandise credit for the retail price, which you may apply toward buying any of our other software. Naturally, we insist that any copy returned for refund include proof of the date and price of purchase, the original program disk, all packaging and documentation, and be in salable condition.

If the ***THE GFA BASIC BOOK*** disk becomes defective within the warranty period, return it to us for a free replacement. After the warranty period, we will replace any defective program disk for $5.00.

We cannot be responsible for any damage to your equipment, reputation, profit-making ability or mental or physical condition caused by the use (or misuse) of our program.

We cannot guarantee that this program will work with hardware or software not generally available when this program was released, or with special or custom modifications of hardware or software, or with versions of accompanying or required hardware or software other than those specified in the documentation.

Under no circumstances will we be liable for an amount greater than your purchase price.

Please note: Some states do not allow limitations on how long an implied or express warranty lasts, or the exclusion or limitation of incidental or consequential damages, so some of the above limitations or exclusions may not apply to you.

UPGRADES AND REVISIONS: If you return your information card, we will notify you if upgrades to ***THE GFA BASIC BOOK*** become available. For minor upgrades and fixes, return the original disks to us with $5.00. For major revisions, the upgrade fee is typically 15-20% of [illegible] original suggested retail price.

FEEDBACK: Customer comments are VERY important to us. We think that the use, warranty and upgrade policies outlined above are among the fairest around. Please let us know how you feel about them.

Many of the program and documentation modifications we make result from customer suggestions. Please tell us how you feel about ***THE GFA BASIC BOOK*** - your ideas could make the next version better for all of us.

The GFA BASIC Book

Published in the U.S.A. by MichTron, Inc.
576 South Telegraph
Pontiac, Michigan 48053

1st English Edition: September 1987
2nd English Edition: April 1988

Manual Written by Frank Ostrowski
Text Translated by Wilford Niepraschk
Book Design by Thomas L. Logan
Cover Design by Paul Deckard

88 89 90 91 92 10 9 8 7 6 5 4 3 2

ISBN 0-923213-85-6

Printed in the United States

Table of Contents

List of Diagrams

INTRODUCTION

THE ORIGINS OF GFA BASIC

It began with an *Atari 400*, a small computer similar to the *Commodore 64* There existed a BASIC for that computer. This BASIC was neither fast nor comfortable to use, but, with only 16K of *RAM*, large programs could not be written for it anyway. After upgrading the computer to 48K of memory and 88K of disk space, I wrote some programs in assembly code. Eventually I ended up with a number of help routines, and a BASIC language with which I could marginally use these routines. After trying FORTH, I decided to take a closer look at BASIC, and slightly modified it. These modifications eventually became so numerous that I decided to completely replace many of the routines. To remain compatible, some of the routines were left untouched. There were a lot of commands I did not like, such as computed *GOTO*s, and line numbers were a nuisance.

This new BASIC was published in a computer magazine. Shortly thereafter I received an offer from GFA Systemtechnik GmbH to write a workable BASIC for a new computer, the *Atari ST*.

The *Atari ST* incorporates a fresh modern processor. Its operating system, although certainly not the newest or fastest (nor is it very compact, having been written in *C* language) is very powerful. And, even though it lacks multitasking, one can write programs in high level languages that offer exciting performance.

Shipped with a BASIC language that did not even measure up to the one included with the Atari 400, the *ST* was destined to become a language developers dream. It was possible now to develop a BASIC that did not have to conform to the standard of any other interpreter.

This new BASIC should have the simplicity of BASIC combined with the possibility of writing well structured code. The first step was to eliminate the line numbers. This made the task difficult from the outset because a solution had to be found to avoid the usual confusion of *GOTOs* and *GOSUBs*. It was important to be able to pass parameters to procedures and to declare local variables, thus enabling the programmer to use *recursive* programming techniques. The BASIC should also make sure that all loops are properly closed before the program starts execution.

The *GOTO* statement was one of the last statements added to this BASIC. After much thought, I even allowed the *GOTO* command to be used between different procedures.

In an Interpreter it is possible to use segmented *PEEK*s and *POKE*s to simulate one of the Intel-processors. In a compiled program, this would greatly affect execution time.

The unsuitable 16 bit integers would not be used either, as this makes it harder to address all of the memory. Besides, the processor already uses 32 bits internally, thus allowing it to process larger numbers without speed loss.

The editor of this BASIC had to be *screen oriented* and *not* use the windows of *GEM*. It would be virtually impossible to create non *GEM* programs from within the *GEM* interface. Other reasons exist for not having the editor run under *GEM*.

In the case of a program error, it is often possible to save program changes that were made. Something that cannot always be done from within *GEM* because the windows lock up. So it happened that a relatively fast editor, one that could be used without a mouse, was created.

I wanted to write the BASIC completely in machine language so that it would be fast and take up only a small portion of memory. Other languages like *C* use only a few machine instructions outside of the library, and they always pass parameters through the stack. The MC68000 has a very powerful instruction set that can be better utilized with an Assembler.

Taking all this into consideration, Version 1.0 of **GFA BASIC** came into existence less then 6 months later.

While I was writing the interpreter, I carefully made sure that the finished programs could be easily compiled. That is why the *MERGE* command is missing. This command may be useful in an interpreter, but is of little value in a compiler.

While I was working on the compiler, I was confronted with requests to expand the command set. Some of those requests I was able to incorporate in Version 2.0 of the interpreter. Most new commands, like *VOID*, *BASEPAGE*, and *OPTION*, were inserted to give the compiler more optimization opportunities, and to provide the programmer with more control over the compiling process.

Even an extensive computer language cannot fulfill all the wishes of everyone who uses it. This book will, therefore, show you how to create necessary routines using **GFA BASIC**.

This book does *not* present you with completed applications; it gives routines that can be incorporated into your own programs.

GFA BASIC is only a BASIC and not a *Modula 2*, therefore you cannot create modules in quite the same way. *Modula 2* takes a lot more coding and a multitude of small modules to write an application. *Modula 2* can only be used as a compiler. BASIC includes numerous commands that would have to be created within libraries in *Modula 2*.

Part of this book discusses many different operating system routines which include examples wherever I felt they were needed. Naturally, there are routines that can be run directly without going to the low-level operating system routines, but rather with built in commands.

An important part of the book is the last chapter, where a complete *GEM* program is shown. It demonstrates how to use all parts of a window. This is not easy to do in *GEM*, but it makes it convenient for the user of the program. For the programmer, *GEM* always means extra work. There are many programs where most of the code is written just to manipulate the window.

I hope that you find the routines and tips in this book useful, and I wish you much success.

CHAPTER 1

OPTIMIZATION

After you have written a good program you naturally want to distribute it, sell it, or use it yourself. Now you discover that the program runs, but it is unacceptably slow.

The first step toward *Optimization* is to determine which part of the program takes up so much time...

1.1 Title Screen

Often a program displays a graphic screen which contains many pieces of information inside little rectangles. The remaining area of these rectangles is filled using the *FILL* command. This *FILL* command takes up a lot of time, and, if used every time you return to the main menu, it could easily daunt you with its slowness. It would be better to use the *PBOX* command on the background and the *PBOX* command on the foreground, but without a fill pattern.

Or you can draw the title picture once and then use the *SGET/SPUT* command to quickly display it on the screen. This has the disadvantage that 32K bytes of memory are needed to store the picture. But this is usually not a problem on the *1040ST* or *MEGA ST*.

A third method is to use many screen pages with a method called *page_flipping* (*see graphics without flicker*).

The final, and most elegant, method is to use a *RCS* file that will create the screen almost by itself. This usually means more coding, but it is advantageous in that you can change the screen independently of the program. If this is still too slow then you will have to wait until the blitter chip becomes available.

1.2 Diskette data

Another source of slowness is receiving data from diskettes. Take this for example:

```
OPEN "O",#1,"TEST.DAT"
FOR I%=0 TO 999
    PRINT #1,A(I%)
NEXT I%
CLOSE #1
OPEN "I",#1,"TEST.DAT"
FOR I%=0 TO 999
    INPUT #1,A(I%)
NEXT I%
CLOSE #1

BSAVE "TEST.DAT",VARPTR(A(0)),6000
BLOAD "TEST.DAT",VARPTR(A(0))
```

The first routine takes about six times as long as the *BSAVE* command (four times for a hard disk) and about twelve times as long to read compared to the *BLOAD* command (40 times with a hard disk). The *BSAVE* command takes about 6000 bytes of file space while the print command takes anywhere between 3000 and 20,000 bytes (depending on the number: "1" to "-1.2345678901E+123" and a *CR-LF* character sequence as a separator.

Reading from a diskette using:

```
OPEN "I',#1,"TEST.DAT"
BGET #1,VARPTR(A(0)),6000
CLOSE #1
```

is quicker than *BLOAD*, but not on the hard disk. Also *A(I%)=CVF(INPUT$(6,#1))* is pretty fast, but *PRINT*#1,MKF$(A$(I%)) is not.

If you would like to write your program so that you are able to transfer data to future *GFA BASIC* versions that might have different internal number representation, you may want to use *PRINT/INPUT*. If the program is converted to the new interpreter, you can then write a conversion program to convert like this:

```
numsize=VARPTR(a(1))-VARPTR)a(0))
BSAVE "TEST. DATA",VARPTR(a(0)),1000*numsize
```

1.3 Calculations

If the program spends a lot of its time computing (*SIN/COS*...), you have the following options:

① Add a floating point processor (68881)

Advantages
Very quick
Little or no programming changes

Disadvantages
Computer needs to be modified (soldering?)
Expensive (68881 costs several hundred dollars (*now!!*))
Runs only on a computer that is modified

② Search for faster algorithms:

Advantages
No hardware changes
Often faster than with the 68881

Disadvantages
Often very difficult and time consuming during development

This leaves you with two choices: expensive hardware or expensive software, where the latter choice represents a true accomplishment. Anyone can make a program run faster by improving the hardware — *if* you have the money. (This is why I have asked you to please not pirate software, because even in short programs a lot of mental work has often been invested). No one can help you find new algorithms, but by studying mathematic books and magazines you can often find your own. Computer magazines like *BYTE*, etc, are also very helpful.

Programs can also be optimized without changing the existing algorithm.

FOR-NEXT loops should not use floating point variables. They should use integers instead. This is especially true if those variables are used to index an array.

For compiled programs:

```
FOR i%=1 to 1000
  ......
NEXT i%
```

should be replaced with:

```
i%=1
REPEAT
  .......
  INC i%
UNTIL i%>1000
```

- Use *INC a* or *INC a%* instead of *a=a+1* or *a%=a%+1*
- Calculate numbers in advance (like *deg.rad=PI/180* instead of */180*PI*)
- Create tables:

```
FOR I%=0 TO 359
```

```
    A(I%)=A(I%)*SIN(I%/180*PI)
NEXT I%
```

①

```
deg.rad=PI/180
for i%=1 to 359
    MUL a(i%),SIN(i%*deg. rad)
next i%
```

②

```
DIM sinus(360)
FOR i%=0 to 360
    sinus(i%)=sin(i%*PI/180)
NEXT i%
.....
FOR i%=0 to 359
    MUL a(i%),sinus(i%)
NEXT i%
```

The last version gains most by compiling — but it's fastest for the interpreter as well. In this routine it would not be advisable to replace the *FOR-NEXT* with a *REPEAT-UNTIL*, because the looping takes only a minimal part of the execution time, and the interpreted version would slow down greatly.

- Fill arrays with constants using *ARRAYFILL*
- Move one numeric field to another using this method:

```
BMOVE VARPTR(a(0)),VARPTR(b(0)),6*DIM?(a())
```

This is equivalent to:

```
FOR i%=0 TO DIM?(a())-1
    b(i%)=a(i%)
NEXT i%
```

but much faster.

Optimizing is often best learned by looking at other programs (public domain or from magazines). Many of these are not particularly good, but they can be useful nevertheless. By looking at a program, it is usually easy to determine how long the programmer has been using the computer language.

Take a program from a magazine and try to optimize it until you are completely satisfied with the performance.

Let the program rest for two weeks and then try to read it. Do you still understand what it is doing? Is it well documented? Did you flag the changes that were made? Does the program have a date? Are all the improvements you made worthwhile? Could further improvements be made?

Of course it's a matter of taste, how meaningful your variable names are — but long names have no effect on execution speed.

After practicing in this way, you will be able to determine quickly if a program in a magazine has been written well, or whether it was written in haste. When the program was written by many authors, you will often be able to tell which person wrote a particular section.

It is also important to limit yourself: If the program runs without errors and is fairly fast and does not use too much memory; then by all means please leave it alone. Making a program worse is very easy.

> *One more tip: If you have corrected a program, please save the old version on diskette.*

With *GFA BASIC* it is also important to save a version as a *LST*-file, since it happens that the *ST* computer will occasionally destroy a file. With a tokenized file it is almost impossible to repair the file. It may, however, be possible with a *LST*-file.

✧
✧ ✧

1.4 Sorting

It happens quite often that a field must be sorted. A rapid sort process is available with *QUICKSORT*, a recursive sort method that is often used to show the advantage of *PASCAL* or other similar languages. There are some *BASIC* versions of *QUICKSORT* available that simulate recursion, since normal *BASIC*s do not know what recursion is. When using *GFA BASIC* it is best to use the real recursive method.

```
' QSORT. BAS
'
DIM a$(1000)
t%=TIMER
FOR i%=0 TO 999
  a$(i%)=MKI$(XBIOS(17))+MKI$(XBIOS(17))+MKI$(XBIOS(17))
  a$(i%)=a$(i%)+MKI$(XBIOS(17))+MKI$(XBIOS(17))
NEXT i%
PRINT (TIMER-t%)/200
t%=TIMER
@quicksort(*a$(),0,999)
PRINT (TIMER-t%)/200
'
  PROCEDURE quicksort(str. arr%,l%,r%)
  LOCAL x$
  SWAP *str. arr%,a$()
  @quick(l%,r%)
```

```
        SWAP *str. arr%,a$()
    RETURN
    PROCEDURE quick(l%,r%)
            LOCAL ll%,rr%
        ll%=l%
        rr%=r%
        x$=a$((l%+r%)/2)
        REPEAT
            WHILE a$(l%)<x$
                INC l%
            WEND
            WHILE a$(r%)>x$
                DEC r%
            WEND
            IF l%<=r%
                SWAP a$(l%),a$(r%)
                INC l%
                DEC r%
            ENDIF
        UNTIL l%>r%
        IF ll%<r%
            @quick(ll%,r%)
        ENDIF
        IF l%<rr%
            @quick(l%,rr%)
        ENDIF
    RETURN
```

The *QUICKSORT* can be further improved: It takes a long time to sort if most of the fields are already in order. The biggest improvement is made by checking if the range from the left limit and the right limit exceeds a determined amount, and then sort those fields using a different method.

Example:

```
    Procedure quick(l%,r%)
        IF r%-l%=1
            IF a$(l%)>a$(r%)
                SWAP a$(l%),a$(r%)
```

```
        ENDIF
        GOTO qsortx
    ENDIF
    ' Insert the above procedure
    qsortx:
RETURN
```

This small change will improve the sort by about 4 percent when using the interpreter. In compiled programs this version is a few milliseconds slower, since in the compiler the recursion is greatly accelerated. This can change in future versions of the compiler or the interpreter. No program will absolutely be slower, only the relationship will change.

Further speed improvements can be made by setting the limit to 2 or 3 instead of 1.

1.5 Mini Data

The following program demonstrates how to search quickly through a set of data in a file which is not sorted:

```
' minidat
'
Max%=100                          ! number of data sets
'
Open "O",#1,"test.dat"
Dim Ind%(1000),Key$(1000)
I%=0
Repeat
    A$=""
    For L%=0 To 10+Random(20)
        A$=A$+Chr$(Random(26)+65)
    Next L%
    Ind%(I%)=Loc(#1)
    Key$(I%)=A$                   ! Key field
    Inc I%
    Print #1,A$
    Print #1,A$+A$                ! Data field
    Print #1,A$+A$+A$
Until I%>Max%
Close #1
'
```

```
@Sort
'
Open "i",#1,"test. dat"
Do
    Line Input "Search after (+/-)";A$
    If A$="+"
        Q%=Min(Q%+1,Max%)
    Else
        If A$="-"
            Q%=Max(Q%-1,0)
        Else
            V%=Max%/2
            S%=V%
            While S%>1
                Sub S%,S% Div 2
                If Key$(V%)>A$
                    V%=Max(V%-S%,0)
                Else
                    V%=Min(V%+S%,Max%)
                Endif
            Wend
            Q%=Max(V%-2,0)
            While Key$(Q%)<A$ And Q%<Max%
                Inc Q%
            Wend
        Endif
    Endif
    Print Q%
    Seek #1,Ind%(Q%)
    Line Input #1,A$
    Line Input #1,B$
    Line Input #1,C$
    Print A$
    Print B$
    Print C$
Loop
'
' Now insert the QUICKSORT program
'
' After every      : SWAP a$(l%),a$(r%)
```

```
' Insert          : SWAP ind%(l%),ind%(r%)
```

(The demo program on the diskette sorts directly on the key field *key$()*).

The data consists of random input that contains one key and two data fields.

For every record the program stores the key and (*LOC*) the corresponding *LOC-Pointer* in two arrays.

The key field *key$()* is then sorted and the pointers in the other array are arranged in the same order.

One can then search for the data by using the key field that is in memory and then locating the rest of the data by using the data pointer.

It is actually not necessary to sort the data if it is contained in memory, but it is still faster to search for the data by using a binary search.

This routine is not very elegant, but it fulfills its purpose.

Advice for building a real data manager:

- Keep the sort key the same length (By using *LSET* for example).

The data can be built up in the following manner:

XXX.DAT	: The complete data
XXX.IDX	: The keyfield along with the record pointer (using MKL$/CVL)

or:

XXX.DAT : The complete data set
XXX.IDX : Save only the record pointer by using BSAVE "XXX.IDX",VARPTR(ind%(0),max%*4 (this is the fastest way)

Hint: If it is possible to make the key field 4 (or 8,12,...) characters long, then you can save the key as an integer rather then as a string. This way you will save having to build the descriptors and you will also be able to save the keys with the BSAVE/BLOAD *(or* BPUT/BGET*).*

```
OPEN "O",#1,"XXX. IDX"
BPUT #1,VARPTR(idx%(0)),max%*4
BPUT #1,VARPTR(key0%(0)),max%+4
'BPUT #1,VARPTR(key1%(0)),max%*4
CLOSE #1
```

The security of the data is extremely important. In the above examples it is easy to reconstruct the key field in case the index file is distorted or lost.

You can also save disk space by using only *CHR$(10)* instead of the normal *CHR$(13)+CHR$(10)* combination as it happens when using the *PRINT* command. In the *MINIDAT* program just replace the line as follows: *PRINT #1,a$;chr$(10);*

This does not have any effect on the data other then saving disk space. The data input routine does not have to be altered. It will simply read the data slightly faster.

The problem with these methods of storing data is that if you add a new record, or the length of the data changes, then it must be added at the end of the file, the record index must be updated, and some parts of the file will contain garbage.

It is best to replace the record with the null character. In this case the records will automatically move toward the front of the file during a sort and can thus be easily removed.

You could also include the current length of the record as part of the data, and when the record changes, or new records are added, the program merely has to match the length with an already existing record previously deleted in the program.

Eventually you must run a routine that will remove all the dead space.

It is also possible to speed up the search process of multiple fields by creating key fields for more than one field.

The purpose of this chapter was to show you that there is not a given recipe to optimize a program. Often it is not possible to improve the program by optimizing the structure of the data.

CHAPTER 2

GRAPHICS

There are many graphic commands in *GFA BASIC* and most of them are fairly easy to use. For example, to draw a box all you need are the coordinates of two opposite corners.

BOX	= Draws a box
PBOX	= Draws a painted box
RBOX	= Draws a box with rounded corners
PRBOX	= Draws a painted and rounded box
CIRCLE	= Draws a circle

These simple graphic commands (called primitives in the *GEM-VDI* nomenclature) are easy to understand and simple to use. Before we move on to the more complicated graphic operations, let's look at a few "*forgotten*" graphic commands. Whenever you draw a filled rectangle using the *PBOX* command, it contains a border. There is a *VDI*-routine that will eliminate that border or perimeter. *PBOX, PCIRCLE, PELLIPSE* and *PRBOX* can all be drawn without a frame.

```
Procedure vsf_perimeter(flg!)
    DPOKE INIT, flg!
    DPOKE CONTRL+2,0
    DPOKE CONTRL+6,1
    VDISYS 104
RETURN
```

Setting the *flag!* to *true* turns the frame on and *false* turns the frame off. If you would like to call this routine by some other name, you may do so, but I tried to use the descriptions in the *GEM-VDI* literature.

The *FILL*-command in ***GFA BASIC*** calls the *v_contour*-routine. The area is filled from the starting point to the edge of the screen, or to a change of color. So the ***GFA BASIC*** *FILL*-command can be used with color monitors, a -1 must be chosen for edge color, in other words, the fill is terminated as soon as a pixel with a color other then the starting one is encountered.

```
Procedure v_contour(x%,y%,f%)
    DPOKE PTSIN,x%              ! coordinates just
    DPOKE PTSIN+2,y%            ! as the FILL
    DPOKE INTIN,f%              ! frame color!!
    DPOKE CONTRL+2,1
    DPOKE CONTR+6,1
    VDISYS 103
Return
```

This routine, for example, would allow you to fill everything on the color screen that was not enclosed by a green line or a line of any color *L%*. Since this is used very seldom, I did not want to modify the *FILL* command to accommodate this.

2.1 Setcolor

There is also a routine in *GFA BASIC* that allows you to change the color registers: *SETCOLOR n%, r%, g%, b%* or *SETCOLOR n%,&Hrgb.* To determine the color register use the following routine:

```
DEFFN getcolor(n%)=XBIOS(7,n%,-1) AND &H777
```

Unfortunately, the order between *COLOR* and *SETCOLOR* was totally mixed up by either *ATARI* or *DIGITAL RESEARCH*. The *VDI* also contains a *SETCOLOR*-Routine that works somewhat differently:

```
PROCEDURE v_setcolor(n%,r%,g%,b%)
    DPOKE CONTRL+6,4
    DPOKE INTIN,n%
    DPOKE INTIN+2. r%
    DPOKE INTIN+4,g%
    DPOKE INTIN+6,b%
    VDISYS 14
RETURN
```

The colors red (*r%*), green (*g%*) and blue (*b%*) must be set between 0 and 1000. You may also inquire as to the current color as follows:

```
PROCEDURE v_getcolor(n%)
```

```
    DPOKE CONTRL+6,2
    DPOKE INTIN,n%
    DPOKE INTIN+2,0
    VDISYS 26
RETURN
```

The result can be found as:

```
n%:= DPEEK(INTOUT),r%:= DPEEK(INTOUT+2) etc.
```

2.2 *Clipping*

Whenever you open a window, these graphic commands work slightly different. For example, the null point moves from the top left corner of the screen to the top left corner of the window. Other functions like lines, circles, etc. are truncated at the window's border.

Moving the origin:

```
PROCEDURE origin(x%,y%)
    DPOKE WINDTAB+64,x%
    DPOKE WINDTAB+66,y%
RETURN
```

To truncate lines at the borders of a rectangular area:

```
PROCEDURE vs_clip(xl%,yl%,x2%,y2%)
    DPOKE PTSIN,x1%
    DPOKE PTSIN+2,yl%
    DPOKE PTSIN+4,x2%
    DPOKE PTSIN+6,y2%
    DPOKE INTIN,1
    DPOKE CONTRL+2,2
    DPOKE CONTRL+6,1
    VDISYS 129
RETURN
```

Example: If using *@vs_clip(100,120,200,180),* confines the graphic output to a section of the rectangle at (100,120) to (200,180); then using *@origin(100,120),* sets the origin point for graphic input to the top left corner of this rectangle.

The *ORIGIN* and *CLIPPING* commands are only valid for normal graphics commands, *not* for the *PUT, GET, BITBLT*, or any *AES* commands. Nor can they be used with almost anything that contains *PTSIN+...* or *PTSOUT+..., GINTIN+...* or *GINTOUT+..., MENU* commands, or coordinates contained within object trees (see Chapter 6 on *RSC*-files).

CLIPPING can be completely turned off if desired. This speeds up the drawing commands by about two percent, It can, however, cause even a simple *PLOT* command to change memory locations outside the screen buffer thus causing bombs or other malfunctions within the computer: ***use caution!***

```
DPOKE INTIN,0
POKE CONTRL+2,2
DPOKE CONTRL+6,1
VDISYS 129
```

CLIPPING can be restored by calling the *vs_clip* routine. Unfortunately, moving the origins does not cause the mouse input (*MOUSEX, MOUSEY, MOUSE* ...) to return a negative number whenever the mouse is above or to the left of the origin. Instead, 65536 is added to those negative coordinates. For those who would rather have a more meaningful value returned the *@ext(MOUSEX)* or *@ext(DPEEK(X)* routine may be used:

```
DEFFN ext(x%)=x%+65536*(x%>32767)
```

That covers the simple graphics command. Let us move on to the somewhat more difficult Raster Graphic Commands.

2.3 Raster Graphic Commands

If you take a close look at the picture on your monitor you will determine that it is composed of many small dots that are organized by rows and columns. Therefore, a vertical line will always appear ragged (viewing at low resolution will make this more obvious than at high resolution). Every pixel on the screen represents one bit of memory on a Mono *ST* computer or two or four bits on the Color monitor. There are many commands in *GFA BASIC* that allow you to manipulate these pixel-blocks.

GET, PUT and BITBLT

The *GET* command allows you to copy a screen segment to a string and the *PUT* command allows you to restore that segment to the screen. The *BITBLT* command does the same thing, but is somewhat more flexible — *and* more prone to errors.

A string built by using the *GET/PUT* command is composed as follows:

```
svar$ = mki$(Width)
      + mki$(Height)
      + mki$(Bitcount)
      + Bitpattern
```

Where the *width* is the difference between both *X* coordinates of the box.

The *height* is the difference between the *Y* coordinates.

Bitcount is the number of bits it takes for each pixel (1,2 or 4 as explained above).

And *Bitpattern* is the actual graphic information.

The construction of that bit pattern is very complex. Each word consists of 16 bits (This shows that the *ST* is a 16-bit computer). A mono computer contains 16 pixels per word. A color computer is made up of either two or four adjacent words. This is repeated until one line is filled. The remaining part of the word will contain junk (some random data). This line and all following lines are always represented by an even number of bytes (8 bits).

Memory Usage

You will have noticed that this procedure creates a lot of overhead. Which explains why it often takes a lot of memory to store a bit pattern. Now take a look at a mono screen:

```
GET 0,0,00,79,a$
GET 0,0,01,79,b$
GET 0,0,15,79,c$
GET 0,0,16,79,d$
```

Where *a$* contains the 6 byte prefix and 80 rows of words (+2*80+6=166 Bytes). Every row contains only one bit (80/8=10 Bytes). This shows that 90% of *a$* is just ballast.

The *b$* is just as long, but two bits are used instead of one.

The *c$* is also as long as *a$*, but it is put to optimal use.

And *d$* contains an extra word which increases the length to 326 Bytes (6+2*80+2*80=326).

With color the string Lengths are 326 and 646 bytes (medium resolution) or 646 and 1286 bytes (high resolution) instead of the 166 and 326 on the mono screen.

2.4 Graphics Mode

GFA BASIC has a Graphmode command that allows you to select the drawing mode for graphics operations.

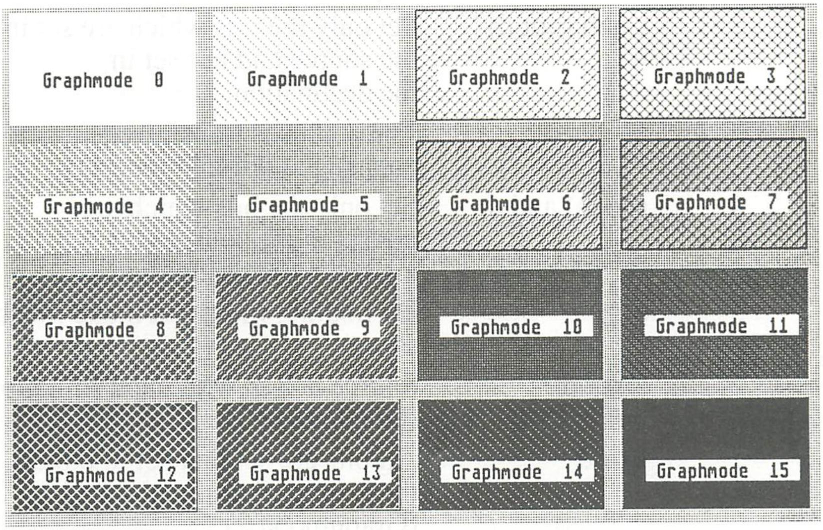
Graphmode 1 = Replace
This is the normal mode which replaces the old picture with a new one.

Graphmode 2 = Transparent
The old picture can still be seen behind the new transparent one.

Graphmode 3 = Xor
This mode intermixes the new picture with the old one (an off pixel turns on and an on pixel turns off). This mode allows you to create blinking screen segments, like the rubberband in drawing programs.

Graphmode 4 = Inverse Transparent
This mode is similar to mode 2 except the new picture is shown in inverse.

Figure 1: Graphmode Settings



This *Graphmode-setting* does not apply to the *PUT* and *BITBLT* commands. There are sixteen *Graphmodes* that can be passed on with these commands. If no mode is selected with the *PUT* command, mode 3 is automatically chosen (this is the same as *Graphmode 1* or the other commands).

The following shows "*s*" as a bit for the source raster and "*d*" for the destination raster.

Nr.	Result	
0	0	All bits are cleared.
1	s and d	Only those bits contained in both stay set.
2	s and (not d)	Set only the bits which are set in the source and not set in the destination.
3	s	The source is transferred unchanged (Graphmode 1).
4	(not s) and d	Set only the bits which are not set in the source and are set in the destination.
5	d	Do nothing (does not make much sense).
6	s xor d	Source is Xor with the destination (Graphmode 3).
7	s or d	All bits are set that set in either the source or destination (Graphmode 2).
8	not (s or d)	Set all bits that are not set in the source or destination.
9	not(s xor d)	Set all bits that set in both rasters or that are not set in both rasters.
10	not d	The destination raster is inverted.
11	s or (not d)	Set all bits that are set in the source and that are not set in the destination.
12	not s	The source raster is inverted before it is transferred.
13	(not s) or d	Graphmode 4.
14	not (s and d)	Set all bits not set in either the destination or source.
15	1	All bits are set.

The important modes are:

3	=replace
4	=xor
7	=transparent
13	=inverse transparent

2.5 Graphics on Diskette

It is quite easy to save a graphic picture to diskette by using the *BSAVE* command. To load it just use the *BLOAD* command. A problem arises when many small pictures are to be saved to the diskette since 100 pictures would require 100 files on the diskette (they each contain at least 1 Kbytes plus 32 bytes for the directory). The following is a program that will save and load 100 GET/PUT segments from the diskette as one file.

To save:

```
OPEN "O",#1,#1,"file.get"
FOR i%=0 TO 99
    PRINT #1,MKI$(LEN(a$(i%)));a$(i%);
CLOSE #1
```

To load:

```
OPEN "I",#1,"file. get"
FOR i%=0 TO 99
a$(i%)=INPUT$(CVI(INPUT$(2,#1)),#1)
NEXT i%
CLOSE #1
```

Explanation: This *GET/PUT* string (*a$*) can contain any character (comma, backspace, line feed, etc.) which makes it impossible to use the *INPUT* command. Instead, the *INPUT$*-function is used. The length is stored in two bytes by using *MKI$*. When reading the data the length is extracted using *CVI(INPUT$(2,#1))*. This length is then used in the outer *INPUT$* function.

■ *BITBLT*

This command performs roughly the same function as the *GET/PUT* combination. This command is somewhat more flexible, but it is also harder to use.

```
BITBLT smfdb%(),dmfdb%(),p%()
```

These parameters already hint that this command is very powerful. The *smfdb* stands for Source Memory Form Description Block, which describes the form and *dmfdb* describes the form for the destination. The *p* stands for point and contains the coordinates for the source and destination rectangles and also the mode of how they overlap (see *PUT* above).

_mfdb%(0) contains the raster address. Usually, at least one of either *smfdb%(0)* or *dmfdb%(0)* equals the screen address *(XBIOS(2))*. This address must be even.

_mfdb%(1) contains the width of the raster in pixels units (640 for mono, 320 or 640 for color). Other numbers divisible by 16 could also be used.

_mfdb%(2) contains the raster height (400 or 200 or ...).

_mfdb%(3) contains the raster width in words. This is always the pixel count divided by 16.

_mfdb%(4) this is always zero in *Atari GEM* since an independent format was not yet implemented.

_mfdb%(5) contains the number of bit planes (mono 1, color 2 or 4).

_mfdb%(6) to mfdb%(8) are reserved for future additions (very unlikely).

If _mfdb%(0)=0, *GEM* will create the rest of the *MFDB*'s by itself pointing to the current screen.

Example (Hires!Midres/Lores):

```
GET 100,110,120,130,a$

a$=MKI$(20)+MKI$(20)+MKI$(1)+SPACE$(84)      !168/336
smfdb%(0)=XBIOS(3)
smfdb%(1)=640                          ! 640/320
smfdb%(2)=400                          ! 200/200
smfdb%(3)=40                           ! 40/ 20
smfdb%(5)=1                            ! 2/ 4
dmfdb%(0)=VARPTR(a$)+6
dmfdb%(1)=32                           ! (Width+16) and &FFF0
dmfdb%(2)=21                           ! Height
dmfdb%(3)=2                            ! dmfdb%(1)/16
dmfdb%(5)=1                            ! 2/ 4
p%(0)=100
p%(1)=110
p%(2)=120
p%(3)=130
p%(4)=0                                ! always the left top corner
p%(5)=0
p%(6)=20
p%(7)=20
p%(8)=3                                ! copy mode
BITBLT smfdb%(),dmfdb%(),p%()
```

Both strings are identical as far as the relevant bits are concerned. The *GET*-command leaves the input string unchanged if the bits are not inside the rectangle. Adding a *a$=SPACE$(90)* (*!174/342*) before the *GET*-command will result in identical strings. Let us continue with a more demanding example, as might be used in a graphics program, a routine that mirrors a rectangle across the vertical or horizontal axis that was read with the *GET* command. *GET/PUT* is slow and if you are not careful you could easily use up more than half of a megabyte of memory just for a mirror effect.

```
Dim Smfdb%(8),Dmfdb%(8),P%(8)
For I%=0 To 639 Step 8
    Line I%,0,639-I%,399
Next I%
Get 0,0,639,399,A$                    ! change if color
T%=Timer
@Mirrorput(0,0,*A$)
Print Timer-T%
Procedure Mirrorput(X%,Y%,S.%)
    If Dpeek(S.%+4)>6                 ! only if something is there
        A%=Lpeek(S.%)
        B%=Dpeek(A%)                  ! width
        H%=Dpeek(A%+2)                ! height
        Smfdb%(0)=A%+6
        Smfdb%(1)=(B%+16) And &HFFF0
        Smfdb%(2)=H%+1
        Smfdb%(3)=Smfdb%(1)/16
        Smfdb%(5)=Dpeek(A%+4)
        Dmfdb%(0)=XBIOS(3)
        On XBIOS(4)+1 Gosub
            Mfdb.lores,Mfdb.midres,Mfdb.hires
        P%(1)=0                               !***
        P%(3)=H%                              !***
        P%(5)=Y%                              !***
        P%(7)=Y%+H%                           !***
        P%(8)=3                               !***
        P%(4)=X%+B%                           !***
        P%(6)=X%+B%                           !***
        For I%=0 To B%                        !***
            P%(0)=I%                          !***
            P%(2)=I%                          !***
            Bitblt Smfdb%(),Dmfdb%(),P%()     !***
            Dec P%(4)                         !***
            Dec P%(6)                         !***
```

```
        Next I%                              !***
      Endif
    Return
    Procedure Mfdb.hires
      Dmfdb%(1)=640
      Dmfdb%(2)=400
      Dmfdb%(3)=40
      Dmfdb%(5)=1
    Return
    Procedure Mfdb.midres
      Dmfdb%(1)=640
      Dmfdb%(2)=200
      Dmfdb%(3)=40
      Dmfdb%(5)=2
    Return
    Procedure Mfdb.lores
      Dmfdb%(1)=320
      Dmfdb%(2)=200
      Dmfdb%(3)=20
      Dmfdb%(5)=4
    Return
```

The program lines marked with !*** must be replaced with the following to mirror across the horizontal axis:

```
P%(0)=0
P%(2)=B%
P%(4)=X%
P%(6)=X%+B%
P%(8)=Modus%
P%(5)=Y%+H%
P%(7)=Y%+H%
For I%=0 To B%
  P%(1)=I%
  P%(3)=I%
  Bitblt Smfdb%(),Dmfdb%(),P%()
  Dec P%(5)
  Dec P%(7)
Next I%
```

At the start of the program, a simple pattern is drawn to the screen and this pattern is then copied to a string (*a$*) with the *GET* command. Procedure *@mirrorput* takes this pattern and mirrors it across the vertical axis. The parameters are similar to the ones needed with a *PUT* command: *X* coordinates, *Y* coordinates, String and Mode. The string is not passed by value, but rather by the pointer using the asterisk symbol.

This results in the string not having to be passed to the corresponding local variable (this saves time). The address (=*Varptr*) is determined with *LPEEK(*a$)* and the length with *DPEEK(*a$+4)*. The procedure *mirrorput* checks to see if the string is longer than 6 characters (it must be longer than 6 for the *GET* command). Next, the starting address, width and height of the *GET*-string are determined and the *mfdbs* (*Memory Form Description Blocks*) are created. Notice that the width and the height must be incremented by one.

The *XBIOS(4)* routine is called to determine the current screen resolution so that the correct *dmfdb* procedure can be called. Next, a loop is executed that increments or decreases the X-coordinates of the source rectangle and destination rectangle so that the mirror effect is created.

The following is a demonstration program that allows you to move a picture segment by using the corresponding mouse coordinates. This allows you to test the speed gain that might be achieved with the blitter chip whenever it becomes available.

```
Dim Smfdb%(8),Dmfdb%(8),P%(8)
Graphmode 3
For I%=0 To 639 Step 8
Line I%,0,639-I%,399
Next I%
For I%=0 TO 399
    LINE 639,I%,0,399-I%
```

```
Next I%
Dmfdb%(0)=XBIOS(3)
On XBIOS(4)+1 Gosub Mfdb.lores,Mfdb.midres,Mfdb.hires
Repeat
    Mouse X%,Y%,K%
    If X%<>0 And X%<>639 And Y%<>0 And Y%<>399
        P%(0)=X%
        P%(1)=Y%
        P%(2)=639-X%
        P%(3)=399-Y%
        Q%=Even(X%+Y%)-Odd(X%+Y%)
        P%(4)=P%(0)+Q%
        P%(5)=P%(1)+Q%
        P%(6)=P%(2)+Q%
        P%(7)=P%(3)+Q%
        P%(8)=3
        Bitblt Smfdb%(),Dmfdb%(),P%()
    Endif
Until K% And 2
'Now add the mfdb. xxxx routines from above.
```

Caution: It is extremely important that the coordinates of the destination rectangle reside within the picture. There is no safety check in the VDI routine. With color, the coordinates (639 and 399) must be adjusted. If the source and destination rectangles overlap, the destination is never changed, before the corresponding part of the screen is used as source. A similar effect is done inside the BMOVE routine.

If the size of the source and destination rectangle are different, the connection is made with the size of the source rectangle. Nevertheless, both corner points of the rectangle must always be supplied.

2.6 Flicker Free Graphics

When moving Bit blocks (with *BITBLT* or *GET/PUT*), the picture on the screen may flicker. To eliminate this flickering, you would want to display a picture on screen and then build a new picture off screen in memory and display it when the first is done.

The *ST* contains a *XBIOS* routine that helps with this process called *setscreen*. This routine allows you to switch between the physical (as displayed) and the logical (as being built) screen address. It is important that the screen address is divisible by 256.

```
Dim Screen%(32255/4)
Graphmode 3
For I%=0 to 639 Step 4
    Line 0,0 I%,399
    Line 639,0,I%,399
Next I%
Get 0,0,99,99,A$
A%=XBIOS(3)
B%=(Varptr(screen%(0)+255) And &HFFFFF00
Sget H$
Repeat
    Swap A%,B%
    Void XBIOS(5,L:A%,L:B%,-1)
```

```
        Sput H$
        Mouse X%,Y%,K%
        Put X%,Y%,A$
        If K%=1
            Sget H$
        Endif
        Line X%,0,X%,399
        Line 0,Y%,639,Y%
    Until K%=2
    A%=Max(A%,B%)
    Void XBIOS(5,L:A%,L:A%,-1)
    Sput H$
```

A second screen is stored in an integer field. The size of this field is 32000 Bytes (screensize) plus 255 bytes to make sure the screen address resides within a 256-byte boundary. This number 32255 is then divided by 4, the size of the integer number. The screen address is the first address within this field that is divisible by 256 (this is accomplished with *AND &HFFFFF00*). The screen background is saved into string *H$* (also with a simple pattern).

```
    XBIOS(5,L:A%,L:B%,-1)          ! setscreen
```

This *XBIOS* call sets the logical screen address to the value in variable *A%* and the physical screen address to the value in variable *B%*.

One of the variables contains the old screen address (*XBIOS(3)*) and the other variable contains the second screen address. The two screen addresses are then used to set the logical and physical screen base to different values. The *swap* makes sure that next time the *XBIOS(5...)* is called, one image pops onto the screen while the other vanishes and waits to be replaced by a new one.

The *SPUT* command restores the background of the logical screen. After the mouse input, the screen segment is drawn with the *PUT* command. If the mouse button is pressed, the current picture is copied into the background string. A cross is then drawn which disappears the next time the *SPUT* command is issued. This process is repeated until the right mouse button is pressed. Finally both screen addresses are set to the original value (this is easy with the *max()* function). You can also reserve more than two screen sections (16 of them will occupy 1/2 megabytes).

It is also possible to vertically scroll through more than one picture by changing the screen address in smaller steps. The screen address can only be changed in steps of 256 bytes. Since a screen line contains 80 bytes (160 bytes with color), the scrolling is only possible in steps of 16 lines (the smallest common denominator of 256 and 80 is 1280, 16*80, or 2560, 16*160 for color). This small program puts three overlapping pictures into an integer field and then scrolls them.

```
'Scroll Demo
'===========

Dim A%((32000*3+255)/4)
A%=Varptr(A%(0))+255 And &HFFFFF00
Graphmode 3
For I%=0 To 639
    Line 0,0,I%,399
Next I%
Bmove XBIOS(3),A%,32000
Cls
For I%=0 To 639
    Line 0,399,I%,0
Next I%
Bmove XBIOS(3),A%+32000,32000
Bmove A%,A%+64000,32000
Repeat
    For B%=A% To A%+64000-1280 Step 1280
        Void XBIOS(5,L:-1,L:B%,-1)
        Vsync
    Next B%
Until Mousek
```

```
Void XBIOS(5,L:-1,L:XBIOS(3),-1)
```

The last parameter in *XBIOS(5) (-1)* indicates that the resolution should not be changed. Unfortunately, an *orderly* change in the resolution is not possible with *GEM*. In any case, 0 is for low, 1 for medium, and 2 for high resolution. Parameters can be set to *-1*, which tells the operating system not to change the corresponding value.

You can, of course, change not only the address, but also find out the current values.

```
Physical_address=XBIOS(2)          ! physbase
Logical_address=XBIOS(3)           ! logbase
Resolution=XBIOS(4)                ! getrez
```

If you now think that *BITBLT* is only a pure graphic command, I must disappoint you. It is possible to use the *BITBLT* command in a bit pattern that is contained in memory as long as it can be interpreted as a raster. For example, you could create such a raster in an integer field and then set all field elements to null (*ARRAYFILL*) or change only certain bits within each field. Possible uses are left to your imagination.

For fans of nice looking character sets, here is a tip: The bit pattern of fonts may be moved relatively easily with the *BITBLT* command as is often needed with proportional spacing.

CHAPTER 3

TIPS & PROGRAMS

As the title of this chapter indicates, the following pages will try to show a variety of different concepts. Some pages contain programs that could be put to immediate use.

Examples of these are the *Input-routine* (Chapter 3.1) or the *FONTDEMO* at the end of this chapter (Chapter 3.12). Some of the concepts in these examples may not be clear to you until after you have studied Chapters 4 through 6.

There are many other things discussed in this chapter, like *SCAN*-Codes (Chapter 3.5) and *Recursion* (Chapter 3.10).

Since most of the concepts in this Chapter are not related to each other as in previous chapters every sub chapter will start on a new page.

3.1 Dialog Boxes Homemade

It is possible to create Dialog boxes using a *Resource Construction Set* and then manipulate these boxes using the corresponding *GEM* calls. But it is also possible to write your own *Input-Routine* using ***GFA BASIC*** and thereby gain a lot more control over your input.

The following program contains an *Input-Routine* somewhat similar to the one used with *GEM.*

```
' Input.bas
'
Dim X%(10),Y%(10),T$(10),L%(10),I$(10),V%(10)
For I%=0 To 6
    Read X%(I%),Y%(I%),T$(I%),L%(I%),V%(I%)
    I$(I%)=""
Next I%
Data 0,0,"Last Name :",20,0
Data 0,1,"First Name :",20,0
Data 0,2,"Street :",20,0
Data 0,3,"City:",16,0
Data 23,3,"State :",2,0
Data 0,4,"Zip Code:",5,1
Data 16,4,"Tel.  :",20,2
Do
    @Input_routine(6,100,100,1)
    For I%=0 To 6
```

```
        Print T$(I%)'I$(I%)
    Next I%
    Print
Loop
Procedure Input_routine(N%,X%,Y%,F%)
    Vdisys 38                        ! gets character size
    Cb%=Dpeek(Ptsout+4)              ! in pixels and character
    Ch%=Dpeek(Ptsout+6)              ! spacing
    Lh%=Dpeek(Ptsout+2)
    Ll%=Ch%-Lh%
    Insflg!=True
    Spec$=Chr$(8)+Chr$(13)+Chr$(27)
    Sp.scan$=Chr$(&H48)+Chr$(&H4B)+Chr$(&H50)+Chr$(
&H4D)+Chr$(&H52)
    Spr$=Mki$(0)+Mki$(Lh%)+Mki$(-1)+Mki$(1)+Mki$(0)
    For I%=1 To Ch%
        Spr$=Spr$+Mkl$(&H8000)
    Next I%
    Spr$=Left$(Spr$+String$(74,0),74)
    U$=String$(100,"_")
    Dim Tx%(N%),Ty%(N%)
    Mx%=0
    My%=0
    For I%=0 To N%
        Tx%(I%)=X%+Cb%*(X%(I%)+Len(T$(I%)))
        Ty%(I%)=Y%+Ch%*Y%(I%)
        Mx%=Max(Mx%,Tx%(I%)+Cb%*L%(I%))
        My%=Max(My%,Ty%(I%))
    Next I%
    If F%
        Get X%-10,Y%-10-Ch%,Mx%+10,My%+10,Temp$
    Endif
    Deffill 1,0
    Color 1
    Pbox X%-10,Y%-10-Ch%,Mx%+10,My%+10
    Box X%-5,Y%-5-Ch%,Mx%+5,My%+5
    ' or deffill ,2,1 pbox . . .
    For I%=0 To N%
        T$=T$(I%)+Left$(I$(I%)+U$,L%(I%))
        Text X%+X%(I%)*Cb%,Y%+Y%(I%)*Ch%,T$
```

```
        Next I%
        E%=0
        T$=I$(E%)
        C%=0
        Repeat
            @E.curson
            Repeat
                Mouse Mox%,Moy%,Mok%
                K$=Inkey$
            Until Len(K$) Or Mok%
            @E. cursoff
            If K$<>""
                If Len(K$)=1
                    @E.do_char(Asc(K$))
                Else
                    @E.do_scan(Asc(Right$(K$)))
                Endif
            Endif
            If Mok%
                If Mox%>=X% And Moy%>=Y%-Ch%
                    If Mox%<=Mx% And Moy%<My%
                        @E.do_mouse
                    Endif
                Endif
            Endif
        Until E%>N% Or E%<0
        If F%
            Put X%-10,Y%-10-Ch%,Temp$
        Endif
        Erase Tx%()
        Erase Ty%()
    Return
    Procedure E.dsp.In
        If Len(T$)>L%(E%)
            Out 2,7
            T$=Left$(T$,L%(E%))
        Endif
        C%=Min(C%,Len(T$),L%(E%))
        Text Tx%,Ty%,Left$(T$+U$,L%(E%))
    Return
```

```
Procedure E.curson
    Tx%=Tx%(E%)
    Ty%=Ty%(E%)
    Sprite Spr$,Tx%+C%*Cb%,Ty%
Return
Procedure E. cursoff
    Sprite Spr$
Return
Procedure E. ins_char(K%)
    Do
        Exit If V%(E%)=1 AndInstr("0123456789",
Chr$(K%))=0
        Exit If V%(E%)=2 And Instr("0123456789/()-",Chr$
    (K%))=0
        ' here you can easily add your own types
        If Insflg! Or C%=Len(T$)
            T$=Left$(T$,C%)+Chr$(K%)+Mid$(T$,C%+1)
        Else
            Mid$(T$,C%)=Chr$(K%)
        Endif
        Inc C%
        @E.dsp.ln
        Goto E.insx
    Loop
    Out 2,7
    E.insx:
Return
Procedure E.do_char(K%)
    V%=Instr(Spec$,Chr$(K%))
    If V%
        On V% Gosub E.backs,E.enter,E.esc
    Else
        @E.ins_char(K%)
    Endif
Return
Procedure E.backs
    If C%>0
        T$=Left$(T$,C%-1)+Mid$(T$,C%+1)
        Dec C%
        @E.dsp.ln
```

```
    Endif
Return
Procedure E.enter
    I$(E%)=T$
    Inc E%
    T$=I$(E%)
    C%=0
Return
Procedure E.esc
    T$=""
    C%=0
    @E.dsp.ln
Return
Procedure E. do_scan(K%)
    V%=Instr(Sp.scan$,Chr$(K%))
    If V%
        On V% Gosub E.up,E.lft,E.dwn,E.rgt,E.insert
    Else                                        ! see text
    Endif
Return
Procedure E.up
    I$(E%)=T$
    If E%
        Dec E%
    Else
        E%=N%
    Endif
    T$=I$(E%)
    C%=Len(T$)
Return
Procedure E.dwn
    I$(E%)=T$
    If E%<N%
        Inc E%
    Else
        E%=0
    Endif
    T$=I$(E%)
    C%=Len(T$)
Return
```

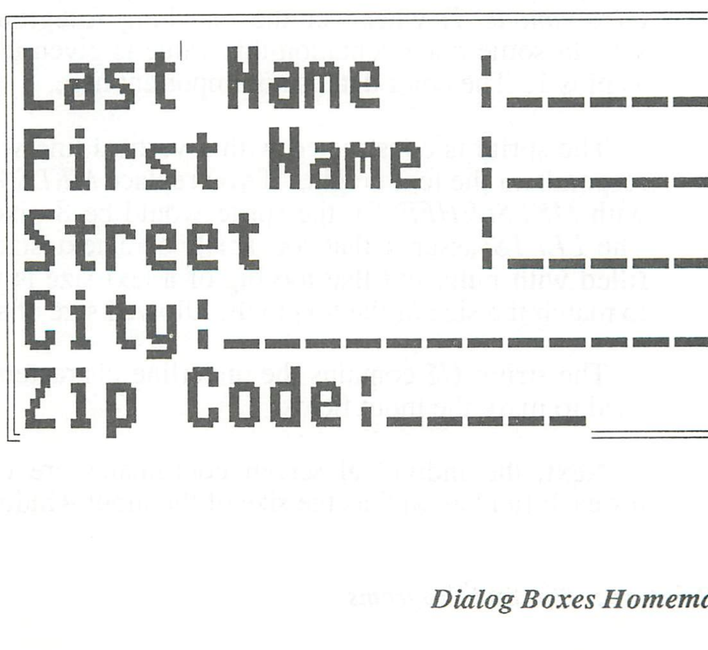
```
Procedure E.lft
    If C%
        Dec C%
    Endif
Return
Procedure E.rgt
    If C%<Len(T$)
        Inc C%
    Endif
Return
Procedure E.insert
    Insflg!=Not Insflg!
Return
Procedure E.do_mouse
    Qx%=(Mox%-X%)/Cb%
    Qy%=(Moy%-Y%)/Ch%+1
    I%=0
    Repeat
        If Qy%=Y%(I%)
            Q%=Qx%-X%(I%)-Len(T$(I%))
            If (Q% And 255)<=L%(I%)
                Goto E.dom.ok
            Endif
        Endif
        Inc I%
    Until I%>N%
    If 0
        E.dom.ok:
        I$(E%)=T$
        E%=I%
        T$=I$(E%)
        C%=Min(Q%,Len(T$))
        @E.curson
        @E.dsp.ln
    Endif
Return
```

First, all of the global arrays are dimensioned.

The Data arrays describe the input:

> The first two numbers determine the row and column position at which the input field should start (0,0 is top left corner and 0,1 is the line below it).
>
> Next the actual text that describes the field is given.
>
> The next number indicates the maximum number of digits or characters that this field may contain.
>
> The last number decides what kind of input is legal. The following kinds are legal for this program:
>
> 0 - All characters are allowed.
> 1 - Only numeric digits (0-9) are allowed.
> 2 - Only a number, a slash, a parenthesis, or a minus sign is allowed, as would be used in a telephone number.

Figure 2: Input Screen



After the data is read into the corresponding fields, the number of fields (*N%*), screen position (*X%,Y%*) and a flag (*F%*) are passed to the *Input-Routine*. The flag determines whether the corresponding segment should be saved with a *GET/PUT* command. The *Input-Routine* determines the size of the text. *VDISYS* 38 returns in *PTSOUT* the width and the height of the text and also the width and height of the box that surrounds the box. This procedure is used to allow the text to be displayed in any resolution by calculating the pixel oriented screen coordinates. This size is also used to determine the size of the vertical line which is used as the cursor.

The string *Spec$* contains the *ASCII* value for special keys, (*Backspace*, *Return* and *Esc*), which are easily distinguished from the other keys by using the *INSTR* command. The string *Sp.scan$* contains the *SCAN*-codes for special keys, here the codes for the *Arrows* and the *Insert* key are used.

String *Spr$* is a sprite, which will serve as the cursor. The vertical offset *MKI$(lh%)* serves to quickly position the cursor to the correct vertical line. The *Format-flag MKI$(-1)* makes sure that the sprite is inverted (*Graphmode 3*) whenever the matching foreground-bit is set. In some documentation this value is given incorrectly as plus 1. The color data is not important here.

The sprite is constructed with a vertical line which corresponds to the text height. If you replace *MKL$(&H8000)* with *MKL$(&HFF00),* the sprite would be 8 pixels wide. The *LEFT$* assures that too small of a text size will be filled with nulls and that too big of a text size is truncated to match the size of the text to the allowed size of sprites.

The string *U$* contains the underline characters that are used to mark the input fields.

Next, the individual screen coordinates are calculated for each field as well as the size of the input window.

If the flag *f%* is set, the picture is saved in string *Temp$*.

The *PBOX* command erases the window. The *BOX* command draws the double border around the text.

Next, all the field names, as well as the underline characters or the already existing input data contained in string *I$*, are put into string *T$*. This string is then used in the *TEXT* command to display the fields.

After the initialization of a few variables (*e%*=number of the field that contains the cursor, *T$*=the actual field contents and *C%*=the relative cursor position), the program continues with the main loop.

Next, the cursor is made visible and the program will loop until either a key is struck or the mouse button is pressed.

Next, the cursor is turned off.

If a key has been pressed, a routine to handle the *ASCII* character (*do_char*) or a routine for a *SCAN*-Code (*do_scan*) is called. Whenever a mouse button is pressed that resides within the range of the input, the cursor position is changed to that new position. If the cursor has not reached the end (pressing Return in the last field) and there was no error (*e%=-1*), the main loop is continued.

Otherwise, the picture is restored and the fields that contained the pixel coordinates,(*tx%* and *ty%*) are erased.

Routine *E.dsp.ln* displays the input field. If the maximum length is exceeded, the bell will sound and the string is truncated. The cursor cannot move past the end of the field.

Routine *E.curson* places the cursor position in variables *Tx%* and *Ty%* and then turns the sprite cursor on.

Routine *E.cursoff* turns the sprite off.

Routine *E.ins_char* inserts a character from the cursor position into string *T$* and then displays that line.

This construction with the *DO-LOOP* and the *EXIT* command is one possible means of checking for legal characters. You could have used nested *IF* statements, but the *DO-LOOP-EXIT* combination is easier to expand upon. A *GOTO* command was used to ring the bell only in case of an error.

Routine *E.do_char* calls special routines to handle the *Backspace*, the *Return/Enter* and the *Esc* key. All other characters are passed to the field by means of the *E.ins_char* routine.

Routine *E.do.scan* calls on special routines for *arrow* keys and for the *Insert* key (to change between add and insert mode).

You can also substitute the following for the *ELSE* branch:

@e.inschar(k% XOR 128)

something similar is performed by the ***GFA BASIC*** editor.

If you wish to use function keys use the following routine:

```
Procedure E.f1
  If Not Inp?(2)
      @E.dostring(Chr$(27)+"Werner"
    Endif
Return
Procedure E.dostring(A$)
```

```
    For li%=1 To Len(A$)
      @E.do_char(Asc(Mid$(A$,li%)))
    Next li%
Return
```

INP?(2) checks to see if another key was pressed before it executes the *F1*-Routine.

If you replace the *E.do_char* with *E.ins_char*, the routine would be faster but you would not be able to use *Control* characters, like the *Esc* key .

It would be even faster if you would write a routine that would directly manipulate the String *T$,* similar to the *E.ins_char*. This would also eliminate the need for the *INP?(2)*-function.

The function keys could be used in data files to read the next or previous record.

The *UNDO* key could also have some useful function.

The *E.do_mouse-Routine* changes the mouse coordinates to line coordinates. A check is made to determine if the mouse points to one of the input fields. It then changes the cursor position.

■ Remarks:

The program contains some sloppy code. I decided to leave it in the program to show that even the *GOTO* statement can have a useful purpose.

The *GOTO* in the *E.ins_char* routine could easily be replaced with a couple of nested *IF* statements.

The *GOTO* in the *E.do_mouse* routine could be removed by simply moving the program part between the *If* 0 and the *Endif* to where the *GOTO* is.

The *E.curson* routine was called in the *Mouse*-routine to set the cursor to *Tx%* and *Ty%*.

Also, not all the variables in the *Input*-routine were declared as local, even though this would not have been hard to do. To keep a program short, you can sometimes skip declaring the variables as local as long as you use some discipline to name your variables.

Example:

- Variables that contain only one character or start with the letters T or Q may be used for subroutines.
- All global variables must be a least four characters long.
- All variables that return an error code should start with the letter E.
- etc.

Make your own rules and obey them!

3.2 *Sound*

To find the parameters necessary to create a certain noise with the *SOUND* and *WAVE* command, it is usually best just to experiment. The following program will help to experiment with sound, and also show you how to use the mouse without using *AES* (*resource* file).

```
' SoundExp
' Sound Experiment program
'
@Draw_box(0)                        ! for sound. per
Xa%=-99
@Draw_box(50)                       ! for wave.per
Xb%=-99
@Draw_box(100)                      ! for noise
Xd%=-99
@Draw_box(160)                      ! for envelope curve
Xc%=-99
For I%=0 To 7
    Text 29+I%*35,176,I%+8
Next I%
Do
    Repeat
        Mouse X%,Y%,K%
    Until K%
    If Y%>0 And Y%<19                ! A Quick check to see
        @Sound.per                   ! if one of the rectangles
```

```
        Endif                                   ! was selected
        If Y%>50 And Y%<69
            @Wave.per
        Endif
        If Y%>100 And Y%<119
            @Noise
        Endif
        If Y%>160 And Y%<179
            @Wave.form
        Endif
    Loop
    Procedure Sound.per                                 ! Selecting the
        If X%>300 And X%<320                            ! frequency
            Per%=Min(Per%+1,4095)
        Else
            If X%>0 And X%<20
                Per%=Max(Per%-1,0)
            Else
                If X%>20 And X%<300
                    Per%=(X%-20)/280*4096
                Endif
            Endif
        Endif                                           ! Tone Changes
        Sound 1,8,#Per%
        Wave Rf%*256+1-8*(Rf%<>0),1,Wform%,Wper%
        Text 100,35,"SOUND 1,8,"+Str$(Per%)+" "       ! Info-line
        X%=Per%/4096*280+20
        Color 0
        Line Xa%,1,Xa%,18
        Color 1
        Line X%,1,X%,18
        Xa%=X%
        Pause 2                                     ! a short pause otherwise
    Return                                          ! it would be too fast
    Procedure Wave.per                         ! a selection of the Wave period
        S%=10^(K%-1)                      ! left:%=1/right:s%=10/both:s%=100
        If X%>300 And X%<320
            Wper%=Min(Wper%+S%,65535)
        Else
            If X%>0 And X%<20
```

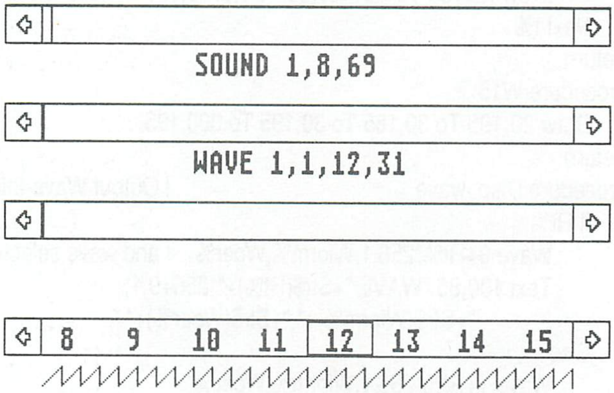
```
            Wper%=Max(Wper%-S%,0)
        Else
            If X%>20 And X%<300
                Wper%=(X%-20)/280*65536
            Endif
        Endif
    Endif
    @Disp_wave                                  ! Info-line
    X%=Wper%/65536*280+20
    Color 0
    Line Xb%,51,Xb%,68
    Color 1
    Line X%,51,X%,68
    Xb%=X%
Return
Procedure Noise                                 ! setting noise period
    If X%>300 And X%<320
        Rf%=Min(Rf%+1,31)
    Else
        If X%>0 And X%<20
            Rf%=Max(Rf%-1,0)
        Else
            If X%>20 And X%<300
                Rf%=(X%-20)/280*32
            Endif
        Endif
    Endif
    @Disp_wave                                  ! Info-line
    X%=Rf%/32*280+20
    Color 0
    Line Xd%,101,Xd%,118
    Color 1
    Line X%,101,X%,118
    Xd%=X%
Return
Procedure Wave.form                             ! set envelope curve
    If X%>300 And X%<320
        Wform%=Min(Wform%+1,15)
    Else
        If X%>0 And X%<20
```

```
            Wform%=Max(Wform%-1,8)
        Else
            If X%>20 And X%<300
                Wform%=(X%-20)/280*8+8
            Endif
        Endif
    Endif
    @Disp_wave                                  ! Info-line
    X%=(Wform%-8)*35+20
    Color 0
    Box Xc%,161,Xc%+34,178
    Color 1
    Box X%,161,X%+34,178
    Xc%=X%
    Deffill 0
    Pbox 20,180,300,199
    On Wform%-7 Gosub W8,W9,W10,W11,W12,W13,W14,W15
Return                              ! This ON-GOSUB serves to quickly
Procedure W8                        ! display the envelope curve
    For I%=20 To 290 Step 10
        Draw I%,185 To I%+10,195 To I%+10,185
    Next I%
Return
Procedure W9
    Draw 20,185 To 30,195 To 300,195
Return
Procedure W10
    For I%=20 To 280 Step 20
        Draw I%,185 To I%+10,195 To I%+20,185
    Next I%
Return
Procedure W11
    Draw 20,185 To 30,195 To 30,185 To 300,185
Return
Procedure W12
    For I%=20 To 290 Step 10
        Draw I%,195 To I%+10,185 To I%+10,195
    Next I%
Return
Procedure W13
```

```
        Draw 20,195 To 30,185 To 300,185
    Return
    Procedure W14
        For I%=20 To 280 Step 20
            Draw I%,195 To I%+10,185 To I%+20,195
        Next I%
    Return
    Procedure W15
        Draw 20,195 To 30,185 To 30,195 To 300,195
    Return
    Procedure Disp_wave                         ! Output Wave-Info
        If Rf%
            Wave 9+Rf%*256,1,Wform%,Wper%     ! and wave selection
            Text 100,85,"WAVE "+Str$(Rf%)+"*256+9,1,
                    "+Str$(Wform%)+","+Str$(Wper%)+" "
        Else
            Wave 1+Rf%*256,1,Wform%,Wper%
            Text 100,85,"WAVE 1,1,"+Str$(Wform%)+","+Str$(Wper%)+" "
        Endif
    Return
    Procedure Draw_box(Y%)              ! subroutine to display boxes with
        Box 0,Y%,319,Y%+19              ! arrows on both sides
        Line 19,Y%,19,Y%+19
        Line 300,Y%,300,Y%+19
        Text 6,Y%+16,Chr$(4)
        Text 307,Y%+16,Chr$(3)
    Return
```

The program displays four horizontal rectangles. If you click in any of the rectangles, a vertical line will appear in the first rectangle. The first rectangle indicates the period of the tone. Directly below that rectangle, the corresponding *SOUND* command is shown. The next rectangle selects the period of the envelope; after which, the rectangle for the period of noise is shown. The last rectangle allows you to set the envelope curve. A graphic display of that curve is shown below that rectangle. The corresponding *WAVE* command is also shown.

Figure 3: SOUND and WAVE



You can, by the way, also listen to the whole thing.

By clicking inside of the rectangle, you can change the corresponding value. And by clicking the arrows, the change is made in single steps. The *WAVE*-period allows you to move in ten step increments by pressing the right mouse button and by one hundred step increments by pressing both mouse buttons.

The drawing of the rectangles with the arrows is performed by procedure *draw_box*. The variables *Xa%,* etc. serve to store the coordinates for markers. The *TEXT*-command writes the value 8 to 15 into the envelope curve rectangle.

The program performs a loop until a mouse button is pressed. If the mouse *Y*-position indicates that a rectangle was chosen, the corresponding routine is called. The main loop is never ending. Every program should always have an exit from a loop, but it was omitted here for clarity.

The next procedures contain routines for each of the rectangles. By using the *MAX* and the *MIN* command, you can easily increase or decrease a value and still stay within bounds.

The envelope curve could have been drawn using the *GET/PUT* command, which would have been faster but would have used more memory. Speed is not that important in this program.

The operating system of the *ST* computer contains a hardcopy routine, which is very easy to call. This routine, however, has a small drawback: The picture of a circle is indeed round, but definitely it is not a circle. The following is a small routine that uses the *Plotter-graphic* mode of an *Epson*-compatible printer to draw a circle.

```
' hardi
Graphmode 3
For I%=0 To 639
    Line I%,0,639-I%,399
Next I%
For I%=0 To 399
    Line 639,I%,0,399-I%
Next I%
T%=Timer
@Hardcopy
Lprint
Lprint Timer-T%
Out 0,12
T%=Timer
Hardcopy
Lprint
Lprint Timer-T%
Out 0,12
Procedure Hardcopy
```

```
    A$=Space$(400)
    G$=" "+Chr$(27)+"*"+Chr$(5)+Chr$(400)+Chr$(400/256)
    Open "",#99,"LST:"
    For S%=Xbios(3) To S%+79
        X%=Varptr(A$)
        For Q%=S%+399*80 To S% Step -80
            Poke X%,Peek(Q%)
            Inc X%
        Next Q%
        Print #99,G$;A$;Chr$(13);
        Print #99,Chr$(27);"J";Chr$(24);
    Next S%
    Close #99
Return
```

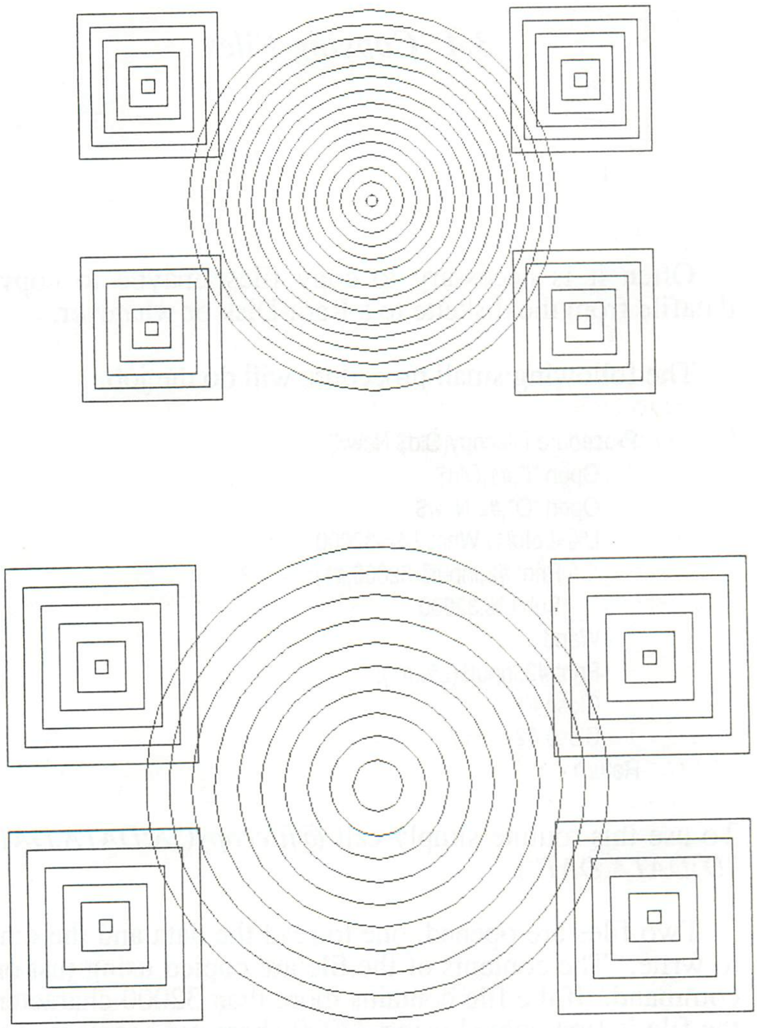
Within procedure Hardcopy, the string *A$* is initialized with 400 spaces. This string serves as the buffer for each graphic line. This buffer is not really necessary, but it serves to repeat the print line (PRINT #99,G$,A$;Chrs$(13);) in order to produce a darker imprint.

In this example, a *STAR SD-19* printer using *IBM* mode was used. The control sequence *ESC-*-5-400* was used to enable the graphics mode with 400 columns and *ESC-J-24* was used to set the line feed to 8 dots (you may have to replace these values with the ones taken from your own printer manual).

You have probably already noticed that the hardcopy was rotated by 90 degrees. This is necessary since 600 dots will not fit on one print line while in *Plotter-mode*. Besides, this method allows for much faster retrieval from memory (by dots).

This routine is somewhat faster than the corresponding *GEM* routine, but that could change with a different printer.

Figure 4: HARDCOPY Routine



3.4 Copying Files

Often it is necessary to copy files, maybe to copy a datafile from the diskette to a Ram-Disk or whatever.

The following small procedure will do the job.

```
Procedure Filecopy(Old$,New$)
    Open "I",#1,Old$
    Open "O",#2,New$
    L%=Lof(#1) While L%>32000
        Print #2,Input$(32000,#1);
        Sub L%,32000
    Wend
    Print #2,Input$(L%,#1);
    Close #1
    Close #2
Return
```

To use this routine simply call *@filecopy*("*A:DATA.DAT*", "*D:DATA.DAT*").

Two files are opened, one to read the data and the other to write. The contents of the file are copied using just one command. If the file contains more than 32000 characters, the file is first copied using 32000 character segments and, finally, the remaining characters are copied. One could copy less than 32000 characters at one time, but this would affect the total copy time.

One could also load the complete file into the memory, make changes, and then copy the file back to the diskette. It is also possible to save the old file as a *BAK*-file as follows:

```
If Exist("FILE.DAT")
If Exist("FILE.BAK")
Kill "FILE.BAK")
Endif
Name "FILE.DAT" as "FILE.BAK"
Endif
'Now you may write the file in the normal fashion
```

Thus: If the file exists, it is renamed to *BAK* and a check is made to see if a *BAK*-file by that name already exists, in which case it is deleted.

3.5 Scan Codes

The keyboard not only transmits an *ASCII* code but also a *SCAN*-Code for every key. For example, all keys that are not assigned in the *ASCII* table return a string containing two characters whenever the *INKEY$* function is called (*Chr$(0)+Chr$(SCAN-Code)*).

BIOS(2,2) or *ON MENU GOSUB* also return the *SCAN*-code besides the *ASCII* value.

Obtain the value of the *SCAN*-code for each key from the table below.

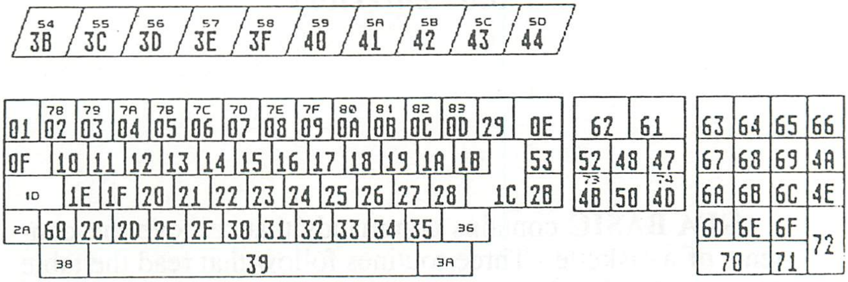
The small number for the function keys represents the *SCAN*-code whenever the key is pressed in combination with the shift key.

The small number above the keys contains the values when the key is pressed in combination with the *ALTERNATE* key.

There are also separate codes for *CONTROL-ARROW-LEFT* and *CONTROL-ARROW-RIGHT*.

The codes for the shift keys (Shift, Control and Alternate) are also shown. They may be used for writing your own keyboard driver. All codes are given in Hexadecimal.

Figure 5: Hexidecimal Key Codes



3.6 Directory

GFA BASIC contains commands that retrieve the contents of a diskette. Three routines follow that read the table of contents (directory).

First, the directory is read into an array, sorted and then printed in three column format. The file size is also printed.

```
' SORT DIR
Dim A$(1000)
@Getdir("*.*",&H37,*A$(),*N%)
@Quicksort(*A$(),0,N%)
For I%=0 To N% Step 3
    Print A$(I%)'''A$(I%+1)'''A$(I%+2)
Next I%
'
'
Procedure Getdir(File_$,Attr%,Str.arr%,Num.%)
    Local I_%,E_%,X$
    Swap *Str.arr%,File$()
    Void Gemdos(26,L:Basepage+128)                  ! setdta
    File_$=File_$+Chr$(0)
    E_%=Gemdos(78,L:Varptr(File_$),Attr%)          ! fsfirst
    While E_%=0 !more Files
        X$=Space$(20)
        Bmove Basepage+158,Varptr(X$),14
```

```
            X$=Left$(X$,Instr(X$,Chr$(0))-1)
            X$=Left$(X$+Space$(20),15)
            L$=Space$(7)
            If Peek(Basepage+149) And 16
                            !dta+21=attribute
              Rset L$="<<DIR>>"
            Else
              Rset L$=Str$(Lpeek(Basepage+154))
                                                  ! dta+26=file size
            Endif
            X$=X$+L$
            File$(I_%)=X$
            Inc I_%
            E_%=Gemdos(79)                        ! fsnext
          Wend
          File$(I_%)=""
          Swap *Str.arr%,File$()
          *Num.%=I_%-1                        ! Highest Index with data
        Return
        '
        Procedure Quicksort(Str.arr%,L%,R%)
          Local X$
          Swap *Str.arr%,A$()
          @Quick(L%,R%)
          Swap *Str.arr%,A$()
        Return
        Procedure Quick(L%,R%)
          Local Ll%,Rr%
          Ll%=L%
          Rr%=R%
          X$=A$((L%+R%)/2)
          Repeat
            While A$(L%)<X$
              Inc L%
            Wend
            While A$(R%)>X$
              Dec R%
            Wend
            If L%<=R%
              Swap A$(L%),A$(R%)
```

```
            Inc L%
            Dec R%
        Endif
    Until L%>R%
        If Ll%<R%
            @Quick(Ll%,R%)
        Endif
        If L%<Rr%
            @Quick(L%,Rr%)
        Endif
    Return
```

By making some small changes, the program could also display the date, the time, the volume name and the data status (read only, hidden, etc.). In this program only one file can be displayed per line.

```
' XDIR
'
Dim A$(1000)
@Getdir("*.*",-1,*A$(),*N%) !-1 means with LABEL
@Quicksort(*A$(),0,N%)
For I%=0 To N%
    Print A$(I%)
Next I%
'
'
Procedure Getdir(File_$,Attr%,Str.arr%,Num.%)
        Local I_%,E_%,X$
    Swap *Str.arr%,File$()
    Void Gemdos(26,L:Basepage+128)              ! setdta
    File_$=File_$+Chr$(0)
    E_%=Gemdos(78,L:Varptr(File_$),Attr%)       ! fsfirst
    While E_%=0                                 ! more Files
        X$=Space$(20)
        Bmove Basepage+158,Varptr(X$),14
        ' Filename
        X$=Left$(X$,Instr(X$,Chr$(0))-1)
        ' fill with equal lengths
        X$=Left$(X$+Space$(20),15)
```

```
' file size or <<DIR>>
L$=Space$(7)
A%=Peek(Basepage+149)              !dta+21 Attribute
If A% And 16
    Rset L$="<<DIR>>"
Else
If A% And 8
    Rset L$="<LABEL> !Diskette name
Else
    Rset L$=Str$(Lpeek(Basepage+154)    ! dta+26=file size
Endif
Endif
    X$=X$+L$
'
' Attribute
'
If A% And 32
    X$=X$+" ."                  ! . means set archive bit
Else                            ! is seldom done by TOS
    X$=X$+" "
Endif
If A% And 16                    ! D = Directory
    X$=X$+"D"
Else
    X$=X$+" "
Endif
If A% And 8                     ! L = Label
    X$=X$+"L"
Else
    X$=X$+" "
Endif
If A% And 4                     ! S = Systemfile
    X$=X$+"S"
Else
    X$=X$+" "
Endif
If A% And 2                     ! H = hidden File
    X$=X$+"H"
Else
    X$=X$+" "
```

```
            Endif
            If A% And 1                          ! R = read-only
                X$=X$+"R"
            Else
            X$=X$+" "
            Endif
            '
            ' Data              dpeek(dta+24)
            '
            D%=Dpeek(Basepage+152)
            D$=" "+Right$("0"+Str$(D%/32 And 15),2)+"/"
            D$=D$+Right$("0"+Str$(D% And 31),2)+"/"
            D$=D$+Str$(D% Div 512+1980)
            X$=X$+D$
            '
            ' time              dpeek(dta+22)
            '
            T%=Dpeek(Basepage+150)
            T$=" "+Right$("0"+Str$(T% Div 2048),2)+":"
            T$=T$+Right$("0"+Str$(T% Div 32 And 63),2)+":"
            T$=T$+Right$("0"+Str$(T%+T% And 63),2)
            X$=X$+T$
            File$(I_%)=X$
            Inc I_%
            E_%=Gemdos(79)                       ! fsnext
        Wend
        File$(I_%)=""
        Swap *Str.arr%,File$()
        *Num.%=I_%-1                             ! Highest index with data
    Return
    '
    Procedure Quicksort(Str.arr%,L%,R%)
        Local X$
        Swap *Str.arr%,A$()
        @Quick(L%,R%)
        Swap *Str.arr%,A$()
    Return
    Procedure Quick(L%,R%)
        Local Ll%,Rr%
        Ll%=L%
```

```
Rr%=R%
X$=A$((L%+R%)/2)
Repeat
    While A$(L%)<X$
        Inc L%
    Wend
    While A$(R%)>X$
        Dec R%
    Wend
    If L%<=R%
        Swap A$(L%),A$(R%)
        Inc L%
        Dec R%
    Endif
Until L%>R%
    If Ll%<R%
        @Quick(Ll%,R%)
    Endif
    If L%<Rr%
        @Quick(L%,Rr%)
    Endif
Return
```

Figure 6: Example of SORTDIR

BAUDRATE.BAS	608	BAUDTEST.BAS	528	BLITDEMO.BAS	1062
BLTMODES.BAK	800	BLTMODES.BAS	800	BOXRSC.BAS	1212
CONRWS.LST	335	DIALOG.BAK	3144	DIALOG.BAS	3144
DIALOG.RSC	1268	ELISE.BAS	2486	ELISE.SND	582
ELISEDMO.BAS	566	EVNT.LST	1345	EXEC3.BAS	308
EXEC3.LST	243	FADEN.BAS	552	FBOXTEXT.BAS	1448
FONTTEST.BAS	2236	FORM.LST	636	FS.BAK	1644
FS.BAS	1658	FS.TTP	7146	FSEL.LST	102
GFABASIC.PRG	57378	GRAF.LST	1742	GROSS.FNT	16984
HARDI.BAS	732	INPUT.BAS	3212	JOYSTICK.BAS	848
KEYTAB.BAS	640	LUPE.ASM	1045	LUPE.BAS	1268
LUPE.PRG	150	MAKEFONT.BAS	2904	MAKEICON.BAS	954
MAKEPRPT.BAS	2118	MENU.LST	897	MIDIBUF.BAS	444
MINIDAT.BAS	1278	MIRRORPU.BAS	1150	MKDATAW.LST	350
MINIDAT.BAS	1278	MIRRORPU.BAS	1150	MKDATAW.LST	350
MOUSE.BAS	1030	MOUSE.LST	441	OBJC.LST	1451
PG274DEM.BAS	11622	PG75DEM1.BAK	1092	PG75DEM1.BAS	1092
PG75DEM2.BAK	1092	PG75DEM2.BAS	1092	QSORT.BAS	832
QSORT.LST	743	QS_TEST.BAS	920	REKURS.BAS	772
RS232BUF.BAS	374	RSCTEST.BAS	2378	RSRC.LST	591
SCREEN.ASM	2930	SCREEN.PRG	320	SCREENTS.BAS	448
SCROLL.BAS	504	SCRP.LST	139	SEARCH.BAS	1494
SHEL.LST	807	SLIDER.BAS	3006	SLIDER.RSC	228
SORTDIR.BAS	1254	SOUNDEXP.BAS	3428	TEST.DAT	13098
TEST.RSC	208	WIND.LST	1286	WIND.RSC	4320

Figure 7: Example of XDIR

```
.              <<DIR>>  D   00/14/2055 03:40:00
..             <<DIR>>  D   00/14/2055 03:40:00
APPL.LST           651      04/22/1987 00:01:10
BAUDRATE.BAS       608      04/22/1987 00:01:18
BAUDTEST.BAS       528      04/22/1987 00:01:26
BLITDEMO.BAS      1062      04/22/1987 00:01:34
BLTMODES.BAK       800      04/22/1987 01:50:32
BLTMODES.BAS       800      04/22/1987 01:58:16
BOXRSC.BAS        1212      04/22/1987 00:01:42
CONRWS.LST         335      04/22/1987 00:01:50
DIALOG.BAK        3144      04/22/1987 00:01:58
DIALOG.BAS        3144      04/22/1987 00:02:06
DIALOG.RSC        1268      04/22/1987 00:02:14
ELISE.BAS         2486      04/22/1987 00:02:22
ELISE.SND          582      04/22/1987 00:02:30
ELISEDMO.BAS       566      04/22/1987 00:02:38
EVNT.LST          1345      04/22/1987 00:02:46
EXEC3.BAS          308      04/22/1987 00:02:54
EXEC3.LST          243      04/22/1987 00:03:02
FADEN.BAS          552      04/22/1987 00:03:10
FBOXTEXT.BAS      1448      04/22/1987 00:03:20
FONTTEST.BAS      2236      04/22/1987 00:03:28
FORM.LST           636      04/22/1987 00:03:36
FS.BAK            1644      04/22/1987 00:03:46
FS.BAS            1658      04/22/1987 00:03:54
FS.TTP            7146      04/22/1987 00:04:04
FSEL.LST           102      04/22/1987 00:04:12
GRAF.LST          1742      04/22/1987 00:04:36
GROSS.FNT        16984      04/22/1987 00:04:46
HARDI.BAS          732      04/22/1987 00:04:56
INPUT.BAS         3212      04/22/1987 00:05:04
JOYSTICK.BAS       848      04/22/1987 00:05:14
KEYTAB.BAS         640      04/22/1987 00:05:22
LUPE.ASM          1045      04/22/1987 00:05:30
LUPE.BAS          1268      04/22/1987 00:05:40
LUPE.PRG           150      04/22/1987 00:05:48
MAKEFONT.BAS      2904      04/22/1987 00:05:58
MAKEICON.BAS       954      04/22/1987 00:06:06
```

Example of XDIR (Cont.)

```
MAKEPRPT.BAS    2118        04/22/1987 00:06:16
MENU.LST         897        04/22/1987 00:06:24
MIDIBUF.BAS      444        04/22/1987 00:06:32
MINIDAT.BAS     1278        04/22/1987 00:06:42
MIRRORPU.BAS    1150        04/22/1987 00:06:50
MKDATAW.LST      350        04/22/1987 00:07:00
MOUSE.BAS       1030        04/22/1987 00:07:08
MOUSE.BAS       1030        04/22/1987 00:07:08
MOUSE.LST        441        04/22/1987 00:07:18
OBJC.LST        1451        04/22/1987 00:07:26
QSORT.BAS        832        04/22/1987 00:08:26
QSORT.LST        743        04/22/1987 00:08:34
QS_TEST.BAS      920        04/22/1987 00:08:44
REKURS.BAS       772        04/22/1987 00:08:52
RS232BUF.BAS     374        04/22/1987 00:09:00
RSCTEST.BAS     2378        04/22/1987 00:09:10
RSRC.LST         591        04/22/1987 00:09:20
SCREEN.ASM      2930        04/22/1987 00:09:28
SCREEN.PRG       320        04/22/1987 00:09:38
SCREENTS.BAS     448        04/22/1987 00:09:46
SCROLL.BAS       504        04/22/1987 00:09:56
SCRP.LST         139        04/22/1987 00:10:04
SEARCH.BAS      1494        04/22/1987 00:10:14
SHEL.LST         807        04/22/1987 00:10:24
SLIDER.BAS      3006        04/22/1987 00:10:32
SLIDER.RSC       228        04/22/1987 00:10:42
SORTDIR.BAS     1254        04/22/1987 00:10:52
SOUNDEXP.BAS    3428        04/22/1987 00:11:00
TEST.DAT       13098        04/22/1987 00:11:12
TEST.RSC         208        04/22/1987 00:11:22
WIND.LST        1286        04/22/1987 00:11:30
WIND.RSC        4320        04/22/1987 00:11:40
WINDM.RSC       2520        04/22/1987 00:11:50
WINDOW.BAK     11620        04/22/1987 00:12:00
WINDOW.BAS     11680        04/22/1987 00:01:50
WIND_MID.BAS   11714        04/22/1987 00:12:12
WOOF1.PI2      32034        04/22/1987 00:12:24
WOOF1.PI3      32034        04/22/1987 00:12:38
XDIR.BAS        2182        04/22/1987 00:12:48
XPRPATCH.PRG     544        04/22/1987 00:12:58
```

The final program allows you to display the contents of a diskette or a hard disk partition, or it allows you to search the diskette for a particular file and then display the full name including the pathname.

```
' SEARCH
'
@Search("A:\","*.ASM","CON:")
'
'
Procedure Search(Path$,File$,Out$)
    Oldpath$=Dir$(0)
    Olddrv%=Gemdos(25)+1
    Open "O",#1,Out$
    If Instr(Path$,":")
        Chdrive Asc(Path$) And 31
        Path$=Mid$(Path$,Instr(Path$,":")+1)
    Endif
    Chdir Path$
    Void Gemdos(26,L:Basepage+128)                    ! setdta
    File$=File$+Chr$(0)
    Star$="*.*"+Chr$(0)
    Drv$=Chr$(Gemdos(25)+65)+":"
    @Search1
    Close #1
    Chdir "\"+Olddir$
    Chdrive Olddrv%
Return
Procedure Search1
    Local W%
    @Fsfirst
    While E%=0
        Print #1,Drv$+Dir$(0)+"\"+X$
        @Fsnext
    Wend
    @Fsfirstdir
    Q%=0
    While E%=0
        If T% And 16
```

```
            If X$<>"." And X$<>".."
               W%=Q%
               Chdir X$
               @Search1
               Chdir ".."
               @Fsfirstdir
               Q%=0
               While W%<>Q%
                  Void Gemdos(79)
                  Inc Q%
               Wend
            Endif
         Endif
         @Fsnext
         Inc Q%
      Wend
   Return
   Procedure Fsfirst
      E%=Gemdos(78,L:Varptr(File$),&H27)           ! fsfirst
      @Getnam
   Return
   Procedure Fsfirstdir
      E%=Gemdos(78,L:Varptr(Star$),16)
      @Getnam
   Return
   Procedure Fsnext
      E%=Gemdos(79)
      @Getnam
   Return
   Procedure Getnam
      If E%
         X$=""
         T%=0
      Else
         X$=Space$(20)
         Bmove Basepage+158,Varptr(X$),14
         X$=Left$(X$,Instr(X$,Chr$(0))-1)
         T%=Peek(Basepage+149)                    ! dta+21 Attribute
      Endif
   Return
```

If you replace the *PRINT*-command with:

```
File$(I%)=.......
Inc I%
```

This would make it possible to sort the directory by filename. In this case, I would address the field directly rather than by pointer since the complete *Directory-tree* is seldom used.

After the *fsfirst* or *fsnext* (*GEMDOS(78/79)* call, this program returns the filename and other information about the file via the *DTA*-buffer.

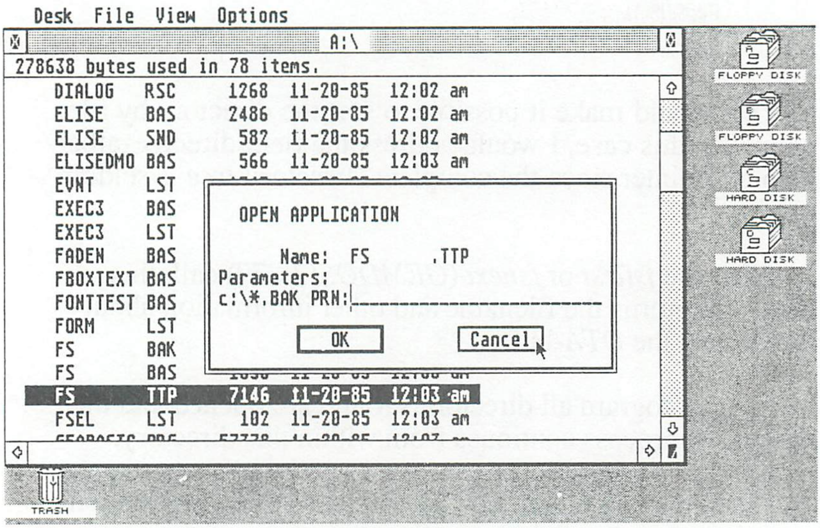
In this program all directories found are opened and then the search process continues from within this directory.

When the end of a directory is reached, the program will read the previous read parent directory again. This is not very elegant or quick, but it is faster to program.

You could have created a separate buffer for each directory which would have saved you the time necessary to read the previous directory.

■ 3.6.1 TTP

Figure 8: TTP



The previous search program could be changed for the compiler version of **GFA BASIC**. Just add the following lines to the beginning of the program, compile it and then save it as a .TTP file. Now you can pass the parameters as text from the desktop or a shell (or even through the *EXEC*-command). Parameters are as follows: Pathname, including the drive ("C:") and the root directory ("\"); the name of the file you wish to find (you may also use wildcards like "*.*"); and the output device ("*CON*:" or "*PRN*:" or "*FILE.EXT*"). Do not include the quotation marks.

```
'FS.TTP
'
a$=SPACE(128)
BMOVE BASEPAGE+129,VARPTR(a$),127
```

```
FOR i%=1 TO LEN(a$)
    IF MID$(a$,1%)=" "
        MID$(a$,i%)=CHR$(0)
    ENDIF
NEXT i%
path$=a$
file$= MID$(a$,INSTR(a$,CHR$(0))+1)
ofile$=MID$(file$,INSTR(file$,CHR$(0))+1)
'@search(path$,file$,ofile$)
'
'followed by the previous program starting with PROCEDURE search
'
```

3.7 Formatting Diskettes

At times, it might be useful to format a diskette from within a program or you might need more storage on your diskette. The following procedure is for that purpose.

The parameters for the *format-call* are as follows:

drv% Drive number, 0 for A:, 1 for B:

sid% 1 for single-sided diskettes, 2 for double-sided diskettes. (Disk drive must be capable of selected option)

trk% Number of tracks diskette should contain (usually 80, but 81 and 82 tracks are possible with most drives)

spt% Number of sectors per track (normally 9, but 10 sectors are also possible)

fat% Size of the file allocation table, usually 5. One and a half bytes per sector are normally used (One FAT-sector per 340 diskette sectors).

dir% Maximum number of files diskette may contain (the standard is 112, must be a multiple of 4 starting with 16).

med% Media number, a number that describes the diskette type. The only importance on the *ST* appears to be that the number is even for single-sided diskettes and odd for double-sided diskettes.

The normal format is as follows:

	sid%	trk%	spt%	fat%	dir%	med%
single-sided	1	80	9	5	112	248
double-sided	2	80	9	5	112	249

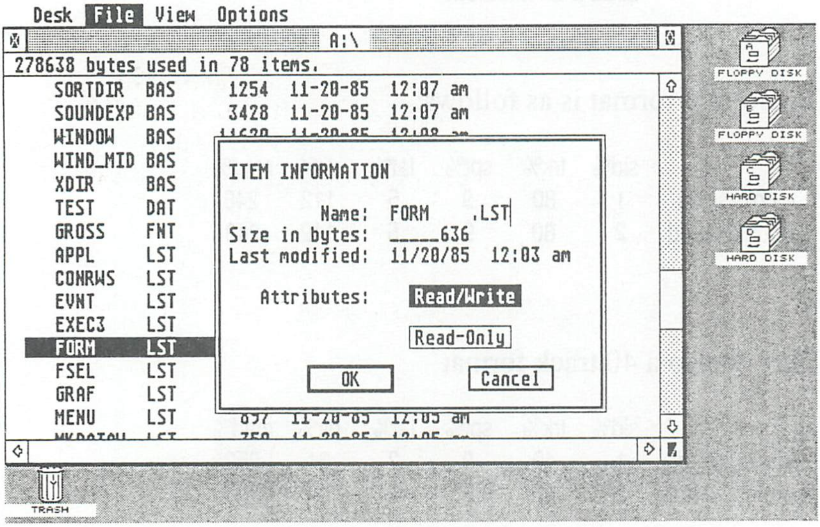
There is also a 40 track format

	sid%	trk%	spt%	fat%	dir%	med%
single-sided	1	40	9	2	64	252
double-sided	2	40	9	2	112	253

For example:

@format(0,2,82,10,6,160,101) will format a double-sided diskette with 82 tracks consisting of 10 sectors per track. Diskettes that were formatted by using a format other than the standard format routine (desktop format command) may not be duplicated by placing drive icons together, but rather must be copied a file at a time. You could, of course, write your own copy routine.

Figure 9: FORMAT.LST




```
Procedure Format(Drv%,Sid%,Trk%,Spt%,Fat%,Dir%,Med%)
    Buf$=Space$(1000)
    Void Fre(0)
    Buf%=Varptr(Buf$)
    For T%=0 To Trk%-1
        For S%=0 To Sid%-1
            E%=Xbios(10,L:Buf%,L:0,Drv%,Spt%,T%,S%,1,L:&H876
                                                    54321,0)
            If E%
                Print
                Print "Side ";S%;" Track ";T%;" Error ";E%;" sector ";
                B%=Buf%
                While Dpeek(B%)
                    Print Dpeek(B%)'
                    Add B%,2
                Wend
            Else
                Out 5,42
```

```
            Endif
        Next S%
    Next T%
    Sec%=Trk%*Spt%*Sid%
    Buf$=String$(6,0)+Mkl$(Xbios(17)+Chr$(0)+Mki$(2)+Chr$(2)
    Buf$=Buf$+Mki$(&H100)+Chr$(2)+Chr$(Dir%)+Chr$(Dir%/256)
    Buf$=Buf$+Chr$(Sec%)+Chr$(Sec%/256)+Chr$(Med%)
    Buf$=Buf$+Mki$(Fat%*256)+Mki$(Spt%*256)+Mki$(Sid%*256)
    Buf$=Buf$+Mki$(0)+String$(512,0)
    Void Xbios(9,L:Varptr(Buf$),L:0,Drv%,1,0,0,1)
    Void Bios(7,Drv%)
    Buf$=Mkl$(&HF7FFFF00)+String$(508,0)
    Void Bios(4,1,L:Varptr(Buf$),1,1,Drv%)
    Void Bios(4,1,L:Varptr(Buf$),1,Fat%+1,Drv%)
    Print
    Print Dfree(Drv%+1);" Bytes free"
Return
```

The procedure starts by initializing a string to serve as the buffer for the format routine *XBIOS(10....)*. To make sure that the string is not moved during the garbage collection (cleanup, whenever the memory allocated for the storage of strings is exceeded), a *FRE(0)* call is issued. That string address is then passed to a variable.

Next, all tracks are formatted starting with track 0. It makes more sense to alternate between sides on double-sided drives, otherwise all tracks on side 0 would be formatted before side 1.

The actual formatting is performed with the *XBIOS(10....)* call. Should an error occur during the formatting, a list of all bad sectors is displayed without interrupting the formatting.

After all tracks are formatted (unfortunately, there is no error message if the maximum track size is exceeded), the boot sector is written to the diskette. All those *Buf$* assignments up to *XBIOS(9...)* are used for that purpose.

The *BIOS*(7...) call reads the newly created boot sector from the diskette so that the start of the *FAT* table may be written to the diskette with the *BIOS(4...)* call. The *FAT* is always written to two different locations on the diskette and always starts with *F7 FF FF FF*.

Finally, the amount of available storage on this newly formatted diskette is displayed on the screen.

You could further modify this routine to verify the just formatted tracks or overwrite the tracks with information from another diskette.

By the way, this routine will *not* copy protected diskettes. It would require a lot more code to accomplish that task. Programmers that use the format routine as copy protection would not appreciate it if those details were made public.

3.8 Printers

There exists a variety of printers and computers with many different character sets. Since most of the sets are foreign characters, I have included a small patch program written in **GFA BASIC** that will convert those characters. Because this task is easier to accomplish with an assembler, I have written the program so that it loads the machine code using *DATA* statements.

```
' MAKEPRPT
Mc$=""
Do
    Read A%
    Exit If A%<0
    Mc$=Mc$+Mki$(A%)
Loop
Mid$(Mc$,Len(Mc$)-17)="EPSON "+Chr$(0)
Do
    Read A$
    Exit If A$=""
    If Val?(A$)=Len(A$)
        A$=Chr$(Val(A$))
    Endif
    B$=""
    Do
        Read C$
        Exit If C$=""
```

```
            If Val?(C$)=Len(C$)
                B$=B$+Chr$(Val(C$))
            Else
                B$=B$+C$
            Endif
        Loop
        Mc$=Mc$+Chr$(Len(B$))+A$+B$
    Loop
    Mc$=Mc$+Chr$(-1)
    If Len(Mc$) And 1
        Mc$=Mc$+Chr$(0)
    Endif
    '
    ' MID$(mc$,109)=MKI$(2)          ! This line for serial printers
    '
    Open "O",#1,"XPRPATCH.PRG"
    Print #1,Mki$(&H601A);Mkl$(Len(Mc$));String$(22,0);Mc$;Mkl$(0);
    Close #1
    ' MCODE \XPTGRAPH.PRG
    DATA 24576,204,4660,34661,12311,20072,2048,13
    DATA 26372,16879,6,3160,64,26204,3160,65533
    DATA 26198,8728,26450,16135,18513,8784,18513,24596
    DATA 4121,24650,4288,17914,65452,45514,25866,24860
    DATA 18951,26122,16890,65338,21377,27364,24846,37855
    DATA 18951,26114,8201,8799,15903,20083,18513,12033
    DATA 17402,65310,37321,18513,18512,16188,65533,16188
    DATA 64,18554,44,16615,20217,4660,22136,18514
    DATA 17914,76,16967,7706,27400,45082,26378,54471
    DATA 24820,9311,24734,4314,20943,65532,9311,24726
    DATA 22671,45215,22215,22671,8735,8799,20085,28672
    DATA 4120,4800,27400,4824,20936,65532,24816,8201
    DATA 37002,512,254,12032,16188,49,20033,17400
    DATA 65535,24894,3232,4660,34661,26126,8800,24882
    DATA 16890,102,24894,16999,20033,17402,65308,24866
    DATA 17914,65294,9352,17914,65410,9352,16890,78
    DATA 24866,16890,170,17402,65478,16999,17914,65009
    DATA 24732,18513,16188,33,16188,5,20045,8256
    DATA 20623,20085,18512,16890,48,24848,8287,24844
    DATA 28927,20936,65534,20936,65534,20085,18512,16188
    DATA 9,20033,23695,20085,28271,29728,26990,29556
```

```
DATA 24940,27749,25613,2560,3338,18246,16672,28769
DATA 29795,26656,26223,29216,20562,20026,3338,12601
DATA 14390,8263,17985,8275,31091,29797,28020,25955
DATA 26734,26987,3338,8224,8224,8262,29281,28267
DATA 8271,29556,29295,30579,27497,3338,17744,21327
DATA 20000,8224,8224,8224,8224,8192,0
DATA -1
DATA Ä,27,R,2,Ä,27,R,0,
DATA Ö,27,R,2,Ö,27,R,0,
DATA Ü,27,R,2,Ü,27,R,0,
DATA ä,27,R,2,ä,27,R,0,
DATA ö,27,R,2,ö,27,R,0,
DATA ü,27,R,2,ü,27,R,0,
DATA ß,27,R,2,ß,27,R,0,
DATA 225,27,R,2,ß,27,R,0,
DATA δ,27,R,2,δ,27,R,0,
DATA
```

The last few *DATA* statements show how the character set is submitted. First, the character to be replaced is given. Then the Control character sequence of the new character is given. Each line ends with a zero. Two zeroes in a row terminate the table. The characters can be supplied as letters, as *ASCII* values or as the hexadecimal, octal or binary equivalent. More than one character like:

```
DATA A,ABCDDCBA
```

would result that the letter "*A*" would print "*ABCDDCBA*" whenever it is passed to the printer (with *LPRINT*, *LLIST* or *OPEN.."PRN:"*).

The following short program was used to create the *DATA* statements from the compiled file.

```
' makedataw
FILESELECT "\*.PRG",".PRG",file$
OPEN "I",#1,file$
OPEN "O",#2,"DATA.LST"
SEEK #1,28
```

```
l%=LOF(#1)
PRINT #2,"' MCODE ";file$;
FOR i%=29 TO l%-8 STEP 2                  !evtl -4
    IF ((i%-29) AND 15)=0
        PRINT #2
        PRINT #2,"D ";
    ELSE
        PRINT #2,",";
    ENDIF
    PRINT #2,CVI(INPUT$(2,#1));
NEXT i%
PRINT #2
PRINT #2,"D -1"
```

The first 28 bytes of a *PRG*-file contain information about the program size, which is of no importance when using a relocatable machine program. The last 8 (sometimes 4 depending on the assembler) bytes are null and may not be ignored. These bytes also indicate whether or not the program is relocatable.

The resulting file *DATA.LST* may then be merged into a **GFA BASIC** program which will read the data into a string. This string can then be called using *CALL* or *C:*.

3.9 Magnify

The following example demonstrates how to use a machine-routine in **GFA BASIC** to serve as a magnify function. This program will serve as an example for making your own routines.

```
        section   text         ;xlupe.asm

x:      move.l    4(sp),a0       ;src-adr
        move.w    8(sp),d0       ;width
        move.w    10(sp),d1      ;height
        move.l    12(sp),a1      ;dest-adr
        cmp.w     #400/8,d1

        bhi.s     error
        cmp.w     #640/8,d0
        bhi.s     error
        cmp.l     #$00ffffff,a0
        bhi.s     error
        cmp.l     #$00ffffff,a1
        bhi.s     error

l0:     move.l    a1,a2
        move.w    d0,d3
l1:     move.w    (a0)+,d6
        moveq     #15,d5
l2:     moveq     #0,d7
```

```
            add.w       d6,d6
            bcc.s       l3
            moveq       #$7f,d7

l3:         move.b      d7,80(a1)
            move.b      d7,160(a1)
            move.b      d7,240(a1)
            move.b      d7,320(a1)
            move.b      d7,400(a1)
            move.b      d7,480(a1)
            clr.b       560(a1)
            move.b      d7,(a1)+
            subq.w      #1,d3
            dbeq        d5,l2
            bne.s       l1              ;next word
            lea         640(a2),a1
            dbra        d1,l0
            moveq       #0,d0
            rts
error:      moveq       #-1,d0
            rts
            end
```

This routine has the following attributes:

-short
-fast
-fully relocatable

That is why it is possible to use this routine in the form of *DATA* statements as follows:

```
Lupe$=""
Do
    Read A%
    Exit If A%<0
```

```
        Lupe$=Lupe$+Mki$(A%)
    Loop
    Void Fre(0)
    ' MCODE \BAS\LUPE.PR
    DATA 8303,4,12335,8,12847,10,8815,12
    DATA 3137,50,25172,3136,80,25166,45564,255
    DATA 65535,25158,46076,255,65535,25150,9289,13824
    DATA 15384,31247,32256,56390,25602,32383,4935,80
    DATA 4935,160,4935,240,4935,320,4935,400
    DATA 4935,480,4807,21315,22477,65498,26322,17386
    DATA 640,20937,65480,28672,20085,28927,20085
    DATA -1
    Graphmode 3
    For I%=0 To 319
        Line I%,0,319-I%,319
        Line 0,I%,319,319-I%
    Next I% Graphmode 1
    Do
        Get X%,Y%,X%+39,Y%+39,A$
        Lupe%=Varptr(Lupe$)
        Hidem
        Void C:Lupe%(L:Varptr(A$)+6,40,39,L:Xbios(3)+40)
        Showm
        Repeat
            Mouse A%,B%,C%
        Until C%
        If A%<320-40 And B%<320-40
            X%=A%
            Y%=B%
        Else
            Color C% And 1 Plot X%+(A%-320)/8,Y%+(B%/8)
        Endif
    Loop
```

After the initializing of the magnify routine, a pattern is drawn on the screen. Then a segment is cut from the left side of the screen, after which that segment is enlarged by using the magnify routine. Whenever a mouse button is pressed, the segment is either moved or a point is plotted depending on the mouse position (Graphmode 3).

If you look closely, you will discover an error in the program that forced me to decrease the length of the *C:*-call by one. This error is a holdover from the testing phase of the machine routine.

The screen segment is read into a string with the *GET* command and the address of that string is then incremented by six and passed to the magnify routine. In this case, the destination address happened to be within the screen boundaries, but you could easily change the machine program so that a *GET/PUT*-string could be used as the destination.

By the way, this Basic program is not optimal. A good program should not always call the magnify routine but rather use the *PBOX*, the *PUT,...0* or *PUT,...15* command to set the individual points. The magnify routine should only be called whenever large changes are made (like the drawing of lines and circles or the displacement or inverting of a picture segment).

3.10 Recursion

There exists a very powerful method of programming, which is called recursion. This method usually shortens the programs, but it makes them harder to understand if you are not used to recursive thinking.

With recursion you can solve problems by separating them into ever decreasing steps.

A small example:

```
'recurs
Faktor=0.55
@Rek(320,200,100)            !change to 320,100,50 for color
Procedure Rek(X%,Y%,R%)
    Box X%-R%,Y%-R%,X%+R%,Y%+R%
    If R%>10
        @Rek(X%+R%,Y%,R%*Faktor)
        @Rek(X%,Y%+R%,R%*Faktor)
        @Rek(X%-R%,Y%,R%*Faktor)
        @Rek(X%,Y%-R%,R%*Faktor)
    Endif
Return
```

That was short, was it not?

Call *@rek(320,200,100)* passes the value 320 for *X%* and 200 for *Y%* and also 100 for *R%* (for color monitor change to 160,100,150).

The procedure draws a box with the a length of twice *R%*.

This procedure is called four more times as long as *R%* is greater than 10. The mid-point is always moved by *R%*, first to the right, then to the bottom, then to the left and finally to the top. The important thing is that *R%* is changed every time the procedure is called.

The first of these four procedures draws a box half as large as the original (*R%*Factor* where *Factor*=0.5), whose mid-point is located exactly in the middle of the right side of the larger box. Since *R%* will still be larger than 10, the procedure will draw another box on the right side of this newly created box.

As soon as the lower limit of the box size is reached (*R%* is not greater than 10), the recursion jumps a level higher (the *RETURN* statement).

Next, the bottom procedure is called. This procedure draws a box in similar fashion, but this time the mid-point is always on the lower side. Next, the left procedure draws boxes on the left mid-point.

The last procedure (top) draws the upper mid-point.

By now all of the routines between the *IF* and *ENDIF* have been executed.

Now the bottom box routine is called with a value of *R%* equal to the previous box. In other words, the size is again large enough so that the four subroutines can be executed again. Again, the smaller boxes are drawn.

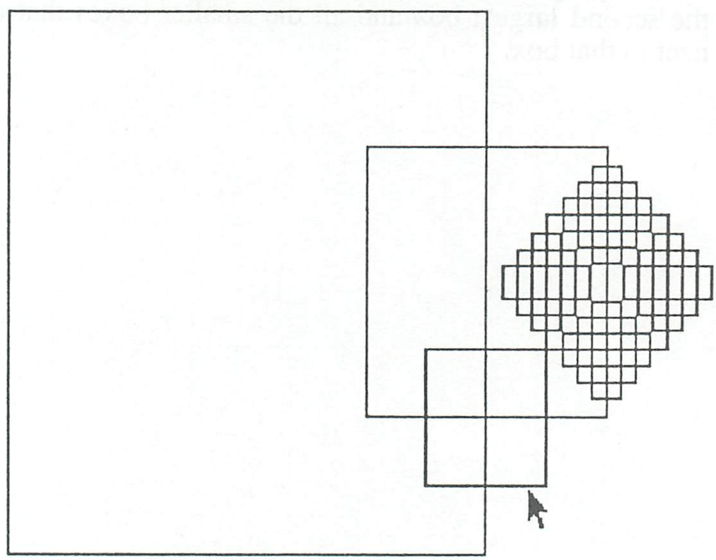
Now the left box routine with the larger box is called, then the top box routine.

Next, the bottom box routine that draws an even larger box is called. This routine again calls all those other routines to draw the smaller boxes. This is then repeated for the left and upper boxes.

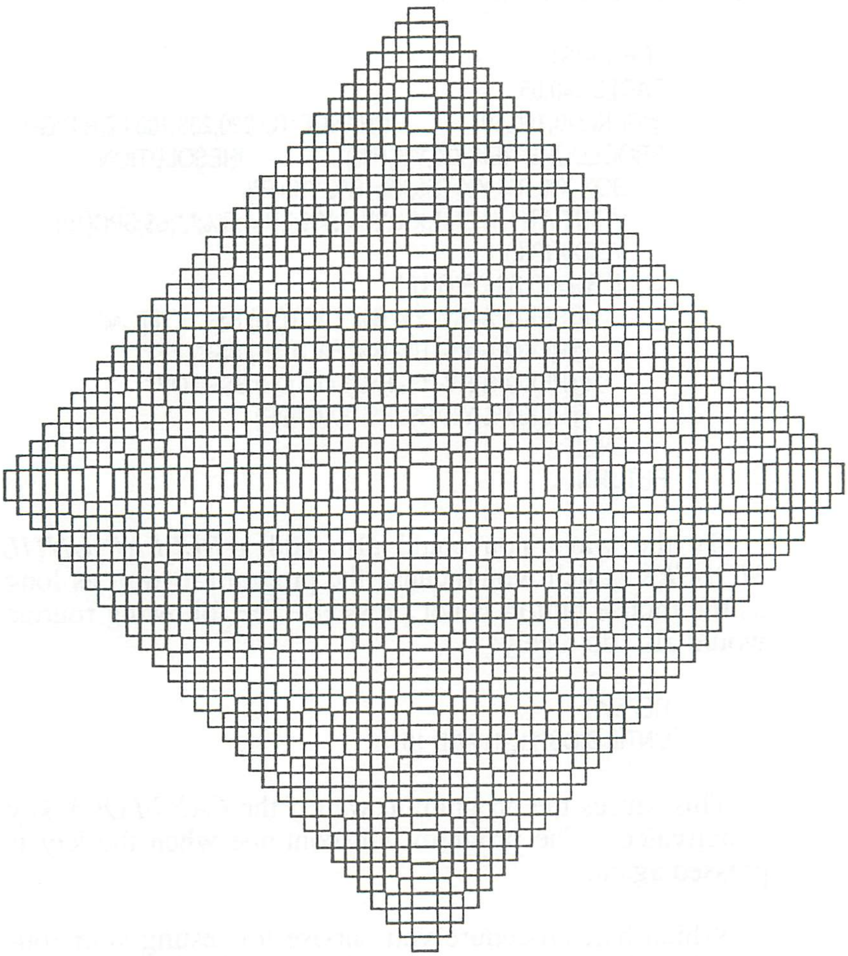
Next, the bottom box routine is called which will draw an even bigger box.

This program continues until the upper box routine drew the second largest box and all the smaller boxes that connect to that box.

Figure 10: An Example of Recursion



The first of these pictures shows the program in progress and the other shows the finished product.



The following modifications to the program will allow you to look at the drawing one box at a time. Values for *X%,Y%,R%* and which of the routines was last called is displayed in the top left corner. Just press any key to continue with drawing the next box.

```
' RECURS1
FACTOR=0.55
@REK(320,100,50)          !CHANGE TO 320,200,100 FOR HIGH
PROCEDURE REK(X%,Y%,R%)         !RESOLUTION
  BOX X%-R%,Y%-R%,X%+R%,Y%+R%
  PRINT AT(1,1);"X=";X%;",Y=";Y%;",R=";R%;",";B$;SPC(10)
  VOID INP(2)
  IF R%>10 B$="RIGHT "
    @REK(X%+R%,Y%,R%*FACTOR) B$="BOTTOM"
    @REK(X%,Y%+R%,R%*FACTOR) B$="LEFT "
    @REK(X%-R%,Y%,R%*FACTOR) B$="TOP "
    @REK(X%,Y%-R%,R%*FACTOR)
  ENDIF
RETURN
```

By the way: You could also add a *REPEAT UNTIL MOUSEK* which would cause the program to wait as long as the mouse button is not pressed. The following routine would also do nicely:

```
REPEAT
UNTIL BIOS(11,-1) AND 16
```

This causes the program to wait if the *CAPS-LOCK* key is activated. The program will continue when the key is pressed again.

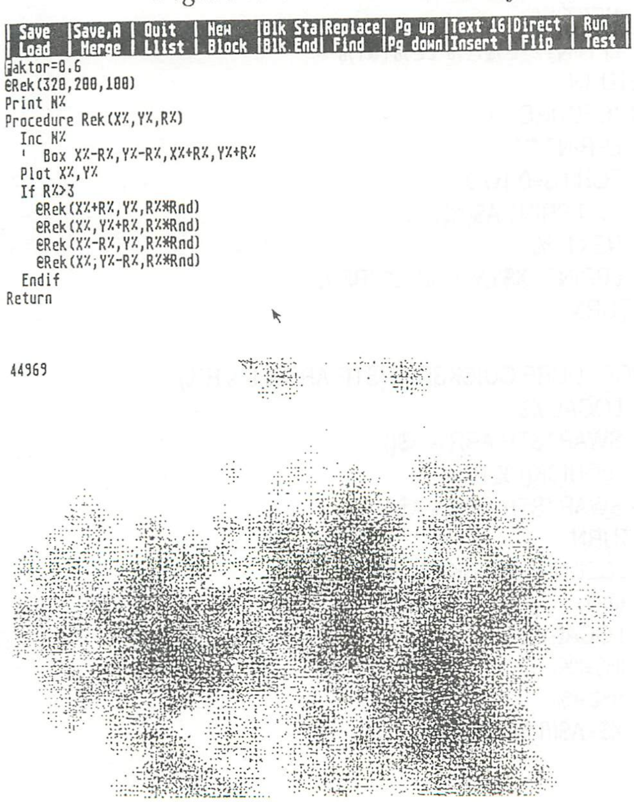
Which halt procedure you choose for testing your routine should depend on the complexity of the recursion. It is absolutely possible that thousands of steps are performed before you will discover the error.

Possible experimental alternatives:

There are many possibilities for modifying this program. This includes changing the range of *R%* (*IF R% > 10*) or changing the factor (The factor must, however, be less than one so that the program will eventually come to a stop, you could of course modify the corresponding *R% > 10*). The factor could be created with the *RND* function, or you could change the *BOX* command to a *CIRCLE X%,Y%,R%*, or you could use a simple *PLOT* command, or ...

One of these modifications is represented in the picture below.

Figure 11: Recursion Modification



```
| Save  |Save,A | Quit  | New   |Blk Sta|Replace| Pg up |Text 16|Direct | Run  |
| Load  | Merge | Llist | Block |Blk End| Find  |Pg down|Insert | Flip  | Test |
Faktor=0.6
@Rek(320,200,100)
Print N%
Procedure Rek(X%,Y%,R%)
  Inc N%
  '  Box X%-R%,Y%-R%,X%+R%,Y%+R%
  Plot X%,Y%
  If R%>3
    @Rek(X%+R%,Y%,R%*Rnd)
    @Rek(X%,Y%+R%,R%*Rnd)
    @Rek(X%-R%,Y%,R%*Rnd)
    @Rek(X%;Y%-R%,R%*Rnd)
  Endif
Return

44969
```

Let us move on to another recursive procedure, namely the often used Quicksort.

```
' QS_TEST.BAS
'
DIM A$(9)
FOR I%=0 TO 9
    READ A$(I%)
NEXT I%
DATA 5,2,4,6,1,3,8,0,9,7
@QUICKSORT(*A$(),0,9)
@D
'
PROCEDURE D
    LPRINT ''
    FOR I%=0 TO 9
        LPRINT A$(I%)'
    NEXT I%
    LPRINT ""'L%'R%''LL%'RR%
RETURN
PROCEDURE D1
    LPRINT "*"'
    FOR I%=0 TO 9
        LPRINT A$(I%)'
    NEXT I%
    LPRINT ''X$'L%'R%''LL%'RR%
RETURN
'
PROCEDURE QUICKSORT(STR.ARR%,L%,R%)
    LOCAL X$
    SWAP *STR.ARR%,A$()
    @QUICK(L%,R%)
    SWAP *STR.ARR%,A$()
RETURN
PROCEDURE QUICK(L%,R%)
    @D
    LOCAL ll%,rr%
    ll%=l%
    rr%=r%
    X$=A$((L%+R%)/2)
```

```
        REPEAT
          WHILE A$(L%)<X$
            INC I%
          WEND
          WHILE A$(R%)>X$
          DEC r%
          WEND
          IF L%<=R%
            SWAP A$(L%),A$(R%)
            @D1
            INC L%
            DEC R%
            ENDIF
        UNTIL L%>R%
        IF LL%<R%
          @QUICK(LL%,R%)
        ENDIF
        IF L%<RR%
          @QUICK(L%,RR%)
        ENDIF
      RETURN
```

This program contains the previously introduced Quicksort, an initialize routine and two report routines.

First, the array *A$()* is filled with data that will then be sorted.

The printer routines *d* and *d1* print the field contents of *A$(),L%,R%,ll%* and *rr%*. The *d1* routine also prints an asterisk and the variable *X$* (this variable is used to compare the elements).

The address of the sort field and the left and right boundaries are then passed to the quicksort routine. The variables *L%* and *R%* select the portion of the array that should be sorted. After the local variable *X$* is declared and the array *A$* is swapped, the program calls the actual quicksort procedure (quick).

This sort routine is the most important routine of the program.

For testing purposes, the print routine (the one without the asterisk) is called. The variable *L%* and *R%* are then passed to the local variables *Ll%* and *Rr%*.

Next, variable *X$* is assigned an element from the array (here the middle element).

The array is then divided into two parts. The left (lower) part contains all elements that are smaller than the compare element and the right (upper) part contains all the elements that are greater. The dividing of the array is as follows:

> Starting with the lowest array element, the table is searched until it is greater or equal to the compare element *(X$)*. Next, the table is searched for the first element by starting with the highest (left) element and searching downward (this is performed by the two *WHILE* loops). Whenever both elements are found, they are switched and then skipped with the *INC* and the *DEC* command (they could have been equal). This is repeated until the left range is greater than the right range, which indicates that both tables contain no more elements which do not belong to them. If the tables contain more then one element, they are then sorted.

Just in case it is still not clear to you, let me give you the listing of the sort process of the table mentioned above.

The original input

5 2 4 6 1 3 8 0 9 7 0 9 0 0 (L%,R%,Ll%,Rr%)

The compare element is 1. The first element on the left that is not less than 1 is 5 and the first element that is not larger is the 0. Those two elements are switched.

```
*0 2 4 6 1 3 8 5 9 7  1 0 7  0 9 (X$,L%,R%,L1%,Rr%)
```

Continue the search. The 2 on the left and the 1 on the right are switched.

```
*0 1 4 6 2 3 8 5 9 7  1 1 4  0 9
```

The left range is now larger then the right. The two parts are 0-1 and 2-9. The 0-1 is sorted first.

```
0 1 4 6 2 3 8 5 9 7  0 1  0 9
```

The zero is the compare elements and it switched with itself.

```
*0 1 4 6 2 3 8 5 9 7  0 0 0  0 1
```

Now we sort the right part 2-9.

```
0 1 4 6 2 3 8 5 9 7  2 9  0 9
```

Compare element is 3, after switching....

```
*0 1 3 6 2 4 8 5 9 7  3 2 5  2 9
*0 1 3 2 6 4 8 5 9 7  3 3 4  2 9
```

...is divided in parts 2-3

```
0 1 3 2 6 4 8 5 9 7  2 3  2 9
*0 1 2 3 6 4 8 5 9 7  3 2 3  2 3
```

...and 4-9

```
0 1 2 3 6 4 8 5 9 7  4 9  2 9
*0 1 2 3 6 4 7 5 9 8  8 6 9  4 9
```

4-9 is split into 4-7 and 8-9

```
0 1 2 3 6 4 7 5 9 8  4 7  4 9
*0 1 2 3 4 6 7 5 9 8  4 4 5  4 7
```

4-7 is split into 4 and 5-7

```
0 1 2 3 4 6 7 5 9 8  5 7  4 7
*0 1 2 3 4 6 5 7 9 8  7 6 7  5 7
```

5-7 is split into 5-6 and 7

```
0 1 2 3 4 6 5 7 9 8  5 6  5 7
*0 1 2 3 4 5 6 7 9 8 6 5 6 5 6
```

Now we do the 8 and 9

```
0 1 2 3 4 5 6 7 9 8  8 9 4 9
*0 1 2 3 4 5 6 7 8 9  9 8 9 8 9
             That is all.
0 1 2 3 4 5 6 7 8 9  0 0 0 0
```

3.11 EXEC

It is relative easy to load and start a complete program with the *EXEC O,"name.prg","cmd","env"* command.

It is somewhat harder if you wish to load a routine written in machine code or *C* only once and then execute it many times. While this is possible with the *EXEC O* command, it would require the routine to be loaded from the diskette or the *RAM*-Disk. Loading from the diskette is very slow and loading from a *RAM*-Disk requires twice the memory since the data must be copied to a *RAM*-Disk.

You could also transform the machine routine into *DATA* statements and then read that data with a corresponding *READ-POKE* loop (or *READ-A$=A$+Mki$()*). This uses a lot of memory; time to read the *DATA* statements and the program would not be relocatable. The advantage, however, is that only one program needs to be loaded.

You could also load the routine(s) with the *EXEC(3...)* command and then execute it with the *C:* command. The problem exists that these routines can not manage their own memory like normal programs can. It is also harder to recover the memory that was used for those routines.

```
'EXEC3
adr%=EXEC(3,"SCREEN.PRG","","")
' IF adr% AND            ! TOS was supposed to deliver a &FFFFFFd9
' ERROR adr%             ! (-39) on errors, but it really returns a
' ENIF                   ! &D9(217). Without an error check you may
screen%=adr%+256         !get 3 bombs (address error).
BLOAD "woof1.pi3",XBIOS(3)-34
HIDEM
REPEAT
  VOID C:screen%(2,1)
  VOID INP(2)
  VOID C:screen%(1,2)
  VOID INP(2)
UNTIL MOUSEK
SHOWM
VOID GEMDOS(73,L:HIMEM)
VOID GEMDOS(73,L:adr%)
```

This small program demonstrates all that is necessary to load a small machine code routine in an orderly manner, to execute that routine with parameters, and to release the memory after the program is finished.

The *EXEC(3...)* command loads the program, relocates it and delivers the address of the corresponding Basepage. Since this routine contains only one starting address, it can be computed by adding 256 to the Basepage. With many routines you could have built a table that consisted of a row of *JMP* commands or as shown here, by passing parameters through the stack.

The first of the two *GEMDOS(73)=m_free* calls returns the memory used for the environment string (at least two bytes), which is always located in the lowest possible address (*HIMEM*) and the second returns the actual memory of the program.

With larger routines you may have to reserve memory before you load the routines using the reserve command. If you wish to also use GEM routines (*RSC*-files,*Fileselect*) or more *EXEC(3...)* commands, you must reserve the necessary memory by using the *GEMDOS* routines *m-shrink*.

Example: A machine program *XXX.PRG* requires 20 Kbytes of memory. The Basic program requires 100 Kbytes for variables and strings and the rest of the memory may be used for *RSC*-files, etc.

```
RESERVE 100000                      !BASIC-memory usage
xxx.base%=EXEC(3,"XXX.PRG","","")
IF xxx.base%<BASEPAGE
  ALERT 1,"Unable to load XXX.PRG",1,"Cancel",dumm%
  END
ENDIF
e%=GEMDOS(74,0,L:xxx.base%,L:20000)
IF e%<0
  ERROR e%
ENDIF '
Here comes the rest of the program
'
VOID GEMDOS(73,L:HIMEM)
' or, if something else was loaded before the EXEC(3...)
' VOID GEMDOS(73,L:xxx.base%-2)
VOID GEMDOS(73,L:xxx.base%)
'
```

Attention: There is a serious bug with the current version of **TOS** ***when using the*** **m_alloc** ***and*** **m_free** ***calls that will cause the system to lock-up after issuing those commands about 20 times. Even the saving to diskette may not work anymore. It is very hard to duplicate this error since it seems to pop up whenever it is least wanted. You should never abort in the middle of a program that uses this kind of memory management.***

Here is an assembly program that will change the picture between the different resolutions of the *ST*, even though this book is supposed to be about BASIC.

```
* screen.asm
* Change screen after changing from resolution a to b
*
* VOID C:screen%(a,b)
*

        Section   Text          ;.text
start   move.w    #3,-(sp)
        trap      #14
        addq.l    #2,sp
        move.l    d0,a0         ;logbase
        move.w    4(sp),d0
        move.w    6(sp),d1
        beq       tolo
        subq.w    #1,d1
        beq.s     tomid
        subq.w    #1,d0
        bmi.s     lohi
        beq.s     mihi
        rts
lohi    move.l    a0,a3
        bsr.s     lomi
        move.l    a3,a0
mihi    move.w    #199,d0       ;200 linex
        mihi1     moveq #39,d1  ;40*2 Words
        move.l    a0,a1ds
mihi2   move.w    (a0)+,(a1)+
        move.w    (a0)+,-(sp)
        dbra      d1,mihi2
        moveq     #39,d1
        move.l    a0,a1
mihi3   move.w    (sp)+,-(a1)
        dbra      d1,mihi3
        dbra      d0,mihi1
        rts
lomi    move.w    #32000/8-1,d0
```

```
lomi1   move.l   (a0)+,d6
        move.l   (a0)+,d7
        moveq    #7,d1
lomi2   add.w    d6,d6
        addx.w   d3,d3
        add.w    d7,d7
        addx.w   d3,d3
        dbra     d1,lomi2
        moveq    #7,d1
lomi3   add.w    d6,d6
        addx.w   d5,d5
        add.w    d7,d7
        addx.w   d5,d5
        dbra     d1,lomi3
        swap     d6
        swap     d7
        moveq    #7,d1
lomi4   add.w    d6,d6
        addx.w   d2,d2
        add.w    d7,d7
        addx.w   d2,d2
        dbra     d1,lomi4
        moveq    #7,d1
lomi5   add.w    d6,d6
        addx.w   d4,d4
        add.w    d7,d7
        addx.w   d4,d4
        dbra     d1,lomi5
        movem.w  d2/d3/d4/d5,-8(a0)
        dbra     d0,lomi1
        rts
tomid   subq.w   #1,d0
        bmi.s    lomi
        beq.s    mimi
himi    move.w   #199,d0
himi1   moveq    #39,d1          ;40*2 Words
        lea      80(a0),a1
himi2   move.w   -(a1),-(sp)
        dbra     d1,himi2
        lea      80(a0),a1
```

```
            moveq      #39,d1
himi3       move.w     (sp)+,(a0)+
            move.w     (a1)+,(a0)+
            dbra       d1,himi3
            dbra       d0,himi1
            mimi       rts
tolo        subq.w     #1,d0
            beq.s      milo
            bmi.s      lolo
hilo        move.l     a0,a3
            bsr.s      himi
            move.l     a3,a0
milo        move.w     #32000/8-1,d0
milo1       movem.w    (a0),d2/d3/d4/d5
            moveq      #7,d1
milo2       add.w      d2,d2
            addx.w     d6,d6
            add.w      d2,d2
            addx.      w d7,d7
            dbra       d1,milo2
            moveq      #7,d1
milo3       add.w      d4,d4
            addx.w     d6,d6
            add.w      d4,d4
            addx.w     d7,d7
            dbra       d1,milo3
            swap       d6
            swap       d7
            moveq      #7,d1
milo4       add.w      d3,d3
            addx.w     d6,d6
            add.w      d3,d3
            addx.w     d7,d7
            dbra       d1,milo4
            moveq      #7,d1
milo5       add.w      d5,d5
            addx.w     d6,d6
            add.w      d5,d5
            addx.w     d7,d7
            dbra       d1,milo5
```

```
                move.l      d6,(a0)+
                move.l      d7,(a0)+
                dbra        d0,milo1
lolo            rts

                end
```

Two parameters are passed to this routine via the stack. The two parameters are the source and destination resolution in the format just like the *XBIOS(4)* function uses (0 for low resolution, 1 for medium resolution and 2 for high resolution).

Color is changed to the equivalent grey scale and vice versa. A color picture is thus converted to a black and white picture that should somewhat resemble the color picture. You could also look at a black and white picture on a color monitor, but you would then have to change the color register to the corresponding grey scale. Experimenting could be a lot of fun.

The previous program *(EXEC3)* loads *Woof* as a high resolution *Degas* picture from the diskette and changes it to a similar color picture in medium resolution.

Owners of monochrome monitors will most likely have some *Degas* color pictures stored on a diskette. With this routine you can look at those color pictures as black and white pictures.

The reason that the routine to change pictures was not written in **GFA BASIC** is because there is just too much bit manipulation. No high level language comes close to performing the routine as well as machine language. To change the pictures from low to medium resolution or vice versa requires 256,000 additions, besides loops and other things. The whole process only takes about half a second. You could write similar routines that could be treated as extensions to the **GFA BASIC** commands. When using *C* it is important that the routine is linked without the usual heading files. It is best to use compilers that create assembler source files. Some compilers may require you to save and restore all the registers.

3.12 Fonts

The *ST* computer is capable of mixing graphic and text on the same screen. Using the *DEFTEXT* you can modify the appearance of the text. But even this powerful command has its limitations when it comes to displaying exceptionally pretty lettering or very large characters. You could of course use the *GET/PUT* command to display the desired graphic, along with characters. This could, however, result in a lot of overhead.

But...

The *ST* comes with *GEM*, and *GEM* can (most of the time) create many different character sets. Unfortunately, the corresponding *VDI*-functions (*load_fonts...*) are not yet implemented on the current version of the *ATARI ST*, at least not to the fullest extent.

But...

You can bypass the corresponding functions (*vst_load_fonts* and *vst_unload_fonts* form *GDOS*) and still create *GEM* character sets (like a proportional character set).

The program *Fontdemo* demonstrates how this works.

The main program loads two character sets from the diskette after it reserved the necessary memory (a generous 100,000 bytes). The program then displays short text in the standard *ST* format and also in the two loaded fonts.

Next, those letters are shown in different sizes which tend to be somewhat slow without the blitter chip, but then, only a few large characters can fit on the screen anyway.

Procedure load_font(file$,adr.%)

This procedure is the real workhorse. After opening the files (with *LOF(#1)*), the memory is reserved by using the *GEMDOS*-function *malloc*. The files are then loaded into that memory using the *BGET* command. Should an error occur during the *GEMDOS* call (malloc=0), **GFA BASIC** will return the corresponding error message.

The beginning of the *FNT*-files contain a row of numbers as 2 or 4 byte integers. Unfortunately, these numbers are not in the *68000* processor format but rather in the Intel processor (*8080,8088, 8086, 80286*) format. A loop is generated to convert those numbers. However, not all of the data is converted since the actual data for the characters is already in the correct format.

First, the offset of the font data is determined (I assume that all numbers will come first and then the actual font data). The *DPOKE*-command converts the bytes.

The three following *LPOKE*-commands switch the low value word (just like Intel) with the high value word and add the starting addresses of the *Font-headers*, so that the correct pointer is stored in the memory for the *68000* (Since the offset of the pointer is always less than 65536, the high value word, which is null anyway, is dropped). For those that want the exact version can change the line to:

```
LPOKE a_%+68,a_%+DPEEK(a_%+68)+65536*DPEEK(a_%+70) etc.
```

Next, the pointer that ties all the character sets together is changed. The address of the *Font-header* is returned to the calling program.

PROCEDURE get_chrlink0

This procedure determines the address of the first *font-header* that is stored in the memory of the *ST*. The following small machine program serves that purpose:

```
.dc.w     $a000     ;a000
move.l    a1,d0     ;2009
rts       ;4e75
```

Here the *Line-A-init-call* is used to determine the address of the table that contains the addresses of the three internal character sets. The second of these character sets (standard 8*8 for color monitor) contains the pointer that points to the corresponding Font-Headers.

PROCEDURE get_chrlink

This procedure uses *get_chrlink0* and processes the whole list until the end is reached.

PROCEDURE unload_font(adr%)

This routine unloads the font at *Adr%* (in other words that font may no longer be used) by replacing the pointer of the previous character with the current character set. The memory that was used by that character set is returned to the system.

PROCEDURE kill_fonts

This routine serves as an emergency exit during the program development. This procedure erases all character sets that were loaded with **GFA BASIC**, independent of whether the corresponding pointers are known (like *ibmhss36%*) or not.

PROCEDURE unreserve

This procedure frees the memory that was reserved with the *RESERVE*-command.

The *RESERVE XBIOS(2)-16386-HIMEM+FREE(0)-nnnn* reserves *nnnn* bytes of memory. This long command allows you to test your program more often without eventually reserving all the available memory as would be the case with the *RESERVE FRE(0)-nnnn* command.

For a finished program it is usually better to use the *RESERVE aaaaaa* command since the memory usage is fixed and no more changes should take place.

> *Attention: If the program is compiled, you must reserve at least 32500 Bytes of memory for the* FILESELECT-*box. Therefore if an external program is selected with the* FILESELECT-*Box, you should issue a* RESERVE 1000 *call only after the* FILESELECT-*call was made. This is not necessary for the interpreter.*
>
> *DEFFN malloc(siz%)*
> *DEFFN mfree(adr%)*
> *DEFFN mshrink(adr%,size%)*

These are the *GEMDOS* functions that control memory usage.

@malloc(nnnn) reserves *nnnn* bytes of available memory (above *HIMEM*). The starting address of that memory is returned after calling this function. If no memory was available, a null is returned.

Special case: *@malloc(-1)* returns the number of bytes of the largest available memory block.

@mfree(aaaa) frees up the memory block at address *aaaa* and releases it to the operating system. An error has occurred if a negative value is returned (like *-40=ERROR - 40*).

@mshrink(aaaa,nnnn) allows you to decrease the size of an already allocated memory block. A negative number is returned if an error occurs.

> ***CAUTION: There is a bug in the operating system that crashes the computer after about* 20-40 @malloc *and* @mfree *cycles. This error will cause the computer to display "memory full" whenever a diskette command* (OPEN, SAVE...) *is issued. The only option available to you is to issue a* LLIST *command and then reset the computer.***

DEFTEXT color,style,rotation,height,face

You should already be familiar with the *DEFTEXT*-command, but only with four parameters. There is a fifth parameter (until now undocumented) that allows you to select the font. You must have called the corresponding *VDI*-call (*vst_font*). The number of the font (called face) is easily determined. This number is always found in the first two bytes of the font. The standard font contains the number 1. All other fonts that are loaded can contain any number and can be selected by you (like *DPOKE ibmhss36,2*, etc). If more than one version of the same font exists in memory, the one that matches the text height the closest (byte 2 and 3) is used.

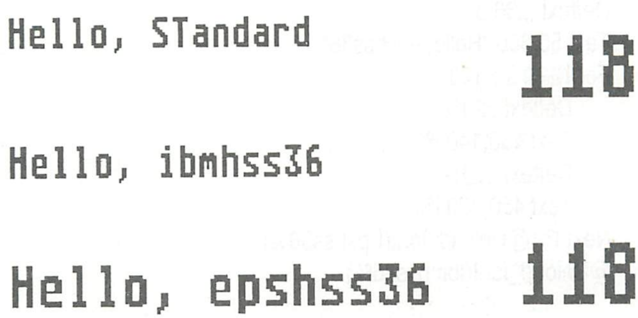
The Font-header:

Byte	Function
0-1	Font ID (face number)
2-3	Character size in points (1/72 inches)
4-35	Name of the font (8*16 Systemfont...)
36-37	First character in the font (often the character after code 32)
38-39	Last character in the font (usually not greater than 127). This is why no foreign characters can be displayed with the example fonts given.
40-41	Top line These are the distances
42-43	Ascent line between the letters
44-45	Half line from the baseline
46-47	Descent line
48-49	Bottom line
50-51	Width of the widest character
52-53	Width of the widest character cell incl. empty space
54-55	Left Offset for cursive text
56-57	Right Offset -"- -"-
58-59	Thickness width (4=extra wide)
60-61	Underline size (7=very thick line)
62-63	Mask for Light text (usually &5555)
64-65	Mask for skewed (italic) text (usually &5555)
66-67	Flags: bit 0=System Font bit 1=Uses horizontal offset table bit 2=Byte-swap-flag for font data Intel=0,Motorola(1) bit 3=Proportional(0)
68-71	Pointer to the Horizontal offset table
72-75	Pointer to the Character offset table
76-79	Pointer to the Font data
80-81	Total width of all characters in pixels
82-83	Height of the character matrix(must match 2-3)
84-87	Pointer to the next font or null

This is followed by the *Character-Offset-Table* which contains the number of pixels of all the preceding characters in the font.

There may also be a *Horizontal-Offset-Table* which contains the additional space required for each character. Finally, the actual character data follows which is stored in an extremely compact format so that each line can start on a word boundary. Compare to *BITBLT*.

Figure 12: Font Examples



```
'************************
'*    FONT DEMO     *
'************************
'
' FONTTEST '
Reserve Fre(0)-100000                           ! Place for Fonts '
@load_font("ibmhss36.fnt",*ibmhss36%)
@Load_font("GROSS.FNT",*Ibmhss36%)
Dpoke Ibmhss36%,2 '
@load_font("epshss36.fnt",*epshss36%)
@Load_font("GROSS.FNT",*Epshss36%)
Dpoke Epshss36%,3
Deftext ,,,36,1
Text 50,100,"Hello, STandard"
Deftext ,,,36,2
Text 50,200,"Hello, ibmhss36"
Deftext ,,,36,3
Text 50,300,"Hello, epshss36"
For I%=0 To 120
    Deftext ,,2,I%
    Text 450,140,I%
    Deftext ,,3,I%
    Text 450,300,I%
Next I% @Unload_font(Epshss36%)
@Unload_font(Ibmhss36%)
'
'''@kill_fonts
@Unreserve
'
Deffn Malloc(Size%)=Gemdos(&H48,L:Size%)
Deffn Mfree(Adr%)=Gemdos(&H49,L:Adr%)
Deffn Mshrink(Adr%,Size%)=Gemdos(&H4A,0,L:Adr%,L:Size%)
'
Procedure Load_font(File$,Adr.%)
    Local L_%,A_%,I_%
    Open "I",#1,File$
    A_%=@Malloc(Lof(#1))                       ! Place reserve
    If A_%<=0
        Error 101                              ! that was nothing
    Endif
```

```
    Bget #1,A_%,Lof(#1)                          ! Font load
    Close #1
    L_%=Peek(A_%+76)+255*Peek(A_%+77)            !L_% Bytes Font-Data
    For I_%=A_% To A_%+L_%-1 Step 2              ! Intel into Motorola
        Dpoke I_%,Peek(I_%)+256*Peek(I_%+1)      !Format calculating
    Next I_%
    Lpoke A_%+68,A_%+Dpeek(A_%+68)               ! Hor-Offs-Tab
        Lpoke A_%+72,A_%+Dpeek(A_%+72)           ! Chr-Offs-
    Tab Lpoke A_%+76,A_%+Dpeek(A_%+76)           ! Font-Data
    @Get_chrlink
    Lpoke Chrlink%+84,A_% *Adr.%=A_%
Return
'
Procedure Get_chrlink
    @Get_chrlink0
    Chrlink%=Chrlink0%
    While Lpeek(Chrlink%+84)
        Chrlink%=Lpeek(Chrlink%+84)
    Wend
Return
'
Procedure Get_chrlink0
    Local A_$,A_%
    A_$=Mkl$(&HA0002009)+Mki$(&H4E75)    ! A000 move.l a1,d0 rts
    A_%=Varptr(A_$)
    Chrlink0%=Lpeek(Lpeek(C:A_%()+4)+84)              ! see text
Return
'
Procedure Kill_fonts
    Do
        @Get_chrlink
        Exit If Chrlink%<Basepage
        @Unload_font(Chrlink%)
    Loop
    Deftext ,,,,1
Return
'
Procedure Unreserve
    Reserve Xbios(2)-16384-Himem+Fre(0)
Return
```

```
'
Procedure Unload_font(Adr%)
  @Get_chrlink0
  While Lpeek(Chrlink0%+84)<>Adr%
    Chrlink0%=Lpeek(Chrlink0%+84)
  Wend
  Lpoke Chrlink0%+84,Lpeek(Adr%+84)
  Void @Mfree(Adr%)
Return
```

CHAPTER 4

GEMDOS, BIOS and XBIOS

Many separate components, with different functions, make up the operating system of the *ST* computer.

Let us start from the top:

TOS (Tramiel Operating System)
The total operating system including *GEM*

GEM (Graphic Environment Manager)
A subsystem of the operating system that contains standardized graphics routines that can run independent of the machine. *GEM* can also be run on an *IBM* or other micro computers.

AES (Application Environment Services)
Responsible for the graphic input functions like the *Mouse-Menu-System.*

VDI (Virtual Device Interface)
Currently this is limited on the *ST* to the screen with few exceptions. Help routines for *AES* and programs, like drawing lines, fill areas, selecting line thickness, etc. The actual

drawing routines for the screen are implemented through the *Line-A-routines*. The *VDI* directs the drawing commands to the *Line-A-Routines* (or to a printer or diskette file)

GEMDOS (GEM Disk Operating System)

This is the actual operating system that was implemented on the *ST* computer. Similar to *CP/M* or *MS-DOS*, it is used for the orderly operation related to accessing disk drives. Through *GEMDOS* the data can only be accessed through the directory (not by sectors). *GEMDOS* controls all saving of disk files on the ST.

BIOS (Basic Input/Output System)
XBIOS (Extended Basic Input/Output System)

These two interfaces are used to control disk access by sectors and for accessing other peripherals, just like the *BIOS* for *CP/M* or *MS-DOS*. The *BIOS* performs all the normal I/O routines (Input/Output). The *XBIOS* allows one to access the enhanced services of the *ST* computer: screen addresses, colors, sound, hardware registers, Interrupt vectors, etc.

It is known that the *BIOS* calls *XBIOS* routines and *GEMDOS* uses *BIOS* and *XBIOS* routines. Eventually the *AES* will call all lower levels.

To use the *ST* efficiently, one must know all aspects of the operating system so that one can select the routines that are best suited. It does not have to be the best or the fastest and if you already have a routine that would be satisfactory then go ahead and use it. It would be senseless for you to write your own routine instead of using an existing **GFA BASIC** one if all that is saved is a few milliseconds.

For example: GFA BASIC uses the *BIOS* routines for the *PRINT* command because it is much faster then the one used by *GEMDOS*. There is also a problem with Control-C in *GEMDOS*. Direct Use of the *BIOS* often results in the loss of *GEMDOS* I/O redirection capabilities, but permits greater values.

In the following pages I will give a short description of the *GEMDOS*, *BIOS* and *XBIOS* routines and give an example where appropriate.

Errors are returned as a negative number and should match the error number of **GFA BASIC**.

4.1 GEMDOS

■ GEMDOS(0) p_termold

This routine ends the program. May not be used in **GFA BASIC.**

■ GEMDOS(1) c_conin

Reads a character from the console (keyboard). It is similar to the *INP(2)* function call. It returns a 32 bit word. In the lower eight bits (c% and 255) you will find the *ASCII* value for the pressed key (if an ASCII key was pressed). In bits 16 to 23 (c%/65536 and 255) you will find the *SCAN*-code of the keystroke. Every key has a code, even the function keys. Bits 24 to 31 contain the keyboard shift key (c%/&h1000000), just like *BIOS(11)* would return. For example *ALT-Left-Shift-A* would return the value &0A1E0041.

Some programs assume bits 24 to 31 are set to null. These bits are cleared (before *EXEC*) by

```
SPOKE &H484,PEEK(&484) OR &HF7
```

To set it to normal:

```
SPOKE &H484,PEEK(&H484) OR 8
```

The character that corresponds to the key pressed is displayed on the screen. Use of Control-C terminates the program (*crashes GEM programs*).

■ **GEMDOS(2,c%)** **c_conout**

Prints a character to the console (screen).

■ **GEMDOS(3)** **c_auxin**

Reads a character from the serial port.

■ **GEMDOS(4,c%)** **c_auxout**

Writes a character to the serial port.

■ **GEMDOS(5,c%)** **c_prnout**

Writes a character to the printer.

■ **GEMDOS(6,c%)** **c_rawio**

Writes a character to the console or if *c%=255* an *INKEY*-routine is executed. If a key is pressed a value will be returned, otherwise, a null is returned.

■ **GEMDOS(7)** **c_rawcin**

See **GEMDOS(8)**.

■ **GEMDOS(8)** **c_necin**

More key input routines. These two do not display the character and Control-C does not cause a break. It returns code as *GEMDOS(1)*.

■ GEMDOS(9,L:adr%) c_conws

Writes a *null-terminated* string to console:

```
a$="Hello"+chr$(0)
VOID GEMDOS(9,L:VARPTR(a$))
```

■ GEMDOS(10,L:adr%) c_conrs

```
@conrs(10,*a$)
PRINT a$
'
PROCEDURE conrs(n%,str. %)
    LOCAL a_$,a%
    a_$=CHR$(n%)+STRING$(n%+2,0)
    a%=GEMDOS(10,L:VARPTR(a_$))
    *str,%=MID$(a_$,3,ASC(MID$(a_$,2)))
RETURN
```

This routine reads an edited string. Because of the Control-C problem, it is almost impossible to use.

■ GEMDOS(11) c_conis

Returns null if no key was pressed

■ GEMDOS(14,d%) d_setdrv

Selects current drive like *CHDRIVE d%+1*.

■ GEMDOS(16) c_conos

Returns null if console is not ready to receive a character. This should never happen.

■ GEMDOS(17) c_prnos

Returns null if printer is not ready.

■ GEMDOS(18) c_auxis

Returns null if no character is available on the serial port.

■ GEMDOS(19) c_auxos

Returns null if serial port is not ready to receive a character.

■ GEMDOS(25) c_getdrv

Returns number of current drive *DEFFN gdrive=GEMDOS(25)+1*

■ GEMDOS(26L:adr%) f_setdta

Set buffer address for *f_sfirst* and *f_snext*. **GFA BASIC** sets this to *BASEPAGE*+128 at program start or when *DIR* or *FILES* command is used.

■ GEMDOS(42) t_getdate

Returns a 16 bit number containing the date (*DATE$*) in this format (Year-1980)*512+month*32+day.

■ **GEMDOS(43,d%)** **t_setdate**

Set the date (*SETTIME*) as above.

■ **GEMDOS(44)** **t_gettime**

Read time (*TIME$*). Returns a 16 bit number in this format Hour*2048+minute*32+seconds/2.

■ **GEMDOS(45,t%)** **t_settime**

Set time (*SETTIME*) as above.

■ **GEMDOS(47)** **f_getdta**

Returns buffer address for *f_sfirst,f_snext.*

■ **GEMDOS(48)** **s_version**

Returns current *GEMDOS* version number.

■ **GEMDOS(49,L:size%,ret%)** **p_termres**

Terminate program and reserve *size%* bytes in *BASEPAGE*. Cannot be used in **GFA BASIC**.

■ **GEMDOS(54,L:adr%,d%)** **d_free**

Returns information about free disk space on drive *d%* like the function *DFREE(d%).* The information is stored in a buffer that is four long words long starting at address *adr%*. You can obtain the space by multiplying the first long word with the third and the fourth (*DFREE*). The capacity by multiplying the second long word with the third and the fourth.

■ **GEMDOS(57,L:adr%)** **d_create**

Create a directory, *MKDIR* ...

■ **GEMDOS(58,L:adr%)** **d_delete**

Delete a directory, *RMDIR* ...

■ **GEMDOS(59,L:adr%)** **d_setpat**

Change directory, *CHDIR* ...

■ **GEMDOS(60,L:adr%,attr%)** **f_create**

Create a new file (Name starts at *adr%*). Returns a file handle that is used in any further operations. The same as *OPEN "O"*. If *attr%* is zero the file is normal, a 1 means file can only be read, a 2 means hidden file, a 4 is a system file and a 8 is the volume label (set while formatting the disk).

■ **GEMDOS(61,L:adr%,mode%)** **f_open**

Opens a file. *Mode%=0* corresponds to *OPEN "I"*, *mode%=1* corresponds to *OPEN "O"* and *mode%=2* corresponds to *OPEN "U"*. Returns information similar to *f_create*.

■ **GEMDOS(62,h%)** **f_close**

Closes file with handle *h%,* corresponds to *CLOSE#n.*

■ GEMDOS(63,h%,L:len%,L:adr%) f_read

Read *len%* bytes from the file that was opened with file handle *h%* into buffer at address *adr%*. Similar to *BGET #h,adr%,len%* and is also used by it and *INPUT*, etc.

■ GEMDOS(64,h%,L:len%,L:adr%) f_write

Writes *len%* bytes to *adr%* into file *h%*. Corresponds to *BPUT #h,adr%,len%*. Is used by the *PRINT* command, etc.

■ GEMDOS(65,L:adr%) f_delete

Deletes a file, *KILL* ...

■ GEMDOS(66,L:n%,h%,mode%) f_seek

mode%=0:	SEEK #h,n%
mode%=1:	RELSEEK #h,n%
mode%=2:	SEEK #h,-n%

■ GEMDOS(67,L:adr%,flg%,attr%) f_attrib

This routine reads or modifies the file attributes.

```
PROCEDURE chmod(file$,attr%)
    LOCAL e%
    file$=file$+CHR$(0)
    e%=GEMDOS(67,L:VARPTR(file$),1,attr%)
    IF e%<0
```

```
      ERROR e%
    ENDIF
RETURN
```

To protect a file: *@chmod*("*NAME.EXT*",1). By changing the file type to directory or the reverse, files can be well protected from unauthorized use.

A tip: Change the directory to a normal file and scramble the contents.

■ **GEMDOS(69,h%)** **f_dup**

■ **GEMDOS(70,n%,s%)** **f_force**

These routines allow you to reroute the input and output of the *GEMDOS-output_routines* (not usable for *GFA_BASIC*).

■ **GEMDOS(71,L:adr%,d%)** **d_getpath**

Corresponds to *DIR$(d%)*, Buffer starts at *adr%*

■ **GEMDOS(72,L:size%)** **m_malloc**

Reserves *size%* bytes of memory for the program. Returns the starting address, if *size%=-1* it returns the maximum available memory. This routine is known to have some bugs in the current *TOS*.

■ **GEMDOS(73,L:adr%)** **m_free**

Frees the memory starting at address *adr%* (*adr%* was the return from *m_malloc*).

■ **GEMDOS(74,0,L:adr%,L:size%)** **m_shrink**

Frees all memory starting at address *adr%* that exceeds *size%*.

■ **GEMDOS(75,f%,L:nam%,L:cmd%,L:env%)** **p_exec**

Executes program as subprogram from the diskette. Corresponds to *EXEC f%,nam$,cmd$,env$.*

■ **GEMDOS(76,ret%)** **p_term**

Terminates the program and passes *ret%* to parent program. *Cannot be used in* GFA BASIC.

■ **GEMDOS(78,L:nam%,attr%)** **f_sfirst**

■ **GEMDOS(79)** **f_snext**

These two routines are useful for searching through a directory. See program *SORTDIR.BAS* in chapter 3.6.

■ **GEMDOS(86,0,L:old%,L:neu%)** **f_rename**

Corresponds to *NAME* old$ AS *new$.*

■ **GEMDOS(87,L:tdbuf%,h%,flg%)** **f_datime**

With this routine you can change the date and time of a file. You must pass the file handle and the 4 byte address of a buffer (like **A%*) in which the date is stored. If *flg%=1* then write the date and if *flg%=0* get the date.

4.2 BIOS

■ **BIOS(0,L:ptr%)** **getmpb**

This routine determines how GEMDOS uses the memory, but without further knowledge of the operating system it is impossible to use.

■ **BIOS(1,d%)** **bconstat**

Similar to INP?(d%)

■ **BIOS(2,d%)** **bconin**

Corresponds to *INP(d%)*. It returns a long word just like *GEMDOS(1)*.

■ **BIOS(3,d%,c%)** **bconout**

Corresponds to *OUT d%,c%*.

■ **BIOS(4,f%,L:buf%,n%,rec%,d%)** **rwabs**

f%=0: Reads *n%* sectors starting at sector rec% on drive d% at buffer address *buf%*.
f%=1: Writes the sectors to the disk drive.
f%=2: Like 0, but ignores media-change.
f%=3: Like 1, but ignores media-change.

■ BIOS(5,n%,L:adr% setexec

Changes an exception vector of the 68000, *n%* is the number of the exception, adr% is the new value for the vector. A negative value returns the previous value.

■ BIOS(6) tickcal

Result is a 20, 20 ms *time-tick.*

■ BIOS(7,d%) getbpb

Returns the address of disk drive parameter block, only useful to monitor disk drives. Divided in 16 bit words: sector size (512), sector number per cluster (2), cluster size (1024), directory size, sector number of second FAT, sector number of first data-cluster, number of *data-clusters*, and flags. *d%* is the number of the drive.

■ BIOS(8,d%) bcostat

Corresponds to OUT?(d%)

■ BIOS(9,d%) mediach

Determines if diskette was changed.
0 = definitely was not changed (Harddisk)

1 = maybe it has been changed
2 = definitely was changed

■ BIOS(10) **drvmap**

Returns a bit pattern with a bit set for each drive that is attached. &x10011 says: drive A:,B: and E: are attached.

■ BIOS(11,x%) **kbshift**

Returns status of the shift key. By *x%=-1* the old status of the key is returned. If value is between 0 and 255, the corresponding key is simulated.

`&x........`	
`&x.......1`	Right shift key
`&x......1.`	Left shift key
`&x.....1..`	Control key
`&x....1...`	Alternate key
`&x...1....`	Caps-lock
`&x..1.....`	AlternateClr/Home
`&x.1......`	Alternate Insert

Simulate *Caps-Lock*:

```
On: VOID BIOS(11,BIOS(11,-1) OR &H10)
Off: VOID BIOS(11,BIOS(11,-1) AND &HEF)
```

4.3 XBIOS

■ **XBIOS(0,t%,L:par%,L:vec%)** **initmous**

This routine allows you to write your own mouse handler. It is not compatible with *GEM*.

■ **XBIOS(1,n%)** **ssbrk**

Reserve memory for the *ROM*-Module.

■ **XBIOS(2)** **physbase**

Get the screen's physical base address currently in use.

■ **XBIOS(3)** **logbase**

Get the screen's logical base address when drawing to the screen.

■ **XBIOS(4)** **getrez**

Returns the screen resolution: 0 = *Lores*, 1 = *Midres*, 2 = *Highres*, 3 = reserved for modified *ST*'s.

■ XBIOS(5,L:l%,L:p%,r%) setscreen

Makes it possible to change resolution (with the color monitor between lowres and highres). Unfortunately may not be used with *GEM*. The screen address may also be changed, separated by the physical and the logical address. See the chapter on flicker free graphics.

■ XBIOS(6,L:adr%) setpallete

This routine allows you to change all of the color registers at one time, as when loading a DEGAS picture:

```
BLOAD "DEGAS. PIx",XBIOS(3)-34,32034
VOID XBIOS(6,L:XBIOS(3)-32)
```

■ XBIOS(7,n%,c%) setcolor

This routine lets you change one color at a time. *SETCOLOR 3,&123* corresponds to *VOID XBIOS(7,3,&123)*. If *c%* is a negative value, the old color register is returned.

```
DEFFN getcolor(n%)=XBIOS(7,n%,-1) AND &777
```

■ XBIOS(8,L:a%,L:0,d%,s%,t%,si%,n%) floprd
■ XBIOS(9,L:a%,L:0,d%,s%,t%,si%,n%) flopwr
■ XBIOS(10,L:a%,L:0,d%,s%,t%, si%,i%,L:magic%,vir%) flopfmt

These routines control the floppy drives at the lowest level. The following values are used:

a% =	address buffer.
d% =	disk drive number (0/1).
s% =	sector number, with flopfmt it contains the number of sectors per track.
t% =	track number.
si% =	side (0/1).
n% =	Number of sectors to be read or written.
i% =	Interleave, determines the order of the sectors within the track, usually set to 1.
magic% =	is a constant that is used during formatting &H87654321.
virgin% =	determines what values the sectors will contain after a format command. It can be changed, however, as long as the high nibbles are not F. &HE5E5.

■ XBIOS(11) getdsb

Not used.

■ XBIOS(12,n%,L:a%) midiws

Writes a string of *n%* + *1* bytes starting at address *a%* to the *MIDI*-port.

■ XBIOS(13,n%,L:v%) mfpint

Set the *MFP* interrupt vector on the *ST*. May only be used with assembly or "*C*".

■ XBIOS(14,d%) iorec

Returns the address of the table that is used by the serial device.

```
XBIOS(14,0)       AUX:-Input
XBIOS(14,1)+14    AUX:-Output
XBIOS(14,1)       Keyboard buffer
XBIOS(14,2)       Midi-Buffer, only input
```

The table is as follows:

long word	Buffer address
word	Buffer size
word	head index
word	tail index The range between head and tail contains data. Buffer is empty if they are equal. If the buffer size is exceeded it will start at the beginning.
word	low water mark
word	high water mark If handshaking is active and the characters in the buffer reaches the high water mark, the computer will send a signal to the sender to stop sending data until the low water mark is reached. Normally: 1/4 to 3/4 of the buffer size.

To erase the keyboard buffer:

```
LPOKE XBIOS(14,1)+6,0
```

To erase the serial output buffer:

```
LPOKE XBIOS(14,0)+14+6,0
```

To enlarge the buffer for *MIDI*:

```
midipar%=XBIOS(14,2)
oldmidibuf%=LPEEK(midipar%)
oldmidisize%=DPEEK(midipar%+4)
'
DIM temp%(20000/4)
SLPOKE midipar%,VARPTR(temp%(0))
SDPOKE midipar%+4,20000
SLPOKE midpar%+6,0
'
' Now we have time to do INP(3)
'
SDPOKE midpar%+4,oldmidisize%
SLPOKE midoar%+6,0
SLPOKE midipar%,oldmidibuf%
SLPOKE midipar%+6,0
ERASE temp%()
```

If the buffer for the serial port is changed then you should also change the low and high water marks.

■ XBIOS(15,b%,f%,u%,r%,t%,s%) rsconf

Configure the serial port. By -1 the parameters are not changed.

*b%=baudrate

0=19200	1=9600	2=4800	3=3600	4=2400	5=2000
6=1800	7=1200	8=600	9=300	10=200	11=150
12=134	13=110	14=75	15=50		

f%=handshake mode

0=none, 1=XON/XOFF, 2=RTS/CTS, 3=BOTH??

u%=MFP-registers in binary format

`&x.....0..`	No parity
`&x.....10.`	Odd parity
`&x.....11.`	Even parity
`&x...01...`	1 stop bit
`&x...10...`	1.5 stop bits
`&x...11...`	2 stop bits
`&x.00.....`	8 data bits
`&x.01.....`	7 data bits
`&x.10.....`	6 data bits
`&x.11.....`	5 data bits
`&x0..00...`	Synchronized,frequency form TC/RC
`&x1..00...`	Synchronized, divided by 16

r%,t%,s%=MFP-registers rsr,trs,scr

A complete description of these binary registers would take up too much space and is seldom used. Normally just set these parameters to -1.

■ XBIOS(16,L:u%,L:s%,L:c%) keytbl

With this routine you can change the keyboard translation tables. It consists of three tables, each with 128 bytes. The keys are converted to the ASCII-Code as follows: u%=unshifted, s%=shifted and c%=caps-lock. A parameter of -1 means not to change the address. The following is an example of how to change the keys of the numbers block to the Greek alphabet.

```
' keytab
Void Xbios(24)                                  !bioskeys
O%=Xbios(16,L:-1,L:-1,L:-1)                     !get pointers
Dim K%(128*3/4)                                 !buffer
K%=Varptr(K%(0))
Bmove Lpeek(O%),K%,128
Bmove Lpeek(O%+4),K%+128,128
```

```
Bmove Lpeek(O%+8),K%+256,128
For I%=0 To 14
    Poke K%+&H63+I%,224+I%                    !Greek Text
    Poke K%+&HE3+I%,239+I%  !more when pressing the shift key
    Poke K%+&H163+I%,128+I%     !Caps: a few international
Next I%
Void Xbios(16,L:K%,L:K%+128,L:K%+256)
Repeat
    Out 5,Bios(2,2)
Until Mousek
Void Xbios(24)
```

Use *XBIOS(24)* to return the keys to normal. An address pointing to the three tables is also returned by *XBIOS(16)*. Field *k%* serves only to store the key. The three Bmoves copy the original table which is then changed for all three conditions. The program then performs a loop that allows you to enter keys until the mouse button is pressed. The *XBIOS(24)* at the end is very important since you would not be able to use the keyboard properly without it. You may want to put in a *STOP* after the first *XBIOS(24)* during program development so that you can run the program with the mouse to return you to normal keys.

■ XBIOS(17) random

Returns random number from 0 to 16777215,24 bits.

■ XBIOS(18,L:a%,L:0,L:s%,t%,f%) protobt

This routine creates a boot sector for the diskette in memory, *a%* points to a 512 byte buffer, *s%* is a serial number that is written as part of the boot sector. If the number is greater than 24 bits a random number is created. Where -1 is the serial unchanged, and *t%* is the disktype:

0	=single sided, 40 tracks (180K)
1	=double sided, 40 tracks (360K) IBM
2	=single sided, 80 tracks (360K) SF 340
3	=double sided, 80 tracks (720K) SF 314
-1	=disktype is unchanged
f%	=0 diskette does not have TOS
f%	=1 diskette contains TOS
f%	=-1 unchanged

■ XBIOS(19,L:a%,L:0,d%,s%,t%,si%,n%) flopver

Verifies storage of the floppy disk. If the value is null then everything checks out OK. If there is an error, you can find the sectors that were bad starting at address *a%*, just like flopfmt.

■ XBIOS(20) scrdmp

Dump screen to printer, just like *HARDCOPY*.

■ XBIOS(21,a%,r%) curscon

Allows you to configure the cursor of the operating system.

f%=0	hide cursor
f%=1	show cursor
f%=2	blinking cursor
f%=3	solid cursor
f%=4	set blink rate
f%=5	return current blink rate

if *f%=4* then *r%* contains the blink rate of the screen (50hz, 60hz for color or 71hz for monochrome).

■ **XBIOS(22,L:dt%)** **bsettime**
■ **XBIOS(23)** **bgettime**

These functions correspond to the *GEMDOS*-routines *SGET/GET* time/date. The date is multiplied by 65536 and then added to the time.

■ **XBIOS(24)** **bioskeys**

see *XBIOS(16)=keytbl*

■ **XBIOS(25,n%,L:a%)** **ikbdws**

Writes *n%-1* bytes from address *a%* to the keyboard processor.

■ **XBIOS(26,n%)** **jdisint**

Disable interrupt number *n%(0-15)* of the *MFP*.

■ **XBIOS(27,n%)** **jenabin**

Enables interrupt *n%* of the *MFP*.

■ **XBIOS(28,c%,n%)** **giacces**

n%=&00..&0F reads the sound register *n%*
n%=&80..&8F writes c% to register *n%*

■ **XBIOS(29,m%)** **offgibit**

■ XBIOS(30,m%) ongibit

Sets the bit of port A on the sound chip to zero or one. With *ONGIBIT* the bit pattern is *OR*ed with the current value, by *OFFGIBIT* the pattern is made with *AND*.

m%= 1:	Select floppy side 0 or side 1
m%= 2:	Floppy A on/off
m%= 4:	Floppy B on/off
m%= 8:	RTS on/off
m%=16:	DTR on/off
m%=32:	Centronics strobe on/off
m%=64:	GPO on/off (a pin in the connector of the moniter (13 Pins))

Example:

```
VOID XBIOS(29,NOT 2)
PRINT "Floppy A: is on"
PAUSE 100
VOID XBIOS(30,2)
PRINT "Floppy A: is off"
```

■ XBIOS(31,n%,c%,d%. L:vec%) xbtimer

Change Timer Nr. *n%* (0=A, 1=B, 2=C, 3=D) of the *MFP*. *c%* and *d%* are written to the Control and Data registers, *vec%* is the pointer to the corresponding interrupt vector.

Example: match the baud rate with timer *D*:

```
' Baud rate calculation
Dim A%(7)
For I%=1 To 7
    Read A%(I%)
Next I%
Data 4,10,16,50,64,100,200
```

```
F$="# #### ######.######## #### ######.#######"
Do
    Input "Baud rate ",A
    For I%=1 To 7
        B=19200*4/A%(I%)/A
        B1=Int(B)
        B2=Int(B+1)
        B1=Max(B1,1)
        B2=Max(B2,1)
    Print Using F$,I%,B1,19200*4/A%
                (I%)/B1,B2,19200*4/A%(I%)/B2
    Next I%
Loop
' Select the Baud rate:
' VOID XBIOS(31,3,i%,b1,l:-1)
' b1>0 und b1<256
```

Caution: These baud rates are real rates for the ST, but when using XBIOS(15..) (rsconf) it does not set 50 or 75 baud, but 80 or 120 instead. A small program follows that uses rsconf (XBIOS(15..)) and then displays the real baud rates.

```
' Baudtest.bas
'
Dim A%(7)
For I%=1 To 7
    Read A%(I%)
Next I%
Data 4,10,16,50,64,100,200
Print "Index","Timer D","Control Data","Result"
For I%=0 To 15
    Void Xbios(15,I%,-1,-1,-1,-1,-1,-1)
    D%=Peek(&HFFFA1D) And 7
    Q%=0
    For J%=1 To 500
    Q%=Max(Q%,Peek(&HFFFA25))
    Next J%
    Print I%,D%,Q%,
```

```
Print 19200*4/A%(D%)/Q%
Next I%
```

The result will be as follows:

Index	Timer D	Control Data	Result
0	1	1	19200
1	1	2	9600
2	1	4	4800
3	1	5	3840
4	1	8	2400
5	1	10	1920
6	1	11	1745.45
7	1	16	1200
8	1	32	600
9	1	64	300
10	1	96	250
11	1	128	150
12	1	143	134.26
13	1	175	109.71
14	2	64	120
15	2	96	80

The last two lines could also be:

	3	64	75
	3	96	50

This is an error in the operating system that will most likely *never* be corrected. For 50 Baud use:

```
VOID XBIOS(31,3,3,64,L:-1)
```

■ XBIOS(32,L:adr%) dosound

This routine allows you to play music independent of the program. The *SOUND*-buffer starting at *adr%* contains the music in form of control bytes.

The end of this chapter contains a program named *Elise* that contains these control bytes. The files that are created by this program may then be read by other programs and played back by using this interrupt.

Used control bytes:

00 yz :	Low-Byte duration sound channel 1
01 0z :	High-Byte duration sound channel 1 as by Sound ... #&xyz
02 yz 03 0x :	same for channel 2
04 yz 05 0x :	same for channel 3
06 ff :	frequency of the wave generators (0..63)
07 xx :	selects the sound channel like Wave, but xx inverted Wave 1 corresponds to 07 FE (NOT 1=&FF) Wave &1009 corresponds to 06 10 07 F6
08 11 :	volume channel 1
09 11 :	volume channel 2
0A 11 :	volume channel 3
0B xy 0C zt :	duration of envelope curve
0D 0h :	envelope curve form Wave ?,h,?,&ztxy
80 xx :	loads xx into a temporary register
81 0r ss ee ww :	loads the register *r* with the value taken from partition (80 xx). Increases it after *ww/50* seconds by *ss*. When it reaches the final value *ee* it stops.

Caution: The interrupt routine uses 4 values, but the counter is only increased by 3 after completion. Therefore, you must follow it with the

value ww (01..0D) to be able to get a meaningful sound routine.

82 xx to FF xx :	Waits xx/50 seconds. Terminates if xx is equal to zero.

■ XBIOS(33,m%) setprt

This routine allows you to configure the printer just like you would with the accessory.

`&x.....?`	0=matrix, 1=Daisy Wheel
`&x....?.`	0=Color, 1=black and white
`&x...?..`	0=1280, 1=960 dots per line
`&x..?...`	0=Draft,1=NLQ
`&x.?....`	0=parallel,1=serial
`&x?.....`	0=continuous, 1=single page
`&x000110`	Normal configuration for Epson compatible printers. (VOID XBIOS(33,6)).

Negative values return the old configuration.

■ XBIOS(34) kbdvbas

This routine returns the address to a table that contains the pointer of the interrupt vectors for communication with the keyboard processor (and midi).

The possible vectors:

midivec	;MIDI input (d0)
vkbderr	;Keyboard error
vmiderr	;MIDI error
statvec	;Keyboard status-packet
mousevec	;Mouse-packet (-->GEM)
clockvec	;Clock-packet

```
joyvec          ;Joystick-packet
```

The *A0* register of the processor points to the input data. The *joyvec* vector may be of great interest to BASIC programmers.

The following program uses this vector to get the joystick values. The string *mc$* contains the interrupt routine which is only used once and stores the address of the joystick from the register *A0* to the variable *a%* (The program is: *move.l a0, *a% rts*). This address is later used to *PEEK* the joystick values.

First, the address of the interrupt vector table is located by using *XBIOS(34)*. Then the old vector is routed to your own routine. The *OUT 4,&16* allows you to get the keyboard processor to read and send the values for both joysticks. As soon as the data is present (when *a%* does not equal null anymore), the old routine is restored. *a%* now points to the data that contains the joystick number (254 for joystick 1 and 255 for joystick 2). The bytes following these are the data. The addresses are stored in the variables *joy0%* and *joy1%*. The *OUT 4,&14* puts the keyboard processor into the joystick mode. The mouse cannot be read anymore, but the joysticks will now automatically return values.

It is also possible to use the mouse by issuing an *OUT 4,&16* before every joystick request (Port 1=mouse,Port 2=joystick). This is strongly recommended during the testing phase since an error would otherwise require you to manually type an *OUT 4,8*. The *OUT 4,8* returns the keyboard processor to mouse mode. The joystick values are in bit format. You may look at the program to determine the bit pattern.

```
' joystick.bas
Mc$=Mki$(&H23C8+Mkl$(*A%)+Mki$(&H4E75)
```

```
V%=Xbios(34)+24
O%=Lpeek(V%)
Lpoke V%,Varptr(Mc$)
A%=0
Out 4,&H16
Repeat                                        ! Wait for Interrupt
Until A%
Lpoke V%,O%
Joy_0%=A%+1
Joy_1%=A%+2
Out 4,&H14
Print At(1,20);"Press any key to quit";
Repeat
  Print At(1,9);"Last: Joystick ";(Peek(A%) And 1)+1
  @Output(Peek(Joy_0%))
  @Output(Peek(Joy_1%))
Until Inkey$<>""
Out 4,8
Procedure Output(X%)
  If X% And 128
  Print "Button ";
    Endif
  If X% And 1
    Print "Up ";
  Endif
  If X% And 2
    Print "Down ";
  Endif
  If X% And 4
    Print "Left ";
  Endif
  If X% And 8
    Print "Right ";
  Endif
  Print Chr$(27);"K"
Return
```

Chr$(27);"K" erases from the cursor to the end of the line. There is a table at the end of the chapter that explains the Escape sequences you may use for screen display without a window (*VT52*).

■ XBIOS(35,d%,r%) **kbrate**

This routine sets the repeat delay (*d%*) and the repeat rate (*r%*). It returns the old key repeat values (*d%*256+r%*). *d%=0* turns the repeat rate off. As usual a negative value does not change the parameters.

■ XBIOS(36,L:pointer) **prtblk**

This routine is a subprogram of the Hardcopy-routine and points to an address that contains all sorts of parameters.

■ XBIOS(37) **vsync**

Corresponds to *VSYNC*

■ XBIOS(38,L:vec%) **superx**

Executes a machine language routine at address *vec%* in supervisory mode without using *GEMDOS*.

■ XBIOS(39) **pntaes**

Turns off *AES* if it is not in *ROM* (reboots).

4.4 ELISE

```
' ELISE
'
@Init
M$=""                                   ! stores sound string
Oct%=4                                  ! default
Dur%=10
L%=10
Do
    Read A$                             ! Read data
    Exit If A$=""
    While A$<>""                        ! executing more than one string
        B$=Upper$(Left$(A$))            ! by passing the string after use
        While B$="."                    ! a$=MID$...
            M$=M$+Mkl$(&H100)+Chr$(-1)+Chr$(1)
            A$=Mid$(A$,2)
            B$=Upper$(Left$(A$))
        Wend
        A$=Mid$(A$,2)
        On Instr("CDEFGABHPOXL+-WR",B$) Gosub
                        C,D,E,F,G,A,B,B,P,O,X,L,Pl,Mi,Wave,R
            ! This line was split because of lack of space
    Wend
Loop
M$=Chr$(7)+Chr$(-2)+M$+Mki$(&HFF00)             ! Tone end
Void Fre(0)
Void Xbios(32,L:Varptr(M$))                     ! play tones
```

```
Print "Write file"
Fileselect "\*.SND",".SND",A$
If Len(A$)
    Bsave A$,Varptr(M$),Len(M$)                    ! save
Endif
Data o5l15
' DATA l16w+10. 1000
Data ed#,ed#e-h+dc,-aaacea
Data hhheg#h,+cccp
Data ed#,ed#e-h+dc,-aaacea
Data hhhd+c-h,aaph
Data +cd,eee-g+fe,ddd-f+ed
Data ccc-e+dc,-hhpe+
Data ed#,ed#e-h+dc,-aaacea
Data hhhd+c-h,aaap
Data
Procedure C                ! for single notes the note number is
    @Note(1)               ! passed to the procedure
Return
Procedure D
    @Note(3)
Return
Procedure E
    @Note(5)
Return
Procedure F
    @Note(6)
Return
Procedure G
    @Note(8)
Return
Procedure A
    @Note(10)
Return
Procedure B                          ! H=B
    @Note(12)
Return
Procedure P                          ! Pause
    D%=Val(A$)                       ! with parameter and without
    If D%=0
```

```
        D%=Dur%
    Endif
    M$=M$+Mki$(&H800)+Chr$(-1)+Chr$(D%)+Chr$(8)+Chr$(L%)
    A$=Mid$(A$,Val?(A$)+1)
Return
Procedure X                            ! Pause without turning tone off
    D%=Val(A$)
    A$=Mid$(A$,Val?(A$)+1)
    If D%
        M$=M$+Chr$(-1)+Chr$(D%)
    Else
        M$=M$+Chr$(-1)+Chr$(Dur%)
    Endif
Return
Procedure O                            ! change only the octave
    Oct%=Val(A$)
    A$=Mid$(A$,Val?(A$)+1)
Return
Procedure Pl                           ! increase the octave
    Inc Oct%
Return
Procedure Mi                           ! decrease the octave
    Dec Oct%
Return
Procedure Note(N%)                     ! subroutine for note
    If Left$(A$)="#"                   ! # increases note
        A$=Mid$(A$,2)                  ! also e#...
        Inc N%
        D%=Val(A$)                     ! change note duration
    Else
        D%=Val(A$)
    Endif
    A$=Mid$(A$,Val?(A$)+1)
    If D%
        Dur%=D%
    Else
        D%=Dur%
    Endif
    ' frq%=125000/(2^oct%*440*(2^(n%/12))/(2^(10/12))/16)+0.5
    M$=M$+Mkl$(Frq%(Oct%,N%))+Chr$(-1)+Chr$(D%)
```

```
        If Wav!
            M$=M$+W$
        Endif
    Return
    Procedure L                     ! volume
        L%=Val(A$)
        M$=M$+Chr$(8)+Chr$(L%)
        A$=Mid$(A$,Val?(A$)+1)
    Return
    Procedure Wave                  ! envelope curve
        Out 2,7
        If Left$(A$)="+"            ! turn wave on
            Wav!=True
            A$=Mid$(A$,2)
        Endif
        If Left$(A$)="-"            ! disable wave
            Wav!=False
            A$=Mid$(A$,2)
        Endif
        If Val?(A$)                 ! When parameter:
            Huell%=Val(A$)          ! set both
            Per%=Val(Mid$(A$,Instr(A$,".  ")+1))
            W$=Chr$(13)+Chr$(Huell%)+Chr$(11)+Chr$(Per%)+Chr$(12)+
                                                Chr$(Per% Div 256)
            M$=M$+W$
            A$=Mid$(A$,Val?(A$)+1)
        Endif
    Return
    Procedure R                     ! noise
        If Left$(A$)="+"            ! enable
            M$=M$+Mki$(&H7F6)
            A$=Mid$(A$,2)
        Endif
        If Left$(A$)="-"            ! disable
            M$=M$+Mki$(&H7FE)
            A$=Mid$(A$,2)
        Endif
        If Val(A$)                  ! change period
            M$=M$+Chr$(6)+Chr$(Val(A$))
            A$=Mid$(A$,Val?(A$)+1)
```

```
    Endif
Return
Procedure Init                          ! Taken from notes of
    Dim Frq%(12,12)                     ! my Physics class
    For N%=0 To 12
        For O%=0 To 12
            F%=125000/(2^O%*440*(2^(N%/12))/(2^(10/12))/16)+0.5
            Frq%(O%,N%)=(F% And 255)*65536+(F% Div 256)+&H100
        Next O%                         ! a=440 hz
    Next N%                             ! 12 notes per octave
Return                                  ! 1 octave=frequency doubler
```

Note ?? stands for value (0 15 1000 &38):

c d e f g a h b c =	Notes by adding a # (sharp)the note is made higher. Optional tone length in 1/50 second is used by all following notes
o??=	Chooses octave
+ =	Increases octave by one
- =	Decreases octave by one
l?? =	Selects volume (0..15, 16 means with envelope curve).
r =	Set Noise
r+ =	On
r- =	Off
r?? =	Selects noise frequency 0..31 also r+?? and r-??
w??.???? =	Set the envelope. Before the "." sets the form and after the "." selects the period. After a w+ or a w+??.???? the envelope is reset after every note. w- or w-??.???? turns this mode off.
p =	Pause. The tone generation stops but unfortunately some noise continues with Wave.
x =	Pause without turning off noise generator.

Who could possibly think of more? Programs that use the *SOUND* and *WAVE* commands or that use the *Dosound*-routine (*XBIOS(32)*) are often disturbed by the keyclick. With the following command it may be disabled:

```
SPOKE &H484,PEEK(&H484) AND NOT 1   ! Keyclick on
SPOKE &H484,PPEK(&H484) OR 1        ! Keyclick off
SPOKE &H484,PEEK(&H484) AND NOT 4   ! Control-G CHR$(7) Bell off
SPOKE &H484,PEEK(&H484) OR 4        ! Bell on
```

The key repeat may be disabled by *AND NOT 2* and enabled by *OR 2*.

4.5 VT 52-Emulator

The *ST* contains a *VT-52*-emulator, which was fashioned after a popular terminal. It may be used for screens that do not use windows.

All the sequences begin with the *ESC* code (*CHR$(27)*).

ESC A:	Cursor moves up one line. It stops at top of screen.
ESC B:	Cursor moves down one line. It stops at bottom.
ESC C:	Cursor moves to the right. It stops at right corner.
ESC D:	Cursor moves to the left. It stops at left corner.
ESC E:	CLS (Clear screen).
ESC H:	Cursor Home (PRINT AT(1,1)).
ESC I:	Cursor moves up one line., scrolls on top.
ESC J:	Erases from cursor to the end of page.
ESC K:	Erases from cursor to the end of the line.
ESC L:	Insert a line.
ESC M:	Erases a line,moves following lines up one line.
ESC Y s z:	Print AT(row,column); s=chr$(row+32) z= chr$(column+32)
ESC b n:	Selects the color for the text, n=chr$(color). By high resolution only AND 1 is used, by

	medium AND 3 and by low resolution AND 15.
ESC c n:	like b, except background color.
ESC d:	Erase from top of page to cursor.
ESC e:	Enable cursor.
ESC f:	Disable cursor.
ESC j:	Save cursor position.
ESC k:	Restores cursor that was saved with ESC j.
ESC l:	Erase line.
ESC o:	Erase line from beginning to cursor.
ESC p:	Select reverse video.
ESC q:	Turns reverse video off.
ESC v:	Wrap at end of line.
ESC w:	Truncate at end of line.

Chapter 5

AES

Not only are there many different routines in *GEMDOS*, *BIOS* and *XBIOS*, but also in *GEM* itself.

Most *VDI* routines exist as **GFA BASIC** commands (*CIRCLE*, *BOX*, *BITBLT*, etc.). Some important routines that are not present in BASIC like *load_font*, *open_work* are not easy to use on the *ST*. These routines were discussed in the chapter on Graphics and Fonts.

I have, for the most part, omitted the parameter value returned since this value will usually be something other then null unless an error was found *DPEEK(GINTOUT)*. If you want, you may add the *@gemerr* call to all routines that have the *?E* table below.

```
PROCEDURE gemerr
  IF DPEEK(GINTOUT)=0
    ERROR 77
  ENDIF
RETURN
```

NOTICE: In case I decide to include some of these AES-routines in **GFA BASIC** *Version 3.0, I will use error numbers between 70 and 79. Some routines like wind_get return many different values. It is faster to use DPEEK(GINTOUT+8) instead* *of returning the value through a pointer. This is especially true for the compiler.*

I mark all variables that are used as pointers by attaching a *".%"*, local variables in the same procedure by attaching a "*_%*" or a "*_$*" or a "*_!*". Unfortunately you may not use a pointer to a global variable in **GFA BASIC** if a local variable with the same name exists. The reason for this is simple: To be able to use the *GOTO* command to exit into another procedure, the local variables must always be found at the same location.

In the following example, *Tree%* indicates that the variable is the address of an object tree. This is the structure contained in *RSC*-files and is automatically created with the *MENU m$()* command. Further explanation may be found in the chapter on Resource.

Let us move on to the *AES*-calls. These routines are represented in decimal number order, *1x* stands for *appl_xxx*, *2x* stands for *evnt_xxx*, etc.

### Name	GINTOUT
10 appl_init	ap_id
11 appl_read	?E
12 appl_write	?E
13 appl_find	ap_id/-1
14 appl_tplay	1
15 appl_trecord	quantity
19 appl_exit	?E
20 evnt_keybd	Key
21 evnt_button	clicks x y button shift
22 evnt_mouse	1 x y button shift
23 evnt_message	1
24 evnt_timer	1
25 evnt_multi	......
26 evnt_dclisk	speed
30 menu_bar	?E
31 menu_icheck	?E

32 menu_ienable	?E
33 menu_tnormal	?E
34 menu_text	?E
35 menu_register	0-5/-1
40 objc_add	?E
41 objc_delete	?E
42 objc_draw	?E
43 objc_find	index/-1
44 objc_offset	?E x y
45 objc_order	?E
46 objc_edit	?E pos
47 objc_chnge	?E
50 form_do	exit_obj
51 form_dial	?E
52 forn_alert	exit_but
53 form_error	1
54 form_center	. x y w h
70 graf_ruberbox	?E w h
71 graf_dragbox	?E x y
72 graf_movebox	?E
73 graf_growbox	?E
74 graf_shrinkbox	?E
75 graf_watchbox	0/1
76 graf_slidebox	0-1000
77 graf_handle	handle wc hc wb hb
78 graf_mouse	?E
79 graf_mkstate	. x y but shift
80 scrp_read	?E
81 scrp_write	?E
90 fsel_input	?E 0/1
100 wind_create	handle/-x
101 wind_open	?E
102 wind_close	?E
103 wind_delete	?E
104 wind_get	?E
105 wind_set	?E
106 wind_find	handle
107 wind_update	?E
108 wind_calc	?E x y w h
110 rsrc_load	?E

111 rsrc_free	?E
112 rsrc_gaddr	?E (addrout)
113 rsrc_saddr	?E
114 rsrc_obfix	?
120 shel_read	?E
121 shel_write	?E
122 shel_get	?E
123 shel_put	?E
124 shel_find	?
125 shel_envrn	?

? reserved/undefined
?E 0=error, otherwise OK
...... many values
. meaning changes

5.1 APPLication Library

The *appl_xxx* routines allow you to have more than one program or application in memory at one time. They are usually used by *GEM* for accessories, but would be even more useful if a multi-tasking version of *GEM* ever appears.

```
PROCEDURE appl_init
    GEMSYS 10
RETURN


PROCEDURE appl_read(id%,len%,buf%)
    DPOKE GINTIN, id%
    DPOKE GINTIN+2,len%
    LPOKE ADDRIN,buf%
    GEMSYS 11
RETURN
PROCEDURE appl_write(id%,len%,buf%)
    DPOKE GINTIN,id%
    DPOKE GINTIN+2,len%
    LPOKE ADDRIN,buf%
    GEMSYS 12
RETURN
```

These two routines allow you to pass messages between several resident *GEM*-applications. The message starts at address *buf%* and is *len%* bytes long. The destination (*appl_write*) or source (*appl_read*) is always the *GEM*-internal message buffer of the application *id%*.

```
PROCEDURE appl_find(name$)
    nam$=nam$+CHR$(0)
    LPOKE ADDRIN,VARPTR(nam$)
    GEMSYS 13
RETURN
```

This routine finds the simultaneous running application with the name of *nam$* and then returns the corresponding *ap_id* or -1 using *GINTOUT*.

```
PROCEDURE appl_tplay(adr%,num%,scale%)
    LPOKE ADDRIN,adr%
    DPOKE GINTIN,num%
    DPOKE GINTIN,scale%
    GEMSYS 14
RETURN
PROCEDURE appl_trecord(adr%,num%)
    DPOKE GINTIN,num%
    LPOKE ADDRIN,adr%
    GEMSYS 15
RETURN
```

These two routines act like a software recorder. A number (*num%*) of events (mouse, timer, keyboard and button) are written to a buffer (at *adr%*) with *TRECORD* which may then be replayed with the *TPLAY*. When replaying you may also apply a sliding scale between 1-1000 that determines the speed at which the user actions are played back. Unfortunately, this routine does not work as described in the *GEM* documentation and, in any case, I cannot determine any practical use of this routine.

```
PROCEDURE appl_exit
    GEMSYS 19
RETURN
```

This routine must always be called before exiting a *GEM* program. **GFA BASIC** automatically calls this routine before exiting.

5.2 *EVENT Library*

The *event_xxx* routines cause the program to wait for an external event (like the user pressing a key). They also supply the limited *multi-tasking* capabilities of *GEM*. Unfortunately, routines like *evnt_fileopen* or *evnt_diskwrite* are missing (the corresponding *BIOS* call would even be better). Even so, these routines make it possible for other programs (accessories) to run in the background without greatly affecting the performance of the main program.

```
PROCEDURE evnt_keybd
  GEMSYS 20
RETURN
```

This is a simple keyboard input routine that still allows the use of accessories. Use *PEEK(GINTOUT+1)* to determine the *ASCII* value of the pressed key and *PEEK(GINTOUT)* to determine the scan code (similar to bconin, etc. in BIOS). *DPEEK(GINTOUT)* will return the combination of those two values.

```
PROCEDURE evnt_button(clicks%,mask%,state%)
  DPOKE GINTIN,clicks%
  DPOKE GINTIN+2,mask%
  DPOKE GINTIN+4,state%
  GEMSYS 21
RETURN
```

If you would like to wait until the user presses a certain mouse button (like double clicking on the right mouse button), you can use the above routine. *Click%* is the maximum number of mouse clicks to wait (usually 2). With *mask%* you can select if the left (1), the right (2) or both (3) mouse buttons are used. *State%* determines the button state for which the application is waiting (usually *state%=mask%*).

```
PROCEDURE evnt_mouse(f%,x%,y%,w%,h%)
    DPOKE GINTIN,f%
    DPOKE GINTIN+2,x%
    DPOKE GINTIN+4,y%
    DPOKE GINTIN+6,w%
    DPOKE GINTIN+8,h%
    GEMSYS 22
RETURN
```

This routine allows you to wait until your mouse pointer is within (*f%*=0) or outside (*f%*=1) the given rectangle.

> ***Important: Here and with all other AES routines the coordinates of the rectangle point to the top left corner, the width and the height. Instead VDI, gives these coordinates as two opposite corners of the rectangle.***

```
PROCEDURE evnt_mesag(adr%)
    LPOKE ADDRIN,adr%
    GEMSYS 23
RETURN
```

A message in *GEM* is an event (like the closing of a window). This message is stored in a buffer (starting at *adr%*) containing 16 bytes. The worst message is the *Redraw* message since it requires a lot of work for a programmer because *GEM* does not use its own buffers for graphics.

```
PROCEDURE evnt_timer(t%)
  LPOKE GINTIN+2,t%
  DPOKE GINTIN,t%
  GEMSYS 24
RETURN
```

This routine is a very unproductive wait loop. The parameter *t%* contains the time in milliseconds that the program must wait. This long word (a day has only 86400 seconds) must be represented using the *Intel* format. The switching to *68000* format is performed by the two *POKE* command; this is only possible because *DPEEK(GINTIN+4)* is not used.

```
PROCEDURE evnt_multi                          !more than one
  DPOKE GINTIN,ev_mflags%                     !flags
  DPOKE GINTIN+2,ev_mbcclicks                 !evnt_button
  DPOKE GINTIN+4,ev_mbmask%
  DPOKE GINTIN+6,ev_mbstate%
  DPOKE GINTIN+8,ev_mm1flag%                  !event mouse 1
  DPOKE GINTIN+10,ev_mm1x%
  DPOKE GINTIN+12,ev_mm1y%
  DPOKE GINTIN+14,ev_mm1w%
  DPOKE GINTIN+16,ev_mm1h%
  DPOKE GINTIN+18,ev_mm2flg%                  !event mouse 2
  DPOKE GINTIN+20,ev_mm2x%
  DPOKE GINTIN+22,ev_mm2y%
  DPOKE GINTIN+24,ev_mm2w%
  DPOKE GINTIN+26,ev_mm2h%
  DPOKE GINTIN+28,ev_mtlocount%               !event_timer
  DPOKE GINTIN+30,ev_mthicount%
  LPOKE ADDRIN,ev_mmgpbuff%                   !for message
  GEMSYS 25
RETURN
```

Do you think *ON MENU* is simpler? *ON MENU* uses the exact routine to sample all possible events. The parameters for the timer (*ev_mtxxcount%*) are set to null so that this routine always returns.

Evnt_multi is a combination of the preceding routines. The first parameter selects the type of events the program is waiting for. *Ev_mflags%* is a six digit binary number.

```
&X.....1    = keybd
&X....1.    = button
&X...1..    = mouse 1
&X..1...    = mouse 2
&X.1....    = message
&X1.....    = timer

&X110001    = timer,message,keybd
```

The parameters are similar to the single events. Results are returned with *DPEEK*(*GINTOUT*) to *DPEEK*(*GINTOUT*+2*6).

With *ON MENU* the parameters are returned with *ON MENU* (*xxx*) *GOSUB* (*KEY*, *BUTTON*, *OBOX*, *IBOX*, *MESSAGE*). Results are found in *MENU*(0) to *MENU*(15).

MENU(0) returns the number of the pulled down menu. *Menu*(1)=10 would key on the 10th element in the array.

MENU(1) To *MENU*(8) contains the message buffer.

MENU(9) and *DPEEK*(*GINTOUT*) contains a flag that contains which event last occurred.

MENU(10) X-position of the mouse.

MENU(11) Y-position of the mouse.

MENU(12) Mouse buttons.

MENU(13) *SHIFT*-Status

MENU(14) returns key pressed (high value byte=*ASCII*, low value byte=*SCANCODE*).

MENU(15) Number of mouse clicks.

MENU(9) TO *MENU*(15) correspond to *DPEEK*(*GINTOUT*) to DPEEK(*GINTOUT*+12).

These values are only valid whenever the corresponding Bit is set in Menu(9).

The following messages are possible (the identification number may always be found in menu(1)):

10 *mn_selected*: A drop-down menu was selected.
(0) Calculated array index.
(4) Object-index of the menu title
(5) Object-index of the menu input

20 *wm_redraw*: Part of the screen must be redrawn.
(4) Window handle
(5-8) XYWH, coordinates, width and height of the area that must be redrawn.

21 *wm_topped*: A window was selected.
(4) Window handle

22 *wm_closed*: The close box was clicked.
(4) Window handle

23 *wm_fulled*: The full window box was clicked.
(4) Window handle

24 *wm_arrowed*: One of the arrows was clicked.
(4) Window handle
(5) Number of the arrow that was clicked
0=Page up, 1=Page down
2=Line up, 3=Line down
4=Page left, 5=Page right

6=Column left, 7=Column right

25 *wm_hslid*: Horizontal slider was moved.
(4) Window handle
(5) Relative position of the slider 0..1000

26 *wm_vslid*: Vertical slider was moved.
(4) Window handle
(5) Relative position of the slider 0..1000

27 *wm_sized*: The size of the window was changed.
(4) Window handle
(5-8) XYWH, position and (new) size of the window

28 *wm_moved*: The position of the window was changed.
(4) Window handle
(5-8) XYWH, (new) position and size of the window

29 *wm_newtop*: A new window was activated.
(4) Window handle (Accessory)

40 *ac_open*: An accessory was selected.
(4) Menu Id. (Accessory)
Should be in four according to GEM documentation but instead is found in Menu(5).

41 *ac_close*: The accessory was closed.
(4) Menu Id. (Accessory)

```
PROCEDURE evnt_dclick)speed%,f%)
    DPOKE GINTOUT,speed%             ! 0= slow..4=fast
    DPOKE GINTIN+2,f%                ! 1=set, 0=read
    GEMSYS 26                        ! Double click-Speed
RETURN
```

The routine *evnt_dclick* allows you to change the speed at which the double clicks are processed. The first parameter must be a number between 0 and 4 (just like in the Control panel). The second parameter must be 1 to set the speed or a 0 to read the current speed setting.

5.3 MENU library (Menu usage)

```
PROCEDURE menu_bar(tree%,flg%)
    LPOKE ADDRIN,tree% !menu m$()
    DPOKE GINTIN,flg%
    GEMSYS 30
RETURN
```

Menu_bar allows you to activate *(flg%=1)* or deactivate (flg%=0) a menu object tree at address *tree%*. **GFA BASIC** first creates a corresponding tree with *MENU m$()* that is then activated with an internal *menu_bar* call. *MENU KILL* deactivates the menu (*flg%=0*).

```
PROCEDURE menu_icheck(tree%,item%,flg%)
    LPOKE ADDRIN,tree%
    DPOKE GINTIN,item%
    DPOKE GINTIN+2,flg%
    GEMSYS 31
RETURN
```

Menu_icheck allows you to insert (*flg%=1*) or to erase (*flg%=0*) a check mark to the left of the menu bar. This corresponds to the *MENU n,1* or *Menu n,0*. *MENU n* requires the index to the array while *menu_icheck* requires the number of the menu object tree.

```
PROCEDURE menu_ienable(tree%,item%,flg%)
```

```
    LPOKE ADDRIN,tree%
    DPOKE GINTIN,item%
    DPOKE GINTIN+2,flg%
    GEMSYS 32
RETURN
```

Menu_ienable allows you to activate (*flg%*=1) or deactivate (*flg%*=0) a menu entry. **GFA BASIC** uses *MENU n,3* or *MENU n,2*.

```
PROCEDURE menu_tnormal(tree%,item%,flg%)
    LPOKE ADDRIN,tree%
    DPOKE GINTIN,item%
    DPOKE GINTIN+2,flg%
    GEMSYS 33
RETURN
```

Menu-tnormal allows you to display an individual menu entry in inverse (*flg%*=0) or normal (*flg%*=1). The corresponding Basic command is *MENU OFF*, but this command automatically returns all menu entries to normal.

```
PROCEDURE menu_text(tree%,item%,txt$)
    txt$=txt$+chr$(0)
    LPOKE ADDRIN,tree%
    DPOKE GINTIN,item%
    LPOKE ADDRIN+4,VARPTR(txt$)
    GEMSYS 34
RETURN
```

Menu_text allows you to change the text of a menu entry. It is important that the new text is not any longer than the old text. The number of the object tree must be given as well as the address of a string that is terminated with a null (*+CHR$(0)*). This command is not used in **GFA BASIC**, instead you must use the *MENU m$()* command to activate a new menu tree.

```
PROCEDURE menu_register(ap. id%,nam$)
    nam$=nam$+
```

```
    CHR$(0)
    BMOVE VARPTR(nam$),BASEPAGE+192,LEN(nam$) DPOKE
                                        GINTIN,al.id%
    LPOKE ADDRIN,BASEPAGE+192
    GEMSYS 35
  RETURN
```

The last routine is probably the most interesting since it allows you to activate an accessory by name. Unfortunately, this routine may not be used in **GFA BASIC** (or even the compiler) since the string must remain at a fixed address. You could of course place this string into the Basepage or into an integer array.

5.4 OBJect library (Object manipulation)

The Object-Library allows you to manipulate objects. Objects are the cornerstone of object trees.

An object in *GEM* is always at least 24 bytes long. Some objects may reach the maximum of 64 Kbytes, but most of the time objects are between 24 to 1000 bytes.

The tree structure and the creation of objects are discussed in more detail in the chapter on resources.

```
PROCEDURE obj_add(tree%,parent%,child%)
    LPOKE ADDRIN,tree%
    DPOKE GINTIN,parent%
    DPOKE GINTIN+2,child%
    GEMSYS 40
RETURN
```

Objc_add establishes a logical link between the object (*child%*) and its parent object. It is important that the parent object was already correctly defined (*ob_head* and *ob_tail* are usually -1). The object specifications are never moved in memory; only the pointers are changed.

```
PROCEDURE obj_delete(tree%,obj%)
    LPOKE ADDRIN,tree%
    DPOKE GINTIN,obj%
```

```
    GEMSYS 41
RETURN
```

This routine removes an object from the tree. Just as with *objc_add* only the pointer is changed.

```
PROCEDURE objc_draw(tree%,start%,depth%,x%,y%,w%,h%)
    LPOKE ADDRIN,tree%
    DPOKE GINTIN,start%
    DPOKE GINTIN+2,depth%
    DPOKE GINTIN+4,x%
    DPOKE GINTIN+6,y%
    DPOKE GINTIN+8,w%
    DPOKE GINTIN+10,h%
    GEMSYS 42
RETURN
```

Objc_draw draws an object on the screen. Besides the address of the object tree, the index of the starting tree is given (*start%*). Then the number of levels of subordinate objects that are supposed to be drawn (*depth%*, 0=only object, 1=object and children, 2=object, children and grandchildren,etc.) is given. Next, the position and the size (*XYWH*) of the clipping rectangle are given.

```
PROCEDURE objc_find(tree%,start%,depth%,x%,y%)
    LPOKE ADDRIN,tree%
    DPOKE GINTIN,start% D
    POKE GINTIN+2,depth%
    DPOKE GINTIN+4,x%
    DPOKE GINTIN+6,y%
    GEMSYS 43
RETURN
```

There are times when you need to know that an object on the screen was selected and then pass along that object's number. With *objc_draw* you can determine which object from the object tree at a certain screen address was selected (*x%/y%, MOUSEX/MOUSEY*). Like *objc_draw* the object index *start%* and levels (*depth%*) are passed along.

```
PROCEDURE objc_offset(tree%,obj%)
    LPOKE ADDRIN,tree%
    DPOKE GINTIN,obj%
    GEMSYS 44
RETURN
```

Objc_offset computes the coordinates of the screen object. *DPEEK(GINTOUT)* contains the X-position and *DPEEK(GINTOUT+2)* contains the Y-position.

```
PROCEDURE objc_order(tree%,obj%,new%)
    LPOKE ADDRIN,tree%
    DPOKE GINTIN,obj%
    DPOKE GINTIN+2,chr%
    GEMSYS 45
RETURN
```

Here the object is logically moved, that means the pointer of the *obj%* is changed to *new%* just like in *objc_add* and *objc_delete* no data is ever moved.

```
PROCEDURE objc_edit(tree%,obj%,chr%,pos%,kind%)
    LPOKE ADDRIN,tree%
    DPOKE GINTIN,obj%
    DPOKE GINTIN+2,chr%
    DPOKE GINTIN+4,pos%
    DPOKE GINTIN+6,kind%
    GEMSYS 46
RETURN
```

This is a subroutine of *form_do*. It lets the user edit the text in an object tree. The character *chr%* is placed at position *pos%*. The following editor functions may be performed: Initialize *(kind %=1)*, edit character (*kind %=2*) and done (*kind%=3*). An error is returned in *DPEEK(GINTOUT)* and the new character position is placed in *DPEEK(GINTOUT+2)*.

```
PROCEDURE objc_change(tree%,obj%,x%,y%,w%,h%,new%,flg%)
  LPOKE ADDRIN,tree%
  DPOKE GINTIN,obj%
  DPOKE GINTIN+2,0 !reserved
  DPOKE GINTIN+4,x%
  DPOKE GINTIN+6,y%
  DPOKE GINTIN+8,w%
  DPOKE GINTIN+10,h%
  DPOKE GINTIN+12,new%
  DPOKE GINTIN+14,flg%
  GEMSYS 47
RETURN
```

Objc_change allows you to change the object status and if *flg% =1* the object will be redrawn. You could also receive the same results with *DPOKE tree%+24*obj%+10,new%* or even with *objc_draw.*

5.5 FORM library (Form handling)

```
PROCEDURE form_do(tree%,start%)
    LPOKE ADDRIN,tree%
    DPOKE GINTIN,start%
    GEMSYS 50
RETURN
```

Just like *objc_draw*, this routine is used for handling forms that were previously drawn with the *objc_draw* command. Parameter *start%* passes the index of the object on which the text cursor (vertical line) is to be positioned. The index of the object that caused the end of the input *(EXIT)* is returned with *DPEEK(GINTOUT)*.

> ***Caution: A missing EXIT-object will cause the computer to lock up.***

```
PROCEDURE form_dial(f%,x%,y%,w%,h%,yb%,wb%,hb%)
    DPOKE GINTIN,f%
    DPOKE GINTIN+2,x%
    DPOKE GINTIN+4,y%
    DPOKE GINTIN+6,w%
    DPOKE GINTIN+8,h%
    DPOKE GINTIN+10,xb%
    DPOKE GINTIN+12,yb%
    DPOKE GINTIN+14,wb%
```

```
    DPOKE GINTIN+16,hb%
    GEMSYS 51
RETURN
```

form_dial contains four routines that perform the following functions depending on *flg%*.

0 = Reserve a screen memory area. Unfortunately, *GEM* does not contain its own buffers for the screens so that *form_dial*(0..) only sets aside the memory for later restoration with the *Redraw* command. All of the programs that use forms contain message #20 (*wm_redraw*) and the screens must be reconstructed.
1 = Draws an expanding box that starts at *x%/y%/w%/h%* and grows until it reaches *xb%/yb%/wb%/hb%*.
2 = Same as 1 except for shrinking box.
3 = Frees the screen space reserved (Causes *Redraw* messages to be sent).

1 and 2 are used for the appearance of a program, 0 and 3 could have been replaced with *(S)GET* and *(S)PUT*. This has the advantage of much greater speed.

```
PROCEDURE form_alert(def%,txt$) txt$=txt$+CHR$(0)
    DPOKE GINTIN,def%
    LPOKE ADDRIN,VARPTR(txt$)
    GEMSYS 52
RETURN
```

This is the routine which is similar to the *ALERT* command.

```
@form_alert(1,[2][This is a test][Ok]")
```

corresponds to:

```
ALERT 2,"This is a test",1,"Ok",dummy%
```

The number of the button is returned with *DPEEK(GINTOUT)*.

```
PROCEDURE form_error(num%)
  DPOKE GINTIN,num%
  GEMSYS 53
RETURN
```

This routine displays a warning message. The routine is not very useful since it displays the *MS-DOS* errors found in *IBM* compatible computer (-33=data not found) rather than the *TOS* errors.

```
PROCEDURE form_center(tree_%,x.%,y.%,w.%,h.%)
  LPOKE ADDRIN,tree%
  GEMSYS 54
  *x.%=DPEEK(GINTOUT+2)
  *y.%=DPEEK(GINTOUT+4)
  *w.%=DPEEK(GINTOUT+6)
  *h.%=DPEEK(GINTOUT+8)
RETURN
```

A dialog box is positioned at 0/0 after being loaded to the screen with the *rsrc_load* command. This routine may be used to center the box. Only the coordinates of the root object are changed (Compare *RSC*).

This routine was written using the long form which returns the full set of parameters. I usually prefer the shorter form that passes the parameters with *DPEEK(GINTOUT+...)* instead of the pointers. This has the advantage of speed since the returned values are seldom used anyway.

You could also use *LPOKE X.%,DPEEK(GINTOUT+2)* instead of **x.%=DPEEK(GINTOUT+2)*. This would execute faster, but you would then have to make sure that no false address is ever passed to the routine.

5.6 *GRAF library (Graphic and mouse routines)*

```
PROCEDURE GRAF_RUBBERBOX(x_%,y_%,w_%,h_%,w.%,h.%)
  DPOKE GINTIN,x_%
  DPOKE GINTIN+2,y_%
  DPOKE GINTIN+4,w_%
  DPOKE GINTIN+6,h_%
  GEMSYS 70
  *w.%=DPEEK(GINTIN)
  *h.%=DPEEK(GINTIN+2)
RETURN
```

This routine draws the famous rubberbox. This routine should only be called whenever the mouse button is pressed; the routine terminates as soon as the button is released. Only the left mouse button performs any useful function in *GEM*. The right button may be used in your programs. The parameters consist of the position (usually the mouse position) and the size of the box. The new size of the box is returned after the mouse button is released.

```
PROCEDURE
graf_dragbox(w_%,h_%,x_%,y_%,bx_%,by_%,bw_%,bh_%,x.%,y.%)
  DPOKE GINTIN,w%
  DPOKE GINTIN+2,y_%
  DPOKE GINTIN+4,x_%
```

```
    DPOKE GINTIN+6,y_%
    DPOKE GINTIN+8,bx_%
    DPOKE GINTIN+10,by_%
    DPOKE GINTIN+12,bw_%
    DPOKE GINTIN+14,bh_%
    GEMSYS 71
    *x.%=DPEEK(GINTIN)
    *y.%=DPEEK(GINTIN+2)
RETURN
```

This routine allows the user to move a predefined box within a boundary rectangle. The mouse button works the same way as it did for *graf_rubbox.* The strange way of passing parameters (size,position,position,size) is important since this is the usual method that *GEM* uses. The new position of the box is returned when the mouse button is released.

```
PROCEDURE graf_movebox(w%,h%,x%,y%,dx%,dy%)
    DPOKE GINTIN,w%
    DPOKE GINTIN+2,h%
    DPOKE GINTIN+4,x%
    DPOKE GINTIN+6,y%
    DPOKE GINTIN+8,dx%
    DPOKE GINTIN+10,dy%
    GEMSYS 72
RETURN
PROCEDURE graf_growbox(x%,y%,w%,h%,dx%,dy%,dw%,dh%)
    DPOKE GINTIN,x%
    DPOKE GINTIN+2,y%
    DPOKE GINTIN+4,w%
    DPOKE GINTIN+6,h%
    DPOKE GINTIN+8,dx%
    DPOKE GINTIN+10,dy%
    DPOKE GINTIN+12,dw%
    DPOKE GINTIN+14,dh%
    GEMSYS 73
RETURN
```

```
PROCEDURE graf_shrinkbox(x%,y%,w%,h%,dx%,dy%,dw%,dh%)
  DPOKE GINTIN,x%
  DPOKE GINTIN+2,y%
  DPOKE GINTIN+4,w%
  DPOKE GINTIN+6,h%
  DPOKE GINTIN+8,dx%
  DPOKE GINTIN+10,dy%
  DPOKE GINTIN+12,dw%
  DPOKE GINTIN+14,dh%
  GEMSYS 74
RETURN
```

These three routines are for the moving of dialog boxes. *Graf_movebox* allows you to move a box from one position to another without changing its size. With *graf_growbox* the box is enlarged and with *graf_shrinkbox* the box becomes smaller.

```
PROCEDURE graf_watchbox(tree%,obj%,instate%,outstate%)
  LPOKE ADDRIN,tree%
  DPOKE GINTIN,0
  DPOKE GINTIN+2,obj%
  DPOKE GINTIN+4,instate%
  DPOKE GINTIN+6,outstate%
  GEMSYS 75
RETURN
```

This routine should really belong to the *obj_xxx* routines. Here the object *obj%* of a tree is monitored. This routine is called whenever the mouse button is pressed. The status of the selected object, whenever the mouse pointer is inside the box, is put in *instate%* otherwise the status is put in *outstate%*. A one is returned in *GINTOUT* if the mouse button was released while inside the box, otherwise a null is returned.

```
PROCEDURE graf_slidebox(tree%,parent%,obj%,flg%)
  LPOKE ADDRIN,tree%
  DPOKE GINTIN,parent%
  DPOKE GINTIN+2,obj%
  DPOKE GINTIN+4,flg%
  GEMSYS 76
RETURN
```

This routine should also belong to the *obj_xxx* routines and it is also activated whenever the mouse button is pressed. Within the parent object *parent%* (always a box), the object *obj%* may be moved. *Flg%* selects whether the object is moved horizontal (*flag%=0*) or vertical (*flag%=1*). This routine returns a 0 whenever the object is in the far left (top for vertical) or a 1000 whenever the object is to the far right (bottom for vertical). The object contained in the resource tree is not updated and it is up to the program to match the coordinates of the object tree and to issue a redraw (*obj_draw*).

```
PROCEDURE graf_handle
  DPOKE GINTIN,num%
  LPOKE ADDRIN,adr%
  GEMSYS 78
RETURN
```

Graf_handle selects the *VDI*-handle that *AES* uses for sharing the graphics commands with *VDI*. The width and height of the characters that are used by *AES* along with the width and height of the character cell are also determined.

This is the routine used by *DEFMOUSE*.

graf_mouse(n,xxxxx) = = Defmouse n(n=0..7)
graf_mouse(255,adr) = = Defmouse A$(adr=Varptr(a$)

Num %=256 turns the mouse pointer off and *num% =257* turns the mouse pointer on again.

```
PROCEDURE graf_mkstate(x.%,y.%,but.%,shft.%)
  GEMSYS 79
  *x.%=DPEEK(GINTOUT+2)
  *y.%=DPEEK(GINTOUT+4)
  *but.%=DPEEK(GINTOUT+6)
  *shft.%=DPEEK(GINTOUT+8)
RETURN
```

This is the *AES* mouse input routine. Just like the other mouse routines, this routine determines the position and status of the mouse buttons and the status of the keyboard (*bios(11)*). Since this routine runs under *AES* it is impossible to query the menu line.

5.7 SCRaP Library (Clipboard)

```
PROCEDURE scrp_read(adr%)
    LPOKE ADDRIN,adr%
    GEMSYS 80
RETURN
    PROCEDURE scrp_write(adr%)
    LPOKE ADDRIN,adr%
    GEMSYS 81
RETURN
```

These routines manage the data communication between *GEM* programs. *Scrp_write* copies a string (terminated with a null) into an internal *GEM* buffer that can then be retrieved with the *scrp_read* routine. The *GEM* documentation does not mention a limit but using more than 100 characters may cause some sensitive memory to be overwritten.

This routine can also be used to communicate between different programs like the ones called with the *CHAIN* command. It could also be used to pass a filename. The following routine simplifies the procedure of passing string while using **GFA BASIC**.

```
PROCEDURE scrp_read(str.%)
```

```
    LOCAL tmp_$
    tmp_$=STRING(200,0)
    LPOKE ADDRIN,VARPTR(tmp_$)
    GEMSYS 80
    *str.%=LEFT$(tmp_$,INSTR(tmp_$,CHR$(0))-1)
RETURN
PROCEDURE scrp_write(x$)
    x$=x$+CHR$(0)
    LPOKE ADDRIN,VARPTR(x$)
    GEMSYS 81
RETURN
```

5.8 FileSELector library

This library only contains a single routine, the well known *Fileselect* routine.

```
PROCEDURE fsel_input(padr_%,fadr%)
    LPOKE ADDRIN,padr_%
    LPOKE ADDRIN+4,fadr_%
    GEMSYS 90
RETURN
```

The parameters contain the address of two strings which contain the pathname and the filename. Both strings are filled with null bytes so that they can contain the longest possible path or filename. After selection the path name and the filename are changed within those strings. The usual error message is returned in *DPEEK(GINTOUT)*. *DPEEK(GINTOUT+2)* contains a 1 if the *Ok* box was pressed and a 0 if the *Cancel* box was pressed. The resulting filename is created by combining the path and filename. The pathname is separated from the filename by a "\".

5.9 WINDow library

```
PROCEDURE wind_create(attr_%,x_%,y_%,w_%,h_%,h.%)
    DPOKE GINTIN,attr_%
    DPOKE GINTIN+2,x_%
    DPOKE GINTIN+4,y_%
    DPOKE GINTIN+6,w_%
    DPOKE GINTIN+8,h_%
    GEMSYS 100
    *h.%=DPEEK(GINTOUT)
RETURN
PROCEDURE wind_open(h%,x%,w%,h%)
    DPOKE GINTIN,h%
    DPOKE GINTIN+2,x%
    DPOKE GINTIN+4,y%
    DPOKE GINTIN+6,w%
    DPOKE GINTIN+8,h%
    GEMSYS 101
RETURN
PROCEDURE wind_close(h%)
    DPOKE GINTIN,h%
    GEMSYS 102
RETURN
PROCEDURE wind_delete(h%)
    DPOKE GINTIN,h%
    GEMSYS 103
RETURN
```

The *wind_create* creates a *GEM* window along with its elements (attributes) and the maximum size of the window. This routine returns the handle, a number by which the window will be identified in other routines. *Wind-open* displays a window with its initial size. *Wind_close* closes the window, it disappears from the screen. *Wind_delete* erases the handle of that *GEM* window.

In **GFA BASIC** I used an expanded form of the *OPENW* command because the *GEM* routine is very sensitive to erroneous handles. The **GFA BASIC** *CLOSEW* command is much more robust.

```
PROCEDURE openw(nr%,attr%,x%,y%,w%,h%)
    LOCAL adr%
    adr%=windtab+12*nr%-12
    DPOKE adr%+2,attr%
    DPOKE adr%+4,x%
    DPOKE adr%+6,y%
    DPOKE adr%+8,w%
    DPOKE adr%+10,h%
    OPENW nr%
RETURN
```

This routine allows you to use the normal *CLOSEW* command to close the window. All of the attributes may be used and the window may be positioned anywhere you like.

attr% is represented with bits:

binary	hex	Name	The window...
&x000000000001	&h001	name	has a title line
&x000000000010	&h002	close	has a close box
&x000000000100	&h004	full	has a full box
&x000000001000	&h008	move	has a move box
&x000000010000	&h010	info	has an information line

&x000000100000	&h020	size	may be enlarged
&x000001000000	&h040	uparrow	has an up-arrow
&x000010000000	&h080	dnarrow	has a down-arrow
&x000100000000	&h100	vslid	has a vertical slider
&x001000000000	&h200	lfarrow	has a left arrow
&x010000000000	&h400	rtarrow	has a right arrow
&x100000000000	&h800	hslid	has a horizontal slider

&x000000100011	&h023	has a title, close box, and may be enlarged

When using *Name* and *Info* you must be careful to issue the corresponding *TITLEW*- and *INFOW*- command before you call the *OPENW* routine, otherwise the corresponding bit will automatically be reset during the *OPENW* command. This turns out to be pretty good since *GEM* always uses constant strings for the name and infoline. This is automatically performed with the *TITLEW/INFOW* command.

```
PROCEDURE wind_get%(h%,f%)
    DPOKE GINTIN,h%
    DPOKE GINTIN+2,f%
    GEMSYS 104
RETURN
PROCEDURE wind_set(h%,f%,a1%,a2%,a3%,a4%)
    DPOKE GINTIN,h%
    DPOKE GINTIN+2,f%
    DPOKE GINTIN+4,a1%
    DPOKE GINTIN+6,a2%
    DPOKE GINTIN+8,a3%
    DPOKE GINTIN+10,a4%
    GEMSYS 105
RETURN
```

These routines allow the user to retrieve or change information about a window.

h% is The handle of the window, or null for the desktop background. *f%* selects what kind of information is to be examined or changed. Values for *a1%* to *a4%* depend on *f%*. Returned values can be found starting in *GINTOUT+2*.

wind_get	Name	Returns
h%,4	workxywh	xywh the coordinates of the work window,
h%=0:	window	size without menu bar.
h%,5	cyrrxywh	xywh the coordinates of the entire window, h%=0: window size with menu bar.
h%,6	prexywh	xywh the coordinates of the previous window.
h%,7	fullxywh	xywh the maximum size of the window.
h%,8	hslide	0-1000 position of the horizontal sliders 0=far left to 1000=far right
h%,9	vslide	0-1000 position of the vertical slider 0=top to 1000=bottom
h%,10	top	handle handle of the top window (active)
h%,11	firstxywh	xywh the coordinates of the first window in the windows rectangle list.
h%,12	nextxywh	xywh the coordinates of the next rectangle in the rectangle list.
h%,15	hslsize	0-1000

		relative size of the horizontal slider in 1/1000, -1 = minimum size (a square)
h%,16	vlsize	0-1000 relative size of the vertical sliders

Wind_set also contains numerous possibilities.

wind_set	**Name**	
h%,1,attr	kind	Changes attributes
h%,2,L:adr	name	< = > Titlew
h%,3,L:adr	info	< = > Infow
h%,5,xywh	currxywh	Changes window size and/or position
h%,8,hslid	hslide	Changes position of the horizontal slider (0-1000)
h%,9,vslid	vslide	Changes position of the vertical slider (o-1000)
h%,10	top	Makes window the top (active) window like the openw command on an open window
h%,14,...	newdesk	
h%,15,x	hslsize	Changes the relative size of the horizontal slider
h%,16,x	vslsize	Changes the relative size of the vertical slider

```
PROCEDURE newdesk(tree%,index%)
  LPOKE GINTIN,14
  LPOKE GINTIN+4,tree%
  DPOKE GINTIN+6,index%
  GEMSYS 105
RETURN
```

This routine allows the user to create a new desktop background in the form of an object tree or with (0,0) the default background is drawn.

```
PROCEDURE wind_find(x_%,y_%,h.%)
    DPOKE GINTIN,x_%
    DPOKE GINTIN+2,y_%
    GEMSYS 106
    *h.%=DPEEK(GINTOUT)
RETURN
```

This routine returns the handle of a window that is positioned at a certain screen position (usually the mouse position).

```
PROCEDURE wind_update(flg%)
    DPOKE GINTIN,flg%
    GEMSYS 107
RETURN
```

This routine freezes the rectangle lists of all the windows on the screen. *@window_update(1)* begins update mode and other programs including accessories may no longer modify the screen. *@window_update(0)* ends the update. *@wind_update(3)* allows application to take over full control of the mouse, in other words the *GEM* functions for menu bars and window attributes are no longer active. *@wind_update(2)* returns mouse to *GEM*. In spite of *@windupdate(3)* , the *MENU KEY*, the *ON MENU BUTTON* and the *ON MENU IBOX/OBOX* are still active. This allows the user to use the following procedure for drawing programs:

```
@wind_update(1)                ! freeze rectangle list
@wind_get(handle%,11)          ! first rectangle
@wind_get(handle%,12)          ! next rectangle
IF LPEEK(GINTOUT+6)=0          ! only one rectangle?
   @wind_update(3)
   ...........              ! many commands without being
   ...........              ! interrupted by menus or
   ...........              ! accessories for as long as
   ...........              ! no cancel request is issued like
   ...........              ! Obox (window).
   ...........              ! Then capture input and enable
   @wind_update             ! messages/mouse.
ENDIF
@wind_update(0)             ! and release the rectangle list
```

It is also possible to execute a drawing program that uses the full screen; this will naturally contain null for the window handle and the redrawing is left to *GEM*. A new desktop is created with the *wind_newdesk* routine that contains a filled white rectangle with maximum size and the bit pattern of the picture as *g_image* (*BITBLK*). This has the advantage that you do not have to concern yourself with *REDRAW*. Also study the chapter on resources.

```
PROCEDURE wind_cal(f%,attr%,x%,y%,w%,h%)
   DPOKE GINTIN,f%
   DPOKE GINTIN+2,attr%
   DPOKE GINTIN+4,x%
   DPOKE GINTIN+6,y%
   DPOKE GINTIN+8,w%
   DPOKE GINTIN+10,h%
   GEMSYS 108
RETURN
```

This routine calculates the dimensions of the total area (including borders) from the the inner dimensions (*f%=0*) or it calculates the inner (working area) dimension from the total area of the window (*f%=1*). This routine is usually used to calculate the correct window size required to hold an object (usually the object was created with *RCS*).

5.10 ReSouRCe Library (Resources, Object trees)

```
PROCEDURE rsrc_load(nam$)
    nam$=nam$+CHR$(0)
    LPOKE ADDRIN,VARPTR(nam$)
    GEMSYS 110
RETURN
```

Rsrc_load loads a *RSC* file. It is important to reserve enough memory. If the file is not found or the memory was not sufficient or another error was found, a null will be returned in *DPEEK*(*gintout*).

> ***Caution: This function will search the given disk drive first and then it will search drive A:!***

```
PROCEDURE rsrc_free
    GEMSYS 111
RETURN
```

Rsrc_free frees the memory that was allocated by the resource.

```
PROCEDURE rsrc_gaddr(type_%,index_%,adr.%)
    DPOKE GINTIN,type_%
    DPOKE GINTIN+2,index_%
    GEMSYS 112
    *ADR.%=LPEEK(ADDROUT)
RETURN
```

Rsrc_gaddr returns the addresses of objects and object trees. On the *ST*, this function seems to only work properly for tree structures (*type_%=0*). From this value you can easily determine the addresses of the object (object address equals tree address plus 24 times the object number).

```
PROCEDURE rsrc_tree(index_%,tree.%)
    LPOKE GINTIN,index_%
    GEMSYS 112
    *tree.%=LPEEK(ADDROUT)
RETURN
```

This routine allows you to determine the addresses of object trees.

```
PROCEDURE shel_find(adr%)
    LPOKE ADDRIN,adr%
    GEMSYS 124
RETURN
```

This routine is supposed to store the addresses of the object trees, but unfortunately you have to use the corresponding *LPOKE* commands--sorry.

```
PROCEDURE rsrc_objfix(tree%,index%)
    LPOKE ADDRIN,tree%
    DPOKE GINTIN,index%
    GEMSYS 114
RETURN
```

This routine converts the coordinates of an object within the tree from character coordinates to pixel coordinates. *Rsrc_load* automatically performs this function for the entire tree structure.

5.11 SHELl Library

This is the routines that the *GEM* desktop uses to start programs and also for the construction of the desktop.

```
PROCEDURE shel_read(nam.%,cmd.%)
    LOCAL nam_$,cmd_$
    nam_$=SPACE$(200)
    cmd_$=SPACE$(200)
    LPOKE ADDRIN,VARPTR(nam_$)
    LPOKE ADDRIN+4,VARPTR(cmd_$)
    GEMSYS 120
    *nam.%=LEFT$(nam_$,INSTR(nam_$,CHR$(0))-1)
    *cmd.%=LEFT$(cmd_$,INSTR(cmd_$,CHR$(0))-1)
RETURN
```

This routine allows the program to identify the command by which it was invoked (this could be the name of a file or a command line). It can be used to match the corresponding *RCS* name.

```
PROCEDURE shel_writr(f1%,f2%,f3%,nam$,cmd$)
    nam$-nam$+CHR$(0)
    cmd$=cmd$+CHR$(0)
```

```
    LPOKE ADDRIN,VARPTR(nam$)
    LPOKE ADDRIN+4,VARPTR(cmd$)
    GEMSYS 121
RETURN
```

This routine makes the *CHAIN* command possible in the compiler version of **GFA BASIC**. *Nam$* is the filename of a program and *cmd$* is the command that is passed to that program. Flags *f1%* to *f3%* selects different codes for the program:

f1% =0:Exit GEM (not very useful with the ST)
f1% =1:Run another program
f2% =0:Program runs without graphics
f2% =1:Program uses graphics
f3% =0:Program is not a GEM application
f3% =1:Program is a GEM application

The *CHAIN* command in the compiler sets all flags to 1. *Nam$* contains the passed name and *cmd$* is passed to *Basepage+128*.

```
PROCEDURE rsrc_tree(index_%,tree.%)
    LPOKE GINTIN,index_%
    GEMSYS 112
    *tree.%=LPEEK(ADDROUT)
RETURN
```

This routine allows the *DESKTOP.INF* to be read from the memory and a changed version may then be written back. If you are familiar with the file format, you could, for example, change the serial baudrate and then write the file to the diskette so that the next boot process will automatically set the correct baudrate. You could also change any other parameter.

```
PROCEDURE shel_find(adr%)
    LPOKE ADDRIN,adr%
    GEMSYS 124
RETURN
```

This routine searches for a file whose name starts at *adr%*. If the file was not found on the current disk drive then drive *A:* is also searched. If successful, the full filename is passed to *adr%* otherwise *DPEEK(gintout)=0.*

```
nam$="B:ABC*.BAS"+STRING$(80,0)
@shel_find(VARPTR(nam$))
IF DPEEK(gintout)
    nam$=LEFT$(nam$,INSTR(nam$,CHR$(0))-1)
ELSE
    nam$=""
ENDIF
```

This routine searches for a *.BAS* file whose name starts with *ABC*. It first checks drive *B:* and then drive *A:*. If found, it returns the full filename; wildcards ("?" and "*") are not changed (like "*A:\ABC*.BAS*").

```
PROCEDURE shel_envrn(ptr%,env%)
    LPOKE ADDRIN,ptr%
    LPOKE ADDRIN+4,env%
    GEMSYS 125
RETURN
```

The exact purpose of this routine is unknown to me. It is supposed to search the environment for a string at address *adr%* and to store the byte that immediately follows at address *ptr%*.

Chapter 6

RSC

You have probably already noticed that many programs not only consist of a *PRG*-file but also of a *RSC*-file. What is the purpose of this file?

These resource files contain menu bars, dialog boxes and the like. They contain everything that is possible with *GEM* (*AES*). A perfect example of a resource file is the *GFA_BCOM.RSC* file which contains a box with all of the possible adjustments.

Many programs exist that do not use a resource file and are still able to use menu bars and dialog boxes. This has the advantage that only one file needs to be loaded and the disadvantage that it is much harder to translate to another language. It is possible that all of the text is contained in the resource file, but usually text is found in many places in the program. It is also usually much harder to create an error free structure in your program than to load it from the diskette as a resource. Writing a program to run under different resolutions is also easier with a resource. In theory you should be able to write the programs in sections that are language independent, but the resulting compilation would probably be larger than if the program was newly compiled with the new language elements.

For the creation of normal resource there exists a *RCS* (*Resource Construction Set*, a construction set for the creation of *RCS* files). The development package from *Atari* contains the original construction set from *Digital Research.* Since the instructions for the construction set are rather flimsy let me give you the structure of those resources.

After you start the *RSC* you will see two windows and two icons (*trash* and *clipboard*) displayed. The top window contains the symbols for the many different kinds of object trees. *Unknown* is for any object tree that is not identified; this happens when a *RSC* file is to be edited and the corresponding *DEF*-file is missing. *Alert* is the *Alertbox* (unfortunately the symbols that appear here are not the ones that appear in the final program). *Menu* is a menu tree that contains the menu bars. *Dialog* represents the dialog box which is the most used form of an object tree; it allows you to create very complex input forms. *Free* is a special form of the dialog box which allows you more freedom in designing the individual objects.

These symbols are moved with the mouse to the working window (It requires extensive use of the mouse button to activate the individual windows). Double-clicking allows you to edit the graphics of the object tree. The corresponding object tree appears in the working window and the top window changes into a *Resource-Partbox* from which many different objects may be selected. You can then manipulate these objects with the mouse.

The size of the objects can usually be changed by clicking the right lower corner of the object and then moving the mouse (mouse pointer changes to a hand). Clicking in the middle of the object allows you to move the object to a new location.

The objects may be changed by double-clicking (*text, color, fill pattern, Radio-Buttons, Touchexit*, etc.).

By single-clicking the corresponding selection from the menu bar, the objects or the object trees may be given a name or the information about the objects (trees) may be retrieved or sorted and the bit pattern of the corresponding data loaded as *ICN*-files., or....

The type of the object tree may also be changed like changing a *dialog* to a *free* in order to change the size of the box and then back to *dialog* in order to position the box in an orderly fashion.

> *Tip: If you hold down the* shift *key while moving the object, the object is copied instead of moved. It is also possible to create an* Alertbox, *change the type to* dialog *(by name), and this will put the newly created* icon *into your* resource. *The* clipboard *also has many possibilities.*

It is important to name all of the objects that are used, or you could sort the objects so that they are in a certain order. If the output is created for *Pascal* (.I) or as a *header* for *C* (.H), the resulting lines may be merged into a **GFA BASIC** program and be edited:

Pascal:

```
DESKRSC = 0;        (*TREE*)
WINDRSC = 1;        (*TREE*)
```

C:

```
#define DESKRSC 0       /*TREE*/
#define WINDRSC 1       /*TREE*/
```

GFA BASIC:

```
DESKRSC% = 0;       !TREE
WINDRSC% = 1;       !TREE
```

Now we just need to find out how to use the newly created *RSC* file.

A resource file is loaded with the *rsrc_load* command (an *AES* routine) and can then be manipulated by many *AES* routines. Often some *DPOKES* are required in the corresponding memory to make it work.

I have tried to limit myself to the structure of the object tree just as it is loaded by the *rsrc_load.* The *RSC* file contains pointers to *offsets* and the coordinates are character oriented rather than pixel oriented so that it is easier to change the resource file to the current resolution. Often, however, it is better to write a different resource file for each resolution since icons and other things might look somewhat distorted under a different resolution. The program can check for the resolution by using *XBIOS(4):*

```
on xbios(4) gosub rsc0,rsc1,rsc2,rsc2
...
Procedure rsc0
    @rsrc_load("demolo.rsc")
return
Procedure rsc1
    @rsrc_load("demomid.rsc")
return
Procedure rsc1
    @rsrc_load("demohi.rsc")
return
```

6.1 Resource Construction

An object tree consists of objects (really!) that are defined in a structure consisting of 24 bytes. Often they also contain a data structure like some text or a bit pattern. A *RSC* file can contain many object trees. Each object consists of 10 words (*DPOKE, DPEEK*) or a long word (*LPOKE, LPEEK*) that points to some data.

+0	+2	+4	+6	+8	+10	+12	+16	+18	+20	+22
NEXT	HEAD	TAIL	TYPE	FLAGS	STATE	SPEC.L	X	Y	W	H

NEXT is the number of the next object on the same level that belongs to the same parent object, or the number of the parent object, or the root object (-1, with *DPEEK=65535*).

HEAD is the number of the first subordinate object if one exists or again a -1 (65535).

TAIL is the number of the last subordinate object. *TAIL* is actually not necessary since you could use *HEAD* and *NEXT* to traverse through the tree. It was added to obtain greater speed.

TYPE describes the kind of object as listed in the table below.

FLAGS describes the attributes of an object such as whether or not the object may be selected. *See table.*

STATE describes the status of the object such as whether the object selected or not, etc. *See table.*

SPEC is a long word that contains an address or other data depending on the *TYPE* of the object. Again *see table.*

X, Y, W and *H* contain the coordinates of an object (*X* and *Y*), the width (*W*) and the height (*H*). The coordinates relate to the full screen with the root object and to the parent object for a subordinate object.

Important: Subordinate objects must always be fully contained within the parent object. This requires that the parent object be some kind of a box object.

Type	Nr.	Spec.
G_BOX	20	BOXINFO rectangle
G_TEXT	21	Pointer to TEDINFO Graphic text
G_BOXTEXT	22	Pointer to TEDINFO Text contained within a box
G_IMAGE	23	Pointer to BITBLK bit image graphic
G_PROGDEF	24	Pointer to APPLBLK machine code or 'C'
G_IBOX	25	BOXINFO invisible box, marked by a double framed box
G_BUTTON	26	Pointer to C-String centered text in a box

G_BOXCHR	27	BOXINFO single character in a box
G_STRING	28	Pointer to C-String text of a menu
G_FTEXT	29	Pointer to TEDINFO Editable graphic text
G_FBOXTEXT	30	Pointer to TEDINFO Editable graphic text in a box
G_ICON	31	Pointer to ICONBLK icon, differs from G_IMAGE by being visible on a non white background
G_TITLE	32	Pointer to C-String menu title

BOXINFO: this long word is in bit format:

```
&x ccc cccc dddd dddd rrrr zzzz qmmm ffff
```

c =Character code for *G_BOXCHAR*
d =width of the border.
  0 =no border
  1...127 =border grows inward
  255.128 =border grows outward (256-xxxx)
r =color of the border
z =color of the character (c)
q =flag to draw character with (1) or without (0) white box (Graphmode 1/2)
m =fill pattern (8 possibilities, 0=empty...)
f =color of fill pattern

Example:

```
&H41031233
  41
    03
      1
        2
          3
            3
```

41	character "A"=CHR$(&41)
03	border thickness 3 inward
1	border color 1
2	character color 2
3	fill pattern 3, without white around "A"
3	fill color 3

C-String: The address of a text that ends with a null.

TEDINFO: The address of a table which contains all sorts of information about a stored text.

Contents of this table (3 long words for the address and 8 words, 28 bytes):

te_ptext	address of the text
te_ptmplt	address of the text template
te_pvalid	address of the text that contains the validation characters
te_font	character set (5=normal, 3=small)
te_resvd1	reserved
te_just	justify text , 0=left, 1=right, 2=centered
te_color	color of text &x rrrr zzzz qmmm ffff (see above)
te_resvd2	reserved
te_thickness	border width (0,1..127, 255..128, see above)
te_txtlen	length of te_ptext+1 (with null byte)
te_tmplen	length of te_ptmplt+1 (with null byte)

te_ptext	points to "1234+chr$(0)
te_ptmplt	points to "Price $__.__"+chr$(4)
te_pvalid	points to "9999"+chr$(0)

The output shows: Price $12.34

The text (*te_ptext9*) replaces the underline characters in the validation string (*tp-ptmplt*). While inputting the text (using *form_do* or *objc_edit*), the characters can be restricted by using *te_pvalid*.

The following are legal:

9 = number
A = uppercase character or space
a = upper and lower case character or space
N = uppercase character, number or space
n = upper and lower case character, number or space
F = TOS-filename and : ? *
P = TOS-filename and \ :
p = TOS-filename and \ : ? *
X = any character

In the current version of *TOS* all validation characters other than 9 and *X* will often cause the computer to crash.

While inputting text into a template like the previous example, you can enter a "." to jump past that character. This may be used for any of the template characters that are not permitted in the text.

BITBLK: This structure marks *G_IMAGE*, it is a bit pattern graphic (like *GET/PUT*) that is always displayed in transparent mode (*PUT ...,7*). This structure is usually only available with a white background, but it saves about half of the memory when compared to *ICONBLK*.

First, comes a long word that contains the address of the bit pattern; next, the width of the bit pattern in bytes (one word) and the height; then the X and Y offsets to the pattern *(&x1000* represents an offset of 3); finally we have the color (0..15).

ICONBLK: This structure is for *ICONS* (graphic symbols). The difference between *BITBLK* and *ICONBLK* is that *ICONBLK* contains two bit patterns. The first pattern contains the mask which erases all pixels from the screen for which a bit is set. The second pattern contains the data necessary to set the correct pixels. This is how the white frame around an icon is drawn. You can also draw an icon that contains two colors. The icon could also contain a text line and a single character like the drive symbols displayed with the desktop.

This structure is somewhat more complex than the others; there are three pointers (long words), followed by eleven words.

ib_pmask	address of the mask
ib_pdata	address of the data
ib_char	address of the icon text
ib_char	the single character
ib_xchar	x coordinates of the character (always relative)
ib_ychar	y coordinates of the character
ib_xicon	x coordinates of the icon
ib_yicon	y coordinates of the icon
ib_wicon	width of the icon in pixels
ib_hicon	height of the icons in pixels
ib_xtext	x coordinates of the text
ib_ytext	y coordinates of the text
ib_wtext	width of the text in pixels
ib_htext	height of the text in pixels

APPLBLK: This is an address for machine code or *C* routines that are responsible for the drawing of the objects. An *APPLEBLK* contains two long words of which the first points to the executable routine and the second is passed to the routine.

Run the *WINDOW.BAS* from the enclosed diskette and try to find which object types are used in this program.

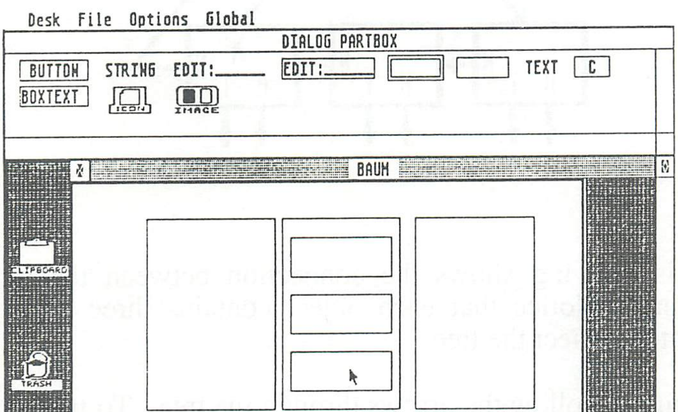
If you think that you have learned how to use *RCS* (If you don't have one then buy one — it is a nightmare to create objects without the *RCS*) then try to follow the structure of the resource. This book contains three pictures that use resources (Unfortunately, each resource is stored by itself, otherwise one could have saved half the file space when using it more than once).

Can you discover how many objects were saved as *Icons* and how many were saved as an *Image*? The screen resolution may be discovered by using the *RCS* Info command.

6.2 RSC1.BAS

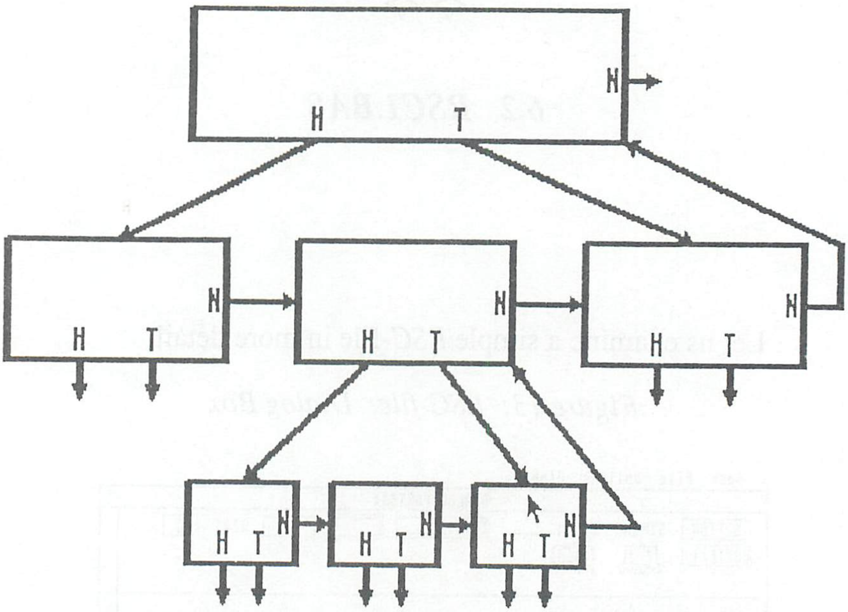
Let us examine a simple *RSC*-file in more detail.

Figure 13: RSC-file: Dialog Box



This *RSC*-file contains one dialog box (BOX) that contains three smaller rectangles (BOX) of which one also contains three smaller rectangles.

Figure 14: Tree Structure



This drawing shows the connection between the tree structure. Notice that each object contains three arrows that interconnect the tree.

You can follow the arrows through the tree. To traverse the whole tree just follow the *Head-pointer* and when that pointer is empty just follow the *Next-pointer*. This is exactly how *GEM* does it. *GEM* also needs to know the maximum level by which the tree should be searched.

The following table shows the *RSC*-file of the above example, first how it looks on the diskette and second, how it looks in memory.

Figure 15: RSC-file Table

On Disk

```
0000 0024 0024 0024 0024 0000 0024 0024 0000
00CC 0007 0001 0000 0000 0000 0000 0000 00D0
Next Head Tail Type Flag Stat   Spec     X    Y    B    H
FFFF 0001 0006 0014 0000 0010 00021100 0000 0000 003D 000E
0002 FFFF FFFF 0014 0000 0000 00FF1100 0009 0002 000F 000A
0006 0003 0005 0014 0000 0000 00FF1100 0019 0002 000E 000A
0004 FFFF FFFF 0014 0000 0000 00FF1100 0001 0001 000C 0002
0005 FFFF FFFF 0014 0000 0000 00FF1100 0001 0004 000C 0002
0002 FFFF FFFF 0014 0000 0000 00FF1100 0001 0007 000C 0002
0000 FFFF FFFF 0014 0020 0000 00FF1100 0029 0002 000E 000A
00000024
```

In Memory

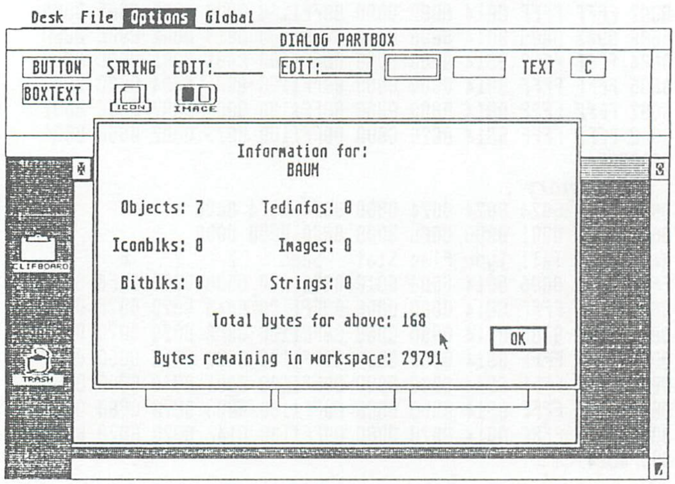
```
0000 0024 0024 0024 0024 0000 0024 0024 0000
00CC 0007 0001 0000 0000 0000 0000 0000 00D0
Next Head Tail Type Flag Stat   Spec     X    Y    B    H
FFFF 0001 0006 0014 0000 0010 00021100 0000 0000 01E8 00E0
0002 FFFF FFFF 0014 0000 0000 00FF1100 0048 0020 0078 00A0
0006 0003 0005 0014 0000 0000 00FF1100 00C8 0020 0070 00A0
0004 FFFF FFFF 0014 0000 0000 00FF1100 0008 0010 0060 0020
0005 FFFF FFFF 0014 0000 0000 00FF1100 0008 0040 0060 0020
0002 FFFF FFFF 0014 0000 0000 00FF1100 0008 0070 0060 0020
0000 FFFF FFFF 0014 0020 0000 00FF1100 0148 0020 0070 00A0
000F4024
```

The beginning of the file contains 18 16-bit numbers that function as pointers. Seven objects follow. The root object can easily be recognized by the *FFFF* in the *NEXT* pointer. The last object has bit #5 set in the *Flag-word.*

More trees can follow with the corresponding construction. The end of the file contains a long word that contains the relative address of the tree at which the file starts. If there is more than one tree the process is repeated. This address will be incremented with the base address during the *rsrc_load* (*HIMEM*, here *&HF4000*).

Looking at the coordinates you can determine that this tree was loaded with a character width of 8 and a character height of 16 (high resolution). With *FREE*-objects the first byte of the coordinates can also contain a gradual step increase of the symbol. This is the reason why there should be different *RSC*-files for each resolution, especially when they contain *Icons* or *Images*. The following hardcopy shows an object tree during the construction with the *RSC* from *Digital Research.*

Figure 16: object tree construction



This small program draws that tree in the center of the screen and inverts the object that the mouse pointer is pointing to.

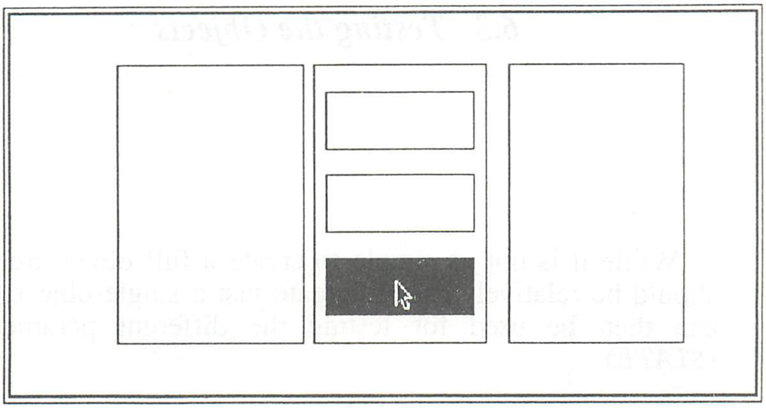
```
' RSCTEST.BAS
'
@Rsrc_free
@Rsrc_load("TEST.RSC")
'
@Rsrc_gaddr(0,0)
Tree%=Lpeek(Addrout)
@Form_center(Tree%)
@Objc_draw(Tree%,0,7,0,0,640,400)
REPEAT
```

```
        @Objc_find(Tree%,0,7,Mousex,Mousey)
        O%=Dpeek(Gintout)
        If O%>0 And O%<1000
            @Objc_change(Tree%,O%,0,0,640,400,1,1)
            Repeat
                @Objc_find(Tree%,0,7,Mousex,Mousey)
            Until O%<>Dpeek(Gintout)
            @Objc_change(Tree%,O%,0,0,640,400,0,1)
        Endif
    Until Mousek
    @Rsrc_free
    '
    Procedure Objc_draw(Tree%,Start%,Depth%,X%,Y%,B%,H%)
        Lpoke Addrin,Tree%
        Dpoke Gintin,Start%
            Dpoke Gintin+2,Depth%
        Dpoke Gintin+4,X%
        Dpoke Gintin+6,Y%
        Dpoke Gintin+8,B%
        Dpoke Gintin+10,H%
        Gemsys 42
    Return
    Procedure Objc_find(Tree%,Start%,Depth%,X%,Y%)
        Lpoke Addrin,Tree%
        Dpoke Gintin,Start%
        Dpoke Gintin+2,Depth%
        Dpoke Gintin+4,X%
        Dpoke Gintin+6,Y%
        Gemsys 43
    Return
    Procedure Objc_change(Tree%,Obj%,X%,Y%,B%,H%,Neu%,Flg%)
        Lpoke Addrin,Tree%
        Dpoke Gintin,Obj%
        Dpoke Gintin+2,0 !reserved
        Dpoke Gintin+4,X%
        Dpoke Gintin+6,Y%
        Dpoke Gintin+8,B%
        Dpoke Gintin+10,H%
        Dpoke Gintin+12,Neu%
        Dpoke Gintin+14,Flg%
```

```
    Gemsys 47
Return
Procedure Form_do(Tree%,Start%)
    Lpoke Addrin,Tree%
    Dpoke Gintin,Start%
    Gemsys 50
Return
Procedure Form_dial(F%,X%,Y%,B%,H%,Xb%,Yb%,Bb%,Hb%)
    Dpoke Gintin,F%
    Dpoke Gintin+2,X%
    Dpoke Gintin+4,Y%
    Dpoke Gintin+6,B%
    Dpoke Gintin+8,H%
    Dpoke Gintin+10,Xb%
    Dpoke Gintin+12,Yb%
    Dpoke Gintin+14,Bb%
    Dpoke Gintin+16,Hb%
    Gemsys 51
Return
Procedure Form_center(Tree%)
    Lpoke Addrin,Tree%
    Gemsys 54
Return
Procedure Rsrc_load(Nam$)
    Nam$=Nam$+Chr$(0)
    Lpoke Addrin,Varptr(Nam$)
    Gemsys 110
Return
Procedure Rsrc_free
    Gemsys 111
Return
Procedure Rsrc_gaddr(Type%,Index%)
    Dpoke Gintin,Type%
    Dpoke Gintin+2,Index%
    Gemsys 112
Return
```

A hardcopy of the screen is shown below.

Figure 17: RSCTEST.BAS hardcopy



6.3 Testing the Objects

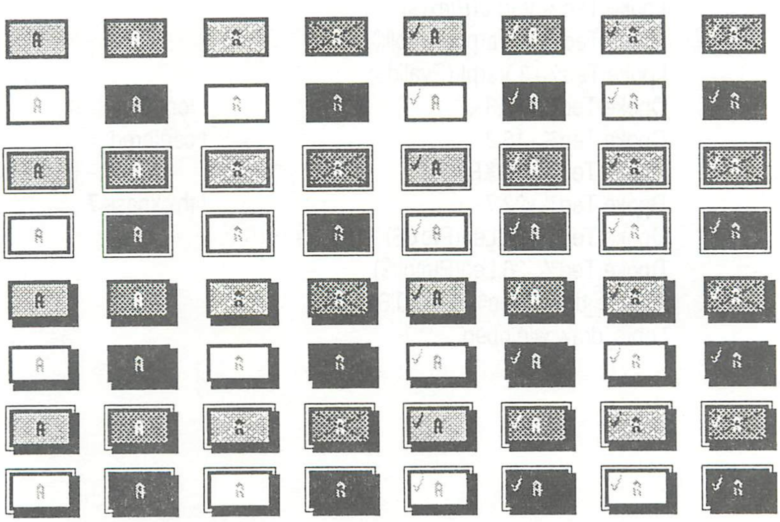
While it is not as simple to create a full object tree, it should be relatively easy to create just a single object that can then be used for testing the different parameters (*STATE*).

```
' BOXRSC
'
'
Dim A%(100)
A%=Varptr(A%(0))
Dpoke A%,-1
Dpoke A%+2,-1
Dpoke A%+4,-1
Dpoke A%+6,27                         ! G_BOXCHAR
Dpoke A%+8,&H20                       ! lastob
Dpoke A%+10,0
Lpoke A%+12,&H41031233
Dpoke A%+16,0
Dpoke A%+18,0
Dpoke A%+20,50
Dpoke A%+22,30                        ! color change to ,15
For I%=0 To 63
   Dpoke A%+10,I%
   Dpoke A%+16,10+(I% And 7)*78
   Dpoke A%+18,10+(I%/8 And 7)*50     ! color change to *25
   @Objc_draw(A%,0,0,0,0,640,400)
```

```
Next I%
Procedure Objc_draw(Tree%,Start%,Depth%,X%,Y%,B%,H%)
    Lpoke Addrin,Tree%
    Dpoke Gintin,Start%
    Dpoke Gintin+2,Depth%
    Dpoke Gintin+4,X%
    Dpoke Gintin+6,Y%
    Dpoke Gintin+8,B%
    Dpoke Gintin+10,H%
    Gemsys 42
Return
```

First, a small object is defined in the tiny program above (*BOXCHAR*, the letter "A" inside of a filled rectangle). Then the position and the *STATE* flags are incremented 64 times, each time showing a different picture on the screen. The result is shown in the printout below. You can easily see the result that the *STATE* attribute has on the object.

Figure 18: Small Object incremented



Those with color monitors will have to change the two lines marked in the comment line.

```
FBOXTEXTFBOXTEXT
' FBOXTEXT
'
Dim Tree%(100),Ted%(100)
Tree%=Varptr(Tree%(0))
Ted%=Varptr(Ted%(0))
Dpoke Tree%,-1
Dpoke Tree%+2,-1
Dpoke Tree%+4,-1
Dpoke Tree%+6,30                    ! G_FBOXTEXT
Dpoke Tree%+8,&H20                  ! lastob
Dpoke Tree%+10,0
Lpoke Tree%+12,Ted%
Dpoke Tree%+16,10
Dpoke Tree%+18,50
Dpoke Tree%+20,300
Dpoke Tree%+22,130
Ptext$="1234"+Chr$(0)
Ptmplt$="Price __.__"+Chr$(0)
Pvalid$="9999"+Chr$(0)
Lpoke Ted%,Varptr(Ptext$)
Lpoke Ted%+4,Varptr(Ptmplt$)
Lpoke Ted%+8,Varptr(Pvalid$)
Dpoke Ted%+12,3                       ! font 5
Dpoke Ted%+16,2                       ! centered
Dpoke Ted%+18,&H1111
Dpoke Ted%+22,7                       ! thickness 7
Dpoke Ted%+24,Len(Ptext$)
Dpoke Ted%+26,Len(Ptmplt$)
@Objc_draw(Tree%,0,0,0,0,640,400)
' objc_draw wie oben
```

Figure 19: Text inside a filled rectangle

This program writes a formatted graphic text inside of a filled rectangle.

Both of the above programs indicate that it takes a lot of effort just to receive some minor results. If you use a *RCS* instead to construct the objects, you can do all of the construction graphically by using the mouse. You can then also change the objects without changing the program as long as the objects remain in the same order. With longer programs all of the *AES*-subroutines necessary for *RSC* will appear to occupy less space.

6.4 ICONs

So far, so good. *RCS* is not bad, but where does the data for the *Icons* and *Images* come from? **GFA BASIC** contains commands that allow you to format the graphic data used with *RCS*. These commands are *GET* and *PUT*.

There are already many icon editors available that are written in **GFA BASIC**. These will allow you to save screen segments to a diskette. They are usually constructed like this:

```
GET x0,y0,x1,y1,x$
BSAVE "ICON.GET",VARPTR(x$),LEN(x$)
```

The same could of course be accomplished from within a program.

The *RCS* expects the source text for the Icon and Image data to be in *C*-compiler format. The conversion, changing a **GFA BASIC** screen segment into that source text, is shown in the program beginning on the next page:

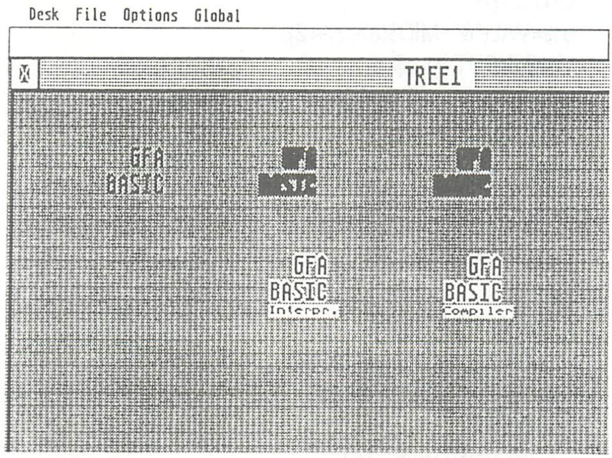
```
' ---------------
' - MAKEICON.BAS -
' ---------------
' GET x,y,z,t,x$                          ! Read into string
' BSAVE "TEST.GET",VARPTR(x$),LEN(x$)              !to Diskette
'
Open "I",#1,"TEST.GET"
A$=Input$(Lof(#1),#1)
Close #1
'
   B%=Cvi(A$)+16 And &HFFF0
   H%=Cvi(Mid$(A$,3))+1
   R%=Cvi(Mid$(A$,5))*2          !Double amount of Bitplanes
   Cls
   Put 0,0,A$
   Get 0,0,B%-1,H%-1,A$
   '
   Open "O",#1,"TEST.ICN"
   Print #1,"/* GFA SHAPE */"
   Print #1,"#define SHAP_W ";@H$(B%)
   Print #1,"#define SHAP_H ";@H$(H%)
   Print #1,"#define DATASIZE ";@H$(B%*H%/16)
   Print #1,"int image[DATASIZE] = int mas"
   Print #1,"{ ";
   For I%=1 To B%*H%/16-1
      Print #1,@H$(Cvi(Mid$(A$,I%*2+5)));", ";
      If I% Mod 4=0
         Print #1
         Print #1," ";
      Endif
   Next I%
   Print #1,@H$(Cvi(Mid$(A$,I%*2+5)))
   Print #1,"};"
   Close #1
RETURN
'
Deffn H$(X%)="0x"+Right$("000"+Hex$(X%),4)
```

This program will create a file with the extension of ".ICON " which may then be used with the *RCS*. Unfortunately, the size of an icon may not exceed 700 bytes when using *RCS*. If *b%*h%/16* is larger than about 350, you must be cautious or the *RCS* will show a mutilated graphic picture (maybe other *RCS* programs can do a better job).

It is also necessary to create a mask for the icon. You can design your own or you can pass on this task to a program. The programmatic solution (never perfect) allows you to surround an all black pixel with other black pixels. It is also possible to create white areas within the inner surface of the symbol.

The program also creates two other files besides *ICON.ICN*. The file *ICONM.ICN* contains a mask with the first format and file *ICONW.ICN* contains a file with the second format.

Figure 20: Icons



The above picture shows three images in the top row consisting of *ICON.ICN*, *ICONM.ICN* and *ICONW.ICN* (from left to right). The second row contains two icons, the left contains *ICONM.ICN* as its mask and the other contains *ICONW.ICN*.

If you want to use this program often, you could further develop it by asking for the filename with *FILESELECT*, offer choices of masks, etc.

If you already have a **GFA BASIC** drawing program, you can expand it by adding an *ICON*-editor option.

It is also possible to convert a drawing program to an *ICON*-editor.

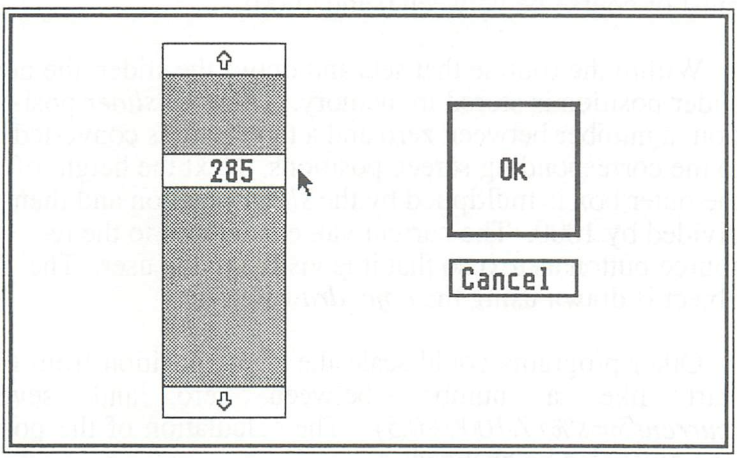
```
'READICON.BAS
'
FILESELECT "\*.ICN",".ICN",file$
OPEN "I",#1,file$
```

```
REPEAT
    LINE INPUT #1,a$
    q%=INSTR(a$,"0X")
UNTIL q%
b%=VAL("&"+MID$(a$,q%+2))
LINE INPUT #1,a$
h%=VAL("&"+MID$(a$,INSTR(a$,"0x")+2))
LINE INPUT #1,a$
size%=VAL("&"+MID$(a$,INSTR(a$,"0x)+2))
GET 0,0,b%-1,h%-1,x$
p%=CVI(MID$(x$,5))
x$=LEFT$(x$,6)
a$=""
FOR i%=1 TO size%
    WHILE INSTR(a$,"0x")=0
        LINE INPUT #1,a$
    WEND
    q%=INSTR(a$,"0x")
    a$=MID$(a$,q%+2)
    x$=x$+STRING$(p%,MKI$(VAL("&"+a$)))
NEXT i%
CLOSE #1
PUT 0,0,x$
' BSAVE "ICON.GET",VARPTR(x$),LEN(x$)
```

You will have to imbed this routine in a drawing program to make it work properly.

6.5 Touchexit

Figure 21: Dialog Box with Slider



Here is an example of a dialog box (*bjc_draw* and *form_do*) for moving an object within a *slider* bar.

In this program a *button* consisting of four numbers is moved within a frame. By using the top and bottom arrows, the slider can be moved in single steps. Even though the button as well as the arrows were defined as *Touchexit*, the control of the resource is passed to the program using *form_do*. The program checks to see if the *slider* or one of the arrows was selected. If *OK* or *Cancel* was selected, the *form_do* will terminate; otherwise, it exits. The subroutine slide controls the *sliders*. This routine uses the *GEM*-routine (*graf_slidebox*) to adjust and reposition the *slider*.

Using the arrows is very simple. First, the selection of the object is canceled. Then the *slider* is moved in the corresponding direction until the mouse button is released (it must of course be between 0 and 1000).

Within the routine that sets and draws the slider, the new slider position is stored in memory. Then the *slider* position, a number between zero and a thousand, is converted to the corresponding screen positions. Next the height of the outer box is multiplied by the *slider* position and then divided by 1000. The current value is written to the resource button as text so that it is visible to the user. The object is drawn using the *objc_draw* routine.

Other programs could scale the *slider* position from the start like a number between zero and seven (*current%=s%*7/1000+0.5*). The calculation of the position must also be changed.

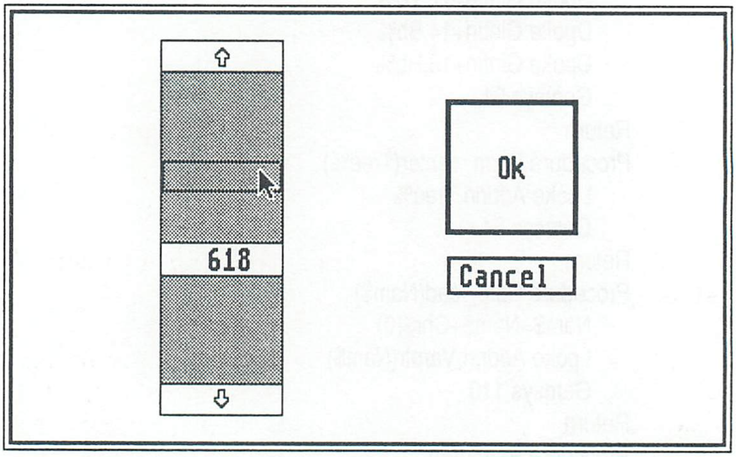
```
'SLIDER
@Rsrc_free
@Rsrc_load("SLIDER.RSC")
'
T.tree%=0               ! (* TREE *)
O.slider%=2             ! (* OBJECT in TREE #0 *)
O.parent%=1             ! (* OBJECT in TREE #0 *)
A.up%=3                 ! (* OBJECT in TREE #0 *)
A.down%=4               ! (* OBJECT in TREE #0 *)
B.ok%=5                 ! (* OBJECT in TREE #0 *)
```

```
B.cancel%=6                    ! (* OBJECT in TREE #0 *)
'
@Rsrc_gaddr(0,T.tree%)
Tree%=Lpeek(Addrout)
@Form_center(Tree%)
@Set_slide(500)
'
@Objc_draw(Tree%,0,7,0,0,639,399)
Repeat
  @Form_do(Tree%,0)
  X%=Dpeek(Gintout)
  If X%=O.slider%
    @Slide
  Else
    If X%=A.up%
      @Slide_up
    Else
      If X%=A.down%
        @Slide_down
      Endif
    Endif
  Endif
Until X%=B.ok% Or X%=B.cancel%
@Rsrc_free
'
Procedure Slide
  @Graf_slidebox(Tree%,O.parent%,O.slider%,1)
  @Set_slide(Dpeek(Gintout))
Return
Procedure Set_slide(S%)
  Current%=S%
  Mh%=Dpeek(Tree%+24*O.parent%+22)             ! Height parent
  Sub Mh%,Dpeek(Tree%+24*O.slider%+22)     ! Height Slider
  Dpoke Tree%+24*O.slider%+18,S%*Mh%/1000
  S$=Right$("   "+Str$(S%),4)
  Sa%=Lpeek(Tree%+24*O.slider%+12)
  Lpoke Sa%,Cvl(S$)
  @Objc_draw(Tree%,O.parent%,1,0,0,639,399)
Return
Procedure Slide_up
```

```
    @Objc_change(Tree%,A.up%,0,0,640,400,0,1)
    Repeat
      @Set_slide(Max(0,Current%-1))
    Until (Mousek And 1)=0
Return
Procedure Slide_down
    @Objc_change(Tree%,A.down%,0,0,640,400,0,1)
    Repeat
      @Set_slide(Min(1000,Current%+1))
    Until (Mousek And 1)=0
Return
'
Procedure Objc_draw(Tree%,Start%,Depth%,X%,Y%,B%,H%)
    Lpoke Addrin,Tree%
    Dpoke Gintin,Start%
    Dpoke Gintin+2,Depth%
    Dpoke Gintin+4,X%
    Dpoke Gintin+6,Y%
    Dpoke Gintin+8,B%
    Dpoke Gintin+10,H%
    Gemsys 42
Return
Procedure Objc_change(Tree%,Obj%,X%,Y%,B%,H%,Neu%,Flg%)
    Lpoke Addrin,Tree%
    Dpoke Gintin,Obj%
    Dpoke Gintin+2,0                            !reserved
    Dpoke Gintin+4,X%
    Dpoke Gintin+6,Y%
    Dpoke Gintin+8,B%
    Dpoke Gintin+10,H%
    Dpoke Gintin+12,Neu%
    Dpoke Gintin+14,Flg%
    Gemsys 47
Return
Procedure Form_do(Tree%,Start%)
    Lpoke Addrin,Tree%
    Dpoke Gintin,Start%
    Gemsys 50
Return
Procedure Form_dial(F%,X%,Y%,B%,H%,Xb%,Yb%,Bb%,Hb%)
```

```
        Dpoke Gintin,F%
        Dpoke Gintin+2,X%
        Dpoke Gintin+4,Y%
        Dpoke Gintin+6,B%
        Dpoke Gintin+8,H%
        Dpoke Gintin+10,Xb%
        Dpoke Gintin+12,Yb%
        Dpoke Gintin+14,Bb%
        Dpoke Gintin+16,Hb%
        Gemsys 51
    Return
    Procedure Form_center(Tree%)
        Lpoke Addrin,Tree%
        Gemsys 54
    Return
    Procedure Rsrc_load(Nam$)
        Nam$=Nam$+Chr$(0)
        Lpoke Addrin,Varptr(Nam$)
        Gemsys 110
    Return
    Procedure Rsrc_free
        Gemsys 111
    Return
    Procedure Rsrc_gaddr(Type%,Index%)
        Dpoke Gintin,Type%
        Dpoke Gintin+2,Index%
        Gemsys 112
    Return
    Procedure Graf_slidebox(Tree%,Parent%,Obj%,Flg%)
        Lpoke Addrin,Tree%
        Dpoke Gintin,Parent%
            Dpoke Gintin+2,Obj%
        Dpoke Gintin+4,Flg%
        Gemsys 76
    Return
```

Figure 22: Positioning the Slider



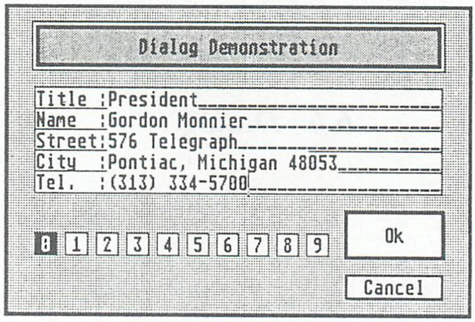
6.6 Dialog

Now the most extensive example of this chapter: A dialog box with text input.

If you have made it this far in *AES* programming, this should not be any harder. This resource consists of a dialog box with some text fields, some radio buttons (contained in a box without a frame), and an *OK* and *CANCEL* button. After *rsrc_load* is called, the tree address is determined. Then *form_center* is called to get the coordinates of this dialog box. After the initializing of the objects, the input routine is called. This routine saves the background with the *SGET* and the *SPUT* command (this is simpler and faster than using *form_dial*(0,...) and *form_dial*(3,...)).

Next, the routine draws an expanding box and also the object. After calling *form_do* the Exit object is deselected. After cancellation, the input is repeated (a real program would handle this differently).

Figure 23: Dialog Box with Text Input



If *OK* is selected, the screen is restored by the *form_dial* effect and the *SPUT* command. Then the text fields are read and the radio buttons are interpreted.

The status (*STATE*) of an object is easy to read or change with the *DPEEK* and *DPOKE* commands.

To read the text is somewhat harder since the object address must be determined and the pointers *Spec* and *Ptext* must be read. By using the combination of *BMOVE* and *INSTR*, we were able to eliminate a loop to check for the null byte and also additions of strings. The writing of the text is somewhat awkward; the *MIN* serves to make sure that the text field does not interfere with the memory of the resource.

While constructing this resource, it is important that the radio buttons are sorted (this greatly simplifies the interrogation) and that all editable text objects contain the full length (this may be recognized in the *DR-RCS* in that the line cursor is positioned at the last character of the last line).

```
' DIALOG
'
Demo%=0                          ! (* TREE *)
Title%=2                         ! (* OBJECT in TREE #0 *)
Nam%=3                           ! (* OBJECT in TREE #0 *)
Street%=4                        ! (* OBJECT in TREE #0 *)
City%=5                          ! (* OBJECT in TREE #0 *)
Tel%=6                           ! (* OBJECT in TREE #0 *)
Ok%=18                           ! (* OBJECT in TREE #0 *)
Cancel%=19                       ! (* OBJECT in TREE #0 *)
Null%=8                          ! (* OBJECT in TREE #0 *)
'
@Rsrc_load("dialog.rsc")
@Rsrc_gtree(Demo%,*Tree%)
@Form_center(Tree%)
X%=Dpeek(Tree%+16)
Y%=Dpeek(Tree%+18)
B%=Dpeek(Tree%+20)
H%=Dpeek(Tree%+22)
@Sstate(Tree%,Null%,1)              ! 0 selected!!!
For I%=Null%+1 To Null%+9           ! 1-9 not
@Sstate(Tree%,I%,0)
Next I%
@Stext(Tree%,Title%,"Firma")
@Stext(Tree%,Nam%,"GFA Systemtechnik GmbH")
@Stext(Tree%,Street%,"Heerdter Sandberg 30")
@Stext(Tree%,City%,"4000 D sseldorf 11")
@Stext(Tree%,Tel%,"0211/588011")
'
@Input_routine
'
Print "Title : ";Title$
Print "Name : ";Nam$
Print "Street: ";Street$
Print "City : ";City$
```

```
Print "Tel. : ";Tel$
Print "Call# : ";Radio%
'

Procedure Input_routine
  Sget Temp$
  Do
    @Form_dial(1,10,10,0,0,X%,Y%,B%,H%)
    @Objc_draw(Tree%,0,8,X%,Y%,B%,H%)
    @Form_do(Tree%,Title%)
    Ex%=Dpeek(Gintout)
    @Sstate(Tree%,Ex%,0)
    Exit If Ex%=Ok%
    Out 2,7
  Loop
  @Form_dial(2,0,0,0,0,X%,Y%,B%,H%)
  Sput Temp$
  '
  @Gtext(Tree%,Title%,*Title$)
  @Gtext(Tree%,Nam%,*Nam$)
  @Gtext(Tree%,Street%,*Street$)
  @Gtext(Tree%,City%,*City$)
  @Gtext(Tree%,Tel%,*Tel$)
  For Radio%=Null% To Null%+9
    @Gstate(Tree%,Radio%,*S%)
    Exit If S% And 1
  Next Radio%
  Sub Radio%,Null%
  '
Return
'
'
Procedure Objc_draw(Tree%,Start%,Depth%,X%,Y%,B%,H%)
  Lpoke Addrin,Tree%
  Dpoke Gintin,Start%
  Dpoke Gintin+2,Depth%
  Dpoke Gintin+4,X%
  Dpoke Gintin+6,Y%
  Dpoke Gintin+8,B%
  Dpoke Gintin+10,H%
  Gemsys 42
```

```
Return
Procedure Form_do(Tree%,Start%)
    Lpoke Addrin,Tree%
    Dpoke Gintin,Start%
    Gemsys 50
Return
Procedure Form_dial(F%,X%,Y%,B%,H%,Xb%,Yb%,Bb%,Hb%)
    Dpoke Gintin,F%
    Dpoke Gintin+2,X%
    Dpoke Gintin+4,Y%
    Dpoke Gintin+6,B%
    Dpoke Gintin+8,H%
    Dpoke Gintin+10,Xb%
    Dpoke Gintin+12,Yb%
    Dpoke Gintin+14,Bb%
    Dpoke Gintin+16,Hb%
    Gemsys 51
Return
Procedure Form_center(Tree%)
    Lpoke Addrin,Tree%
    Gemsys 54
Return
Procedure Rsrc_load(Nam$)
    Nam$=Nam$+Chr$(0)
    Lpoke Addrin,Varptr(Nam$)
    Gemsys 110
Return
Procedure Rsrc_free
    Gemsys 111
Return
Procedure Rsrc_gaddr(Type%,Index%)
    Dpoke Gintin,Type%
    Dpoke Gintin+2,Index%
    Gemsys 112
Return
Procedure Rsrc_gtree(Index_%,Tree.%)
    Lpoke Gintin,Index_%
    Gemsys 112
    *Tree.%=Lpeek(Addrout)
Return
```

```
'
Procedure Gstate(T_%,N_%,X.%)
    *X.%=Dpeek(T_%+24*N_%+10)
Return
'
Procedure Sstate(T_%,N_%,X_%)
    Dpoke T_%+24*N_%+10,X_%
Return
'
Procedure Gtext(T_%,N_%,X.%)
    Local X_$
    X_$=Space$(100)
    T_%=Lpeek(Lpeek(T_%+24*N_%+12))
    Bmove T_%,Varptr(X_$),100
    *X.%=Left$(X_$,Instr(X_$,Chr$(0))-1)
Return
'
Procedure Stext(T_%,N_%,X_$)
    X_$=X_$+Chr$(0)
    T_%=Lpeek(T_%+24*N_%+12)
    Bmove Varptr(X_$),Lpeek(T_%),Min(Len(X_$),Dpeek(T_%+24)-
1)
Return
```

CHAPTER 7

USING WINDOWS

This chapter consists of a long demo program called *WINDOWDEMO*. The source listing is included. This program contains a lot of information since many things can be done with windows. You can move them, enlarge them, shrink them, select them, or you can turn the sliders on and off. The window can also contain text with many different attributes (thick,cursive), or it can contain many different character sets, or a graphic picture in bit pattern format like in drawing programs, or as vector graphic, or as object, resource file, or ...

When you run this program you will see a screen with many different windows. There are four windows all together that partially overlap each other. One window shows text, another shows simple line graphic, another shows a typical object tree and another shows a picture that could have come from a drawing program. There are also ten boxes on the left side of the screen that represent the F1 to F10 function keys. The background is the normal desktop.

As you play with the mouse you will notice the following:

> As you click one of the F-boxes, this box is inverted. Pressing the function key gives you the same result.

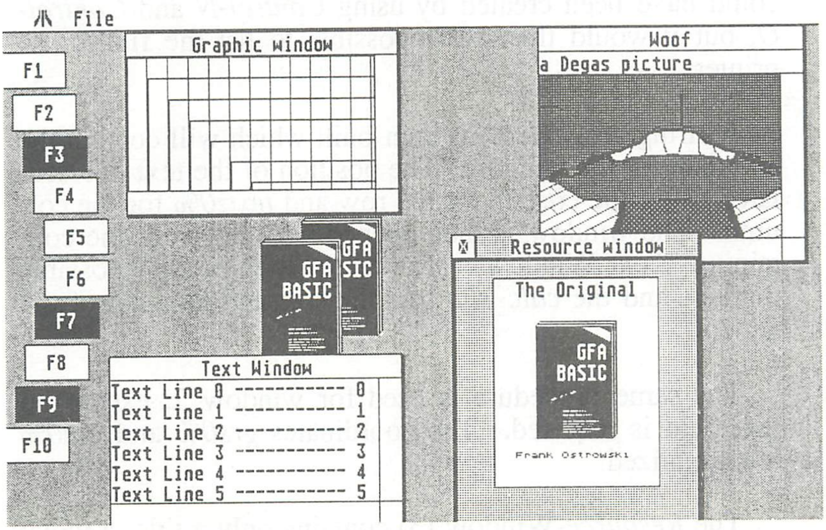
The resource window may be moved or closed.

You can enlarge both the text and the graphic windows and you can move the contents around by using the arrows or sliders.

The window with the line graphics can only be moved eight steps at a time in the horizontal direction. The *Fullw*-fields can also be activated. You can also call accessories.

The menu bar shows the *Atari* symbol and the *Quit* command which is shown under the *File*-Menu. If you move the window (also accessories), the old contents are restored.

Figure 24: Windows



For the creation of this program:

After a procedure is called to set the *GEM*-parameters (close window, *rsrc_free*, etc.), a *Resource*-File is loaded in reserved memory.

This *RSC*-File contains two object trees, a *Deskrsc* for the Function key symbols and a *Windrsc* for one of the windows.

The *Deskrsc* is modified since it is not possible to display a full screen with *Resource. Wind_get(0,4)* is called to set the size of the screen without a menu bar. Afterwards this tree is installed as the new desktop background (with *wind_set(0,14))*.

The default text size and the maximum window size is then determined. It is not always possible to use *Gemsys 77(graf_handle)*.

Next a menu bar is created by reading the selections from a data statement and issuing the Menu command. *Chr$(14)+Chr$(15)* is the *Atari* symbol. This symbol could have been created by using *Control-N* and *Control-O*, but it would then be impossible to list the file to the printer.

A string array (*txt$*) is then built which will contain the text data for the window. The position of the text is calculated using the *txts0%* for the row and *textz0%* for the column. The window is then opened with the Info line containing "Text Window". The *&HFFF* selects all possible markers and the *calc_slid* sets the size and location of the sliders.

The same procedure is used for window 2, except no text field is required. The coordinates *grs0%* and *grz0%* are initialized.

The *Resource*-Window (3) contains only a title, a move bar, and a close box *(&X1011)*. The window size is selected using *wind_calc* so that the corresponding Resource fits exactly into that window. This window does not contain any sliders or a size box, making it impossible to select those fields. The routines for those attributes are therefore not part of this program.

The fourth window has all the attributes. This window contains a graphic that is loaded from the diskette into a string *(X$)*. This is similar to the text field for window (1). The screen is then loaded with the *INPUT$* command. A title and an info line are also added.

By using the *form_dial* command the new screen including the new background are redrawn. The three windows would be redrawn even without this call.

After the *ON-MENU* routines are set, the program performs the main loop which can only be interrupted by setting the *end!* flag. The loop contains only the *evnt_multi* call *(ON MENU)*.

The following lines are only used while testing the program and should be replaced with the *END* command on a finished product.

Entry #1 in the *Menu* routine shows an alert box and entry #14 sets the flag *(end!)* to indicate the end of the program. For easier visibility of the menu bar it is very important to issue the *MENU-OFF* command.

The *Key*-routine selects the *SCAN-Code* to see if a function key was pressed, then it selects the corresponding routine for inverting the *F*-box.

Within the *Button*-routine, it is determined if the mouse was pointing to the background or a window *(wind_find)*. If it points to the background then the *(obj_find)* routine is called to check if the mouse is pointing to a Function box. If a function box was found, the box is inverted.

The routine *desk_change* changes the status of the *F*-symbols. The address of the object is determined (*rsrc_gaddr* does not seem to work) and the new status of the function key displayed (with *XOR 1*). A real program would copy this status to a field for further use. The new inverted symbol is then redrawn by using *obj_draw*.

The *Message*-routine must be able to react to many different actions. First, a check is made to see if a *wm_xxx*-message exists. If none exists, this program will ignore the messages.

The *Window*-handle is assigned to a variable *(hand%)*. The window number is also assigned to the corresponding variable. This somewhat odd looking routine is the most efficient for the compiler.

Every possible message calls its own routine.

The *wm_closed* routine is very simple. If you want more security, you could add an alert box.

The *wm_topped* routine uses the corresponding GEM routine to open the window.

The *wm_moved* routine is not very difficult either since the *wind_set* routine or the *wm_redraw* routine do most of the work. If changes are made to the window, the *modwind* routine is called. This routine checks to make sure that sliders, window size and other functions are within the current limits. The *WINDTAB* is also set to the new position.

The same goes for *wm_sized*.

The routine *wm_fulled* checks to see if the window is already full and then changes the window to the previous size. Variables *wf!(), wx%(), wy%(),wb%()* and *wh%()* are used for that purpose.

The *Modwind* routine is then called to change the position or size of the window with the help of variables *x%, y%, b%* and *h%*. After these parameters are passed to *GEM* (with *wind_set*), the new inner size of the window is inquired so that it can be matched with the coordinates (*grs0%* etc.) to make sure that the graphic or whatever does not overflow the window. The new values for *WINDTAB* and the size and position of the sliders are also updated during this routine.

The *wm_hslid* and the *wm_vslid* routines serve to set the size of the sliders to the overall window (like *txts0%,* etc.). There are different routines for each window. The *calc_slid* routine adjusts the sliders and *do_redraw* draws the new screen.

The same happens for all the arrows that were defined in the *openw* command. There can be up to eight arrow events per window. This *arrowx* routine calls the corresponding routine (*MENU(5)=0* to *MENU(5)=7*). The horizontal scroll size is determined by the size of the window -- one could have used a constant value of course. After the rows and columns are adjusted the new slider positions are set and the *Redraw*-routine is called. For a faster program you could test to see if the *Redraw* routine is even necessary or if part of the picture could be changed by using the *GET/PUT* or *BITBLT* routines. The *do_redraw* routine is now called upon to draw the top window according to the *wm_hslid*, the *wm_vslid* and the *wm_arrow* events.

The *Redraw*-routine is the most difficult. This routine (*wm_redraw*) controls the drawing of new screens caused by event function (like sliders). The routine is split into two parts, *wm_redraw* and *xredraw,* to simplify the slider and arrow events.

The rectangle for the corresponding window is then created. *Wind_get(...,11)* selects the first rectangle in the list and *Wind_get(...,12)* selects all the others. This step is repeated until the width *(DPEEK(gintout+6))* and the height *(DPEEK)gintout+8))* return a null to indicate the end of the list.

The screen segment for every one of those rectangles must be redrawn. Variables *tb%, th%, tx%* or *ty%* are used for that purpose. If the width *(tb%)* and the height *(th%)* is greater than zero the corresponding segment is redrawn.

The routine *Redraw* must now restore this segment. A *clipping* rectangle is then defined and the origins are set to point to the upper left corner. After erasing this segment by using a white *PBOX*, a specific routine is called to recreate the window.

Redrawing the Text Window is very simple. After calculating the number of visible text lines (window height divided by text height plus 2), the column offset is computed (how much the text must be moved to match the window). A vertical offset is then computed so that the first line of the text is visible in the window. Only a few of the lines are released to the *redraw* routine, though it would have been possible to select all lines.

That, however, would have been much slower and could cause a problem because of the 32000 offset range limit. Therefore only those lines which will fit in the visible window are submitted. After every text the vertical position is incremented by the height of the text. If you use a different font, this size may have to be adjusted.

The *Redraw* routine for the window 2 is even simpler. The origin for the graphic commands are set and the boxes are drawn.

The *Redraw* for the *Resource* window is handled by GEM. The origin must be changed in the *Resource* - this is accomplished with the *LPOKE* command. The objects are then drawn on the screen by using *obj_draw*.

For the graphic windows the *BITBLT* command is used to simply copy the picture segment into the window.

It is also possible to pass a different graphic resolution to the *BITBLT* (like 1280*1600 dots=256 KByte), to work on the window in smaller sections and then copy the results back using *BITBLT*. It is also possible to use more than one window for the picture, or to use more than one picture per window, or create an art clipboard (like the function key symbols), or create a window in a loop. While drawing you must use the *Button*-routine and you must update the window contents by using *wind_update(1)* and *wind_update(0)*. The graphic program could also be supplemented with an editor. If the button is pressed on a window, the graphic picture, or the symbol, is modified. With graphic it makes sense to make sure that window boundaries are not exceeded by using the *MENU OBOX* command (also see the explanation of *wind_update* in chapter 5). Let's continue with the program...

The *calc_slide* routine determines the inner size of the window and calls the routine that calculates the slider position and size and passes those parameters to the *set_slid* routine.

The *set_slid* routine changes all four slider positions (GEM expects an integer between 0 and 1000). Those values are then rounded.

The *procedure reset* serves to protect you while the program is being tested. The desktop is restored, all memory that was taken up for the *RSC* is freed, the menu is deactivated and all of the windows are closed. If the program was started from the desktop, it will return to the desktop otherwise it is accomplished automatically with the *QUIT* or *SYSTEM* command.

The *procedure openw* is an extension of the *OPENW* command. With it, the border elements may be defined and the window may be freely positioned.

The *procedure clip* fits the rectangle into the window and sets the corner point for further graphics commands.

The rest of the program creates an interface for the corresponding *GEM*-Routines.

The last routine selects the text size. This could have been handled by *GEMSYS 77 (graf_handle),* but this routine does not always seem to work properly. The *graf_handle* routine is called and the handle is used as a parameter to the corresponding *VDI* call. That's all.

Final word: The *redraw* was difficult but not impossible to solve. Even professional programmers cannot always perform miracles, but they will use those routines that will create the right effect.

Unfortunately, *GEM* does not have internal buffers for window content (even so this would easily be possible in a megabyte of memory), but it puts all of the responsibility of creating orderly windows in the hands of the programmers. It would have been helpful if it at least gave a message whenever a segment or an accessory was called.

As a last reminder: It makes sense to put the *PBOX* command in the *Redraw* routine for erasing a screen segment in procedures redraw1 and *redraw2*. With this the unnecessary erasing of screen segments is eliminated, because the *BITBLT* command or the *obj_draw* call overwrites the contents of the background anyway.

Using *GEM*, it is only possible to use programs that always know the content of that window.

```
' WINDOW.BAS
'
If Xbios(4)<>2
    Alert 1,"This Demo runs only in Hi-rez",1,"Cancel",Dummy%
    End
Endif
'
Dim Wf!(4),Wx%(4),Wy%(4),Wb%(4),Wh%(4)
@Reset
Reserve Xbios(2)-Himem+Fre(0)-16384-5000
@Rsrc_load("wind.rsc")
@Rsrc_gtree(0,*Deskrsc%)
@Rsrc_gtree(1,*Windrsc%)
@Wind_get(0,4)                          ! get desk size
Bmove Gintout+2,Deskrsc%+16,8           ! set into rsc
@Wind_newdesk(Deskrsc%,0)               ! install
'
@Get_textsize
Chrb%=Dpeek(Ptsout)                     ! Text width
Chrh%=Dpeek(Ptsout+2)                   ! text height
Chrbb%=Dpeek(Ptsout+4)                  ! Text box width
Chrbh%=Dpeek(Ptsout+6)                  ! Text box height
'
@Wind_get(0,4)                          ! maximum Window parameter
Scrx%=Dpeek(Gintout+2)
Scry%=Dpeek(Gintout+4)
Scrb%=Dpeek(Gintout+6)
Scrh%=Dpeek(Gintout+8)
'
' Initialize Menu bar
'
Dim M$(50)
For I%=0 To 50
    Read M$(I%)
    Exit If M$(I%)="****"
Next I%
M$(I%)=""
M$(0)=Chr$(14)+Chr$(15)             ! The Atari symbol
Menu M$()
Erase M$()
```

```
'
Data Desk,    Window Demo,-----------------------,1,2,3,4,5,6,
Data File,    Load,        Save,--------,     Quit,***
'
' Initialize Window 1
'
Dim Txt$(99)
For I%=0 To 99
    Txt$(I%)="Text Line "+Str$(I%)+" ----------- "+Str$(I%)
Next I%
Txtz0%=0
Txts0%=0
Titlew 1,"Text Window"
Infow 1,""
@Openw(1,&HFFFF,50,100,150,180)
@Calc_slid(1)
'
' Init window 2
'
Grs0%=0
Grz0%=0
Titlew 2,"Graphic window"
Infow 2,""
@Openw(2,&HFFFF,110,25,170,190)
@Calc_slid(2)
'
' Init window 3
'
Titlew 3,"Resource window"
Infow 3,""
@Wind_calc(0,3,0,0,Dpeek(Windrsc%+20),Dpeek(Windrsc%+22))
@Openw(3,&HB,250,80,Dpeek(Gintout+6),Dpeek(Gintout+8))
'
'
'
Dim Smfdb%(8),Dmfdb%(8),P%(8)
Open "I",#1,"WOOF1.PI3"
Seek #1,34
X$=Input$(32000,#1)
Close #1
```

```
Dmfdb%(0)=Xbios(3)
Dmfdb%(1)=640
Dmfdb%(2)=400
Dmfdb%(3)=40
Dmfdb%(5)=1
Smfdb%(1)=640
Smfdb%(2)=400
Smfdb%(3)=40
Smfdb%(5)=1
'
Titlew 4,"Woof"
Infow 4,"a Degas picture"
@Openw(4,&HFFF,150,150,250,225)
'
On Menu Message Gosub Message
On Menu Button 2,1,1 Gosub Button
On Menu       Gosub Menu
On Menu Key Gosub Key
'
@Form_dial(3,0,0,0,0,0,0,640,400)                    ! redraw all
'
Repeat
    On Menu
Until End!
'
' All done QUIT
'
@Reset
@Wind_get(0,10)
If Dpeek(Gintout+2)
    Alert 1,"Accessories",1,"Close•Quit",X%
    If X%=2
        Quit
    Endif
    Repeat
        @Wind_get(0,10)
    Until Dpeek(Gintout+2)=0
Endif
Reserve Xbios(2)-Himem+Fre(0)-16384
'
```

```
Procedure Menu
  If Menu(0)=1
    Alert 1,"This is an example of•Window technics",1,
                              "GFA•BASIC",Dummy%
  Endif
  If Menu(0)=14
    Let End!=True
    @Reset
  Endif
  Menu Off
Return
'
Procedure Key
  A%=Menu(14) Div 256-58
  If (Menu(14) And 255)=27
    @Wind_update(3)
  Endif
  If (Menu(14) And 255)=13
    @Wind_update(2)
  Endif
  If A%>0 And A%<11
    @Desk_change(A%)
  Endif
Return
'
Procedure Button
  @Wind_find(Menu(10),Menu(11))
  If Dpeek(Gintout=0)
    @Objc_find(Deskrsc%,0,1,Menu(10),Menu(11))
    O%=Dpeek(Gintout)
    If O%>0 And O%<1000
      @Desk_change(O%)
    Endif
  Endif
Return
'
Procedure Desk_change(Nr%)
  Adr%=Deskrsc%+24*Nr%+10
  Ostate%=Dpeek(Adr%)                        ! alter status
  Dpoke Adr%,Ostate% Xor 1
```

```
    @Wind_get(0,11)
    B%=Dpeek(Gintout+6)
    H%=Dpeek(Gintout+8)
    While B% Or H%
        @Objc_draw(Deskrsc%,Nr%,7,Dpeek(Gintout+2),Dpeek(Gintout+4),
B%,H%)
        @Wind_get(0,12)
        B%=Dpeek(Gintout+6)
        H%=Dpeek(Gintout+8)
    Wend
Return
'
Procedure Message
    If Menu(1)>19 And Menu(1)<29        !wm_xxxxx
        Hand%=Menu(4)
        If Hand%=Dpeek(Windtab)          ! this way is best for compiling
            Wind%=1
        Else
            If Hand%=Dpeek(Windtab+12)
                Wind%=2
            Else
                If Hand%=Dpeek(Windtab+24)
                    Wind%=3
                Else
                    If Hand%=Dpeek(Windtab+36)
                        Wind%=4
                    Else
                        Wind%=0
                    Endif
                Endif
            Endif
        Endif
        On Menu(1)-19 Gosub
Wm_redraw,Wm_topped,Wm_closed,Wm_fulled,

Wm_arrowed
        On Menu(1)-24 Gosub Wm_hslid,Wm_vslid,Wm_sized,Wm_moved
    Else
        ' Unknown
    Endif
```

```
Return
'
'
Procedure Wm_closed
   Closew Wind%
Return
'
Procedure Wm_topped
   Openw Wind%
Return
'
Procedure Wm_moved
   Adr%=Windtab+12*Wind%-12
   Dpoke Adr%+4,Menu(5)
   Dpoke Adr%+6,Menu(6)
   @Modwind(Wind%,Menu(5),Menu(6),Menu(7),Menu(8))
Return
'
Procedure Wm_sized
   Adr%=Windtab+12*Wind%-12
   Dpoke Adr%+8,Menu(7)
   Dpoke Adr%+10,Menu(8)
   @Modwind(Wind%,Menu(5),Menu(6),Menu(7),Menu(8))
   @Calc_slid(Wind%)
   Wf!(Wind%)=False
Return
'
Procedure Wm_fulled
   Adr%=Windtab+12*Wind%-12
   If Wf!(Wind%)          !already big
        X%=Wx%(Wind%)
        Y%=Wy%(Wind%)
        B%=Wb%(Wind%)
        H%=Wh%(Wind%)
        Wf!(Wind%)=False
   Else
        @Wind_get(Hand%,5)
        Wx%(Wind%)=Dpeek(Gintout+2)
        Wy%(Wind%)=Dpeek(Gintout+4)
        Wb%(Wind%)=Dpeek(Gintout+6)
```

```
        Wh%(Wind%)=Dpeek(Gintout+8)
        X%=Scrx%
        Y%=Scry%
        B%=Scrb%
        H%=Scrh%
        Wf!(Wind%)=True
    Endif
    Dpoke Adr%+4,X%
    Dpoke Adr%+6,Y%
    Dpoke Adr%+8,B%
    Dpoke Adr%+10,H%
    @Modwind(Wind%,X%,Y%,B%,H%)
Return
'
' This routine is called to expand or change the position
' of a window. Here it is possible to put the window on
' a Byte boundary, to set a maximum ansd minimum size,
' to hold a complete window on the screen at all times,
' and to match the slide bars according to size.
'
Procedure Modwind(Wind%,X%,Y%,B%,H%)
    On Wind% Gosub Modw1,Modw2,Modw3,Modw4
    @Wind_set(Hand%,5,X%,Y%,B%,H%)
    @Wind_get(Hand%,4)
    On Wind% Gosub Mods1,Mods2,Mods3,Mods4
    @Calc_slid(Wind%)
Return
Procedure Modw1
Return
Procedure Modw2
    X%=X%+4 And &HFFF8        !Only in 8 steps movable
Return
Procedure Modw3
Return
Procedure Modw4
Return
Procedure Mods1
    Txts0%=Min(Txts0%,80-Dpeek(Gintout+6)/Chrbb%)
    Txtz0%=Min(Txtz0%,100-Dpeek(Gintout+8)/Chrbh%)
Return
```

```
Procedure Mods2
    Grs0%=Min(Grs0%,1280-Dpeek(Gintout+6))
    Grz0%=Min(Grz0%,800-Dpeek(Gintout+8))
Return
Procedure Mods3
Return
Procedure Mods4
    Pais0%=Min(Pais0%,640-Dpeek(Gintout+6))
    Paiz0%=Min(Paiz0%,400-Dpeek(Gintout+8))
Return
'
Procedure Wm_hslid
    @Wind_get(Wind%,4)
    B%=Dpeek(Gintout+6)
    On Wind% Gosub Hslid1,Hslid2,Hslid3,Hslid4
    @Calc_slid(Wind%)
    @Do_redraw
Return
'
Procedure Hslid1
    Txts0%=Menu(5)*(80-B%/Chrbb%)/1000+0.5
Return
'
Procedure Hslid2
    Grs0%=Menu(5)*(1280-B%)/1000+0.5
Return
'
Procedure Hslid4
    Pais0%=Menu(5)*(640-B%)/1000+0.5
Return
'
Procedure Wm_vslid
    @Wind_get(Wind%,4)
    H%=Dpeek(Gintout+8)
    On Wind% Gosub Vslid1,Vslid2,Vslid3,Vslid4
    @Calc_slid(Wind%)
    @Do_redraw
Return
'
Procedure Vslid1
```

```
    Txtz0%=Menu(5)*(100-H%/Chrbh%)/1000+0.5        !max 100 Lines
Return
'
Procedure Vslid2
    Grz0%=Menu(5)*(800-H%)/1000+0.5
Return
'
Procedure Vslid4
    Paiz0%=Menu(5)*(400-H%)/1000+0.5
Return
'
Procedure Wm_arrowed
    @Wind_get(Wind%,4)
    B%=Dpeek(Gintout+6)
    H%=Dpeek(Gintout+8)
    On Wind% Gosub Arrow1,Arrow2,Arrow3,Arrow4
    @Calc_slid(Wind%)
    @Do_redraw
Return
'
Procedure Arrow1
    On Menu(5)+1 Gosub 1pu,1pd,1lu,1ld,1pl,1pr,1ll,1lr
Return
Procedure 1pu
    Txtz0%=Max(Txtz0%-H%/Chrbh%,0)
Return
Procedure 1pd
    Txtz0%=Min(Txtz0%+H%/Chrbh%,100-H%/Chrbh%)
Return
Procedure 1lu
    Txtz0%=Max(Txtz0%-1,0)
Return
Procedure 1ld
    Txtz0%=Min(Txtz0%+1,100-H%/Chrbh%)
Return
Procedure 1pl
    Txts0%=Max(Txts0%-B%/Chrbb%,0)
Return
Procedure 1pr
    Txts0%=Min(Txts0%+B%/Chrbb%,80-B%/Chrbb%)
```

```
Return
Procedure 1ll
    Txts0%=Max(Txts0%-1,0)
Return
Procedure 1lr
    Txts0%=Min(Txts0%+1,80-B%/Chrbb%)
Return
'
Procedure Arrow2
    On Menu(5)+1 Gosub 2pu,2pd,2lu,2ld,2pl,2pr,2ll,2lr
Return
Procedure 2pu
    Grz0%=Max(Grz0%-H%,0)
Return
Procedure 2pd
    Grz0%=Min(Grz0%+H%,800-H%)
Return
Procedure 2lu
    Grz0%=Max(Grz0%-10,0)
Return
Procedure 2ld
    Grz0%=Min(Grz0%+10,800-H%)
Return
Procedure 2pl
    Grs0%=Max(Grs0%-B%,0)
Return
Procedure 2pr
    Grs0%=Min(Grs0%+B%,1280-B%)
Return
Procedure 2ll
    Grs0%=Max(Grs0%-10,0)
Return
Procedure 2lr
    Grs0%=Min(Grs0%+10,1280-B%)
Return
'
Procedure Arrow4
    On Menu(5)+1 Gosub 4pu,4pd,4lu,4ld,4pl,4pr,4ll,4lr
Return
Procedure 4pu
```

```
    Paiz0%=Max(Paiz0%-H%,0)
Return
Procedure 4pd
    Paiz0%=Min(Paiz0%+H%,400-H%)
Return
Procedure 4lu
    Paiz0%=Max(Paiz0%-10,0)
Return
Procedure 4ld
    Paiz0%=Min(Paiz0%+10,400-H%)
Return
Procedure 4pl
    Pais0%=Max(Pais0%-B%,0)
Return
Procedure 4pr
    Pais0%=Min(Pais0%+B%,640-B%)
Return
Procedure 4ll
    Pais0%=Max(Pais0%-10,0)
Return
Procedure 4lr
    Pais0%=Min(Pais0%+10,640-B%)
Return
'
Procedure Do_redraw
    @Wind_get(Hand%,4)
    @Xredraw(Dpeek(Gintout+2),Dpeek(Gintout+4),Dpeek(Gintout+6),
                                                  Dpeek(Gintout+8))
Return
'
Procedure Wm_redraw
    @Xredraw(Menu(5),Menu(6),Menu(7),Menu(8))
Return
Procedure Xredraw(M5%,M6%,M7%,M8%)
    @Wind_update(1)
    @Wind_get(Hand%,11)
    While Lpeek(Gintout+6)  !width or height <>0
        Tb%=Dpeek(Gintout+2)+Dpeek(Gintout+6)
        Th%=Dpeek(Gintout+4)+Dpeek(Gintout+8)
        Tx%=Max(Dpeek(Gintout+2),M5%)
```

```
            Ty%=Max(Dpeek(Gintout+4),M6%)
            Tb%=Min(Tb%,M5%+M7%)-Tx%
            Th%=Min(Th%,M6%+M8%)-Ty%
            If Tb%>0
                If Th%>0
                    @Redraw(Wind%,Tx%,Ty%,Tb%,Th%)
                Endif
            Endif
            @Wind_get(Hand%,12)
        Wend
        @Wind_update(0)
    Return
    '
    Procedure Redraw(Wind%,X%,Y%,B%,H%)
        @Wind_get(Hand%,4)
        @Clip(X%,Y%,B%,H%,Dpeek(Gintout+2),Dpeek(Gintout+4))
        Graphmode 0
        Deffill ,0
        '       PBOX -99,-99,999,999              !clear box
        ' moved, otherwise there will be flickering
        On Wind% Gosub Redraw1,Redraw2,Redraw3,Redraw4
    Return
    '
    Procedure Redraw1
        Pbox -99,-99,999,999
        Deftext 1,0,0,Chrh%,1                     !Initial deftext
        Anz%=Dpeek(Gintout+8)/Chrbh%+2
        X%=-Txts0%*Chrbb%                         !split offset
        Y%=Chrh%-Chrbh%
        Q%=Txtz0%
        For I%=0 To Anz%
            Add Y%,Chrbh%
            Exit If Q%>99
            Text X%,Y%,Txt$(Q%)
            Inc Q%
        Next I%
    Return
    '
    Procedure Redraw2
        Pbox -99,-99,999,999
```

```
    Defline 1,1,0,0
    Dpoke Windtab+64,Dpeek(Windtab+64)-Grs0%
    Dpoke Windtab+66,Dpeek(Windtab+66)-Grz0%
    For I%=0 To 1279 Step 16
        Box I%,I%,1279-I%,799-I%
    Next I%
Return
'
Procedure Redraw3
    Lpoke Windrsc%+16,Lpeek(Gintout+2)
    @Objc_draw(Windrsc%,0,7,X%,Y%,B%,H%)
Return
'
Procedure Redraw4
    P%(0)=Pais0%+X%-Dpeek(Gintout+2)
    P%(1)=Paiz0%+Y%-Dpeek(Gintout+4)
    P%(2)=P%(0)+B%-1
    P%(3)=P%(1)+H%-1
    P%(4)=X%
    P%(5)=Y%
    P%(6)=X%+B%-1
    P%(7)=Y%+H%-1
    P%(8)=3
    Smfdb%(0)=Varptr(X$)
    Bitblt Smfdb%(),Dmfdb%(),P%()
Return
'
'
'
'
'
'
Procedure Calc_slid(Wind%)
    Hand%=Dpeek(Windtab+12*Wind%-12)
    @Wind_get(Hand%,4)
    B%=Dpeek(Gintout+6)
    H%=Dpeek(Gintout+8)
    On Wind% Gosub Cslid1,Cslid2,Cslid3,Cslid4
Return
'
```

```
'
Procedure Cslid1
    Hp=Txts0%/(80-B%/Chrbb%)
    Vp=Txtz0%/(100-H%/Chrbh%)
    @Set_slid(Hand%,B%/80/Chrbb%,H%/100/Chrbh%,Hp,Vp)
Return
'
Procedure Cslid2
    @Set_slid(Hand%,B%/1280,H%/800,Grs0%/(1280-B%),Grz0%/(800-H%))
Return
'
Procedure Cslid3
Return
'
Procedure Cslid4
    @Set_slid(Hand%,B%/640,H%/400,Pais0%/(640-B%),Paiz0%/(400-H%))
Return
'
Procedure Set_slid(Hand%,Hs,Vs,Hp,Vp)
    @Wind_set(Hand%,15,Hs*1000+0.5,0,0,0)
    @Wind_set(Hand%,16,Vs*1000+0.5,0,0,0)
    @Wind_set(Hand%,8,Hp*1000+0.5,0,0,0)
    @Wind_set(Hand%,9,Vp*1000+0.5,0,0,0)
Return
'
Procedure Reset
    @Wind_olddesk
    Gemsys 111
    Menu Kill
    For I%=4 Downto 0
        Closew I%
    Next I%
Return
'
'
Procedure Openw(Nr%,Attr%,X%,Y%,B%,H%)
    Local Adr%
    Adr%=Windtab+12*Nr%-12
    Dpoke Adr%+2,Attr%
    Dpoke Adr%+4,X%
```

```
    Dpoke Adr%+6,Y%
    Dpoke Adr%+8,B%
    Dpoke Adr%+10,H%
    Openw Nr%
Return
'
Procedure Clip(X%,Y%,B%,H%,X0%,Y0%)
    Dpoke Ptsin,X%
    Dpoke Ptsin+2,Y%
    Dpoke Ptsin+4,X%+B%-1
    Dpoke Ptsin+6,Y%+H%-1
    Dpoke Intin,1
    Dpoke Contrl+2,2
    Dpoke Contrl+6,1
    Vdisys 129
    Dpoke Windtab+64,X0%
    Dpoke Windtab+66,Y0%
Return
'
' GEMSYS Routines
'
Procedure Objc_draw(Tree%,Start%,Depth%,X%,Y%,B%,H%)
    Lpoke Addrin,Tree%
    Dpoke Gintin,Start%
    Dpoke Gintin+2,Depth%
    Dpoke Gintin+4,X%
    Dpoke Gintin+6,Y%
    Dpoke Gintin+8,B%
    Dpoke Gintin+10,H%
    Gemsys 42
Return
Procedure Objc_find(Tree%,Start%,Depth%,X%,Y%)
    Lpoke Addrin,Tree%
    Dpoke Gintin,Start%
    Dpoke Gintin+2,Depth%
    Dpoke Gintin+4,X%
    Dpoke Gintin+6,Y%
    Gemsys 43
Return
Procedure Objc_change(Tree%,Obj%,X%,Y%,B%,H%,Neu%,Flg%)
```

```
    Lpoke Addrin,Tree%
    Dpoke Gintin,Obj%
    Dpoke Gintin+2,0                                    !reserved
    Dpoke Gintin+4,X%
    Dpoke Gintin+6,Y%
    Dpoke Gintin+8,B%
    Dpoke Gintin+10,H%
    Dpoke Gintin+12,Neu%
    Dpoke Gintin+14,Flg%
    Gemsys 47
Return
Procedure Form_dial(F%,X%,Y%,B%,H%,Xb%,Yb%,Bb%,Hb%)
    Dpoke Gintin,F%
    Dpoke Gintin+2,X%
    Dpoke Gintin+4,Y%
    Dpoke Gintin+6,B%
    Dpoke Gintin+8,H%
    Dpoke Gintin+10,Xb%
    Dpoke Gintin+12,Yb%
    Dpoke Gintin+14,Bb%
    Dpoke Gintin+16,Hb%
    Gemsys 51
Return
Procedure Rsrc_load(Nam$)
    Nam$=Nam$+Chr$(0)
    Lpoke Addrin,Varptr(Nam$)
    Gemsys 110
Return
Procedure Rsrc_free
    Gemsys 111
Return
Procedure Rsrc_gaddr(Type%,Index%)
    Dpoke Gintin,Type%
    Dpoke Gintin+2,Index%
    Gemsys 112
Return
Procedure Rsrc_gtree(Index_%,Tree.%)
    Lpoke Gintin,Index_%
    Gemsys 112
    *Tree.%=Lpeek(Addrout)
```

```
Return
Procedure Wind_get(H%,F%)
    Dpoke Gintin,H%
    Dpoke Gintin+2,F%
    Gemsys 104
Return
Procedure Wind_set(H%,F%,A1%,A2%,A3%,A4%)
    Dpoke Gintin,H%
    Dpoke Gintin+2,F%
    Dpoke Gintin+4,A1%
    Dpoke Gintin+6,A2%
    Dpoke Gintin+8,A3%
    Dpoke Gintin+10,A4%
    Gemsys 105
Return
Procedure Wind_find(X%,Y%)
    Dpoke Gintin,X%
    Dpoke Gintin+2,Y%
    Gemsys 106
Return
Procedure Wind_calc(F%,Attr%,X%,Y%,B%,H%)
    Dpoke Gintin,F%
    Dpoke Gintin+2,Attr%
    Dpoke Gintin+4,X%
    Dpoke Gintin+6,Y%
    Dpoke Gintin+8,B%
    Dpoke Gintin+10,H%
    Gemsys 108
Return
Procedure Wind_update(Flg%)
    Dpoke Gintin,Flg%
    Gemsys 107
Return
Procedure Wind_newdesk(Tree%,Start%)
    Lpoke Gintin,14
    Lpoke Gintin+4,Tree%
    Dpoke Gintin+8,Start%
    Gemsys 105
Return
Procedure Wind_olddesk
```

```
    @Wind_newdesk(0,0)
Return
'
Procedure Get_textsize                      ! creates normal text size
    V%=Dpeek(Contrl+12)                     ! gemsys 77 should do it
    Gemsys 77                               ! but I have not had much luck
    Dpoke Contrl+12,Dpeek(Gintout)          ! with it.
    Vdisys 38                               ! Out: (in ptsout)
    Dpoke Contrl+12,V%                      ! h/h (Symbol) b/h (box)
Return
```

APPENDICES

APPENDIX A: BIOS

BIOS(0,L:ptr%)	getmpb	144
BIOS(1,d%)	bconstat	144
BIOS(2,d%)	bconin	144
BIOS(3,d%,c%)	bconout	144
BIOS(4,f%,L:buf%,n%,rec%,d%)	rwabs	144
BIOS(5,n%,L:adr%	setexec	145
BIOS(6)	tickcal	145
BIOS(7,d%)	getbpb	145
BIOS(8,d%)	bcostat	145
BIOS(9,d%)	mediach	145
BIOS(10)	drvmap	146
BIOS(11,x%)	kbshift	146

APPENDIX B: XBIOS

XBIOS(0,t%,L:par%,L:vec%)	initmous	147
XBIOS(1,n%)	ssbrk	147
XBIOS(2)	physbase	147
XBIOS(3)	logbase	147
XBIOS(4)	getrez	147
XBIOS(5,L:l%,L:p%,r%)	setscreen	148
XBIOS(6,L:adr%)	setpallete	148
XBIOS(7,n%,c%)	setcolor	148
XBIOS(8,L:a%,L:0,d%,s%,t%,si%,n%)	floprd	148
XBIOS(9,L:a%,L:0,d%,s%,t%,si%,n%)	flopfnt	148
XBIOS(10,L:a%,L:0,d%,s%,t%,	flopwr	148
XBIOS(11)	getdsb	149
XBIOS(12,n%,L:a%)	midiws	149
XBIOS(13,n%,L:v%)	mfpint	149
XBIOS(14,d%)	iorec	150
XBIOS(15,b%,f%,u%,r%,t%,s%)	rsconf	151
XBIOS(16,Lu%,L:s%,L:c%)	keytbl	152
XBIOS(17)	random	153
XBIOS(18,La%,L:0,L:s%,t%,f%)	protobt	153
XBIOS(19,L:a%,L:0,d%,s%,t%,si%,n%)	flopver	154
XBIOS(20)	scrdmp	154
XBIOS(21,a%,r%)	curscon	154
XBIOS(22,L:dt%)	bsettime	155
XBIOS(23)	bgettime	155
XBIOS(24)	bioskeys	155
XBIOS(25,n%,L:a%)	ikbdws	155

XBIOS(26,n%)	jdisint	155
XBIOS(27,n%)	jenabin	155
XBIOS(28,c%,n%)	giacces	155
XBIOS(29,m%)	offgibit	155
XBIOS(30,m%)	ongibit	156
XBIOS(31,n%,c%,d%. L:vec%)	xbtimer	156
XBIOS(32,L:adr%)	dosound	158
XBIOS(33,m%)	setprt	160
XBIOS(34)	kbdvbas	160
XBIOS(35,d%,r%)	kbrate	163
XBIOS(36,L:pointer)	prtblk	163
XBIOS(37)	vsync	163
XBIOS(38,L:vec%)	superx	163
XBIOS(39)	pntaes	163

APPENDIX C: GEMDOS

GEMDOS(0)	p_termold	135
GEMDOS(1)	c_conin	135
GEMDOS(2,c%)	c_conout	136
GEMDOS(3)	c_auxin	136
GEMDOS(4,c%)	c_auxout	136
GEMDOS(5,c%)	c_prnout	136
GEMDOS(6,c%)	c_rawio	136
GEMDOS(7)	c_rawcin	136
GEMDOS(8)	c_necin	136
GEMDOS(9,L:adr%)	c_conws	137
GEMDOS(10,L:adr%)	c_conrs	137
GEMDOS(11)	c_conis	137
GEMDOS(14,d%)	d_setdrv	137
GEMDOS(16)	c_conos	137
GEMDOS(17)	c_prnos	138
GEMDOS(18)	c_auxis	138
GEMDOS(19)	c_auxos	138
GEMDOS(25)	c_getdrv	138
GEMDOS(26L:adr%)	f_setdta	138
GEMDOS(42)	t_getdate	138
GEMDOS(43,d%)	t_setdate	139
GEMDOS(44)	t_gettime	139
GEMDOS(45,t%)	t_settime	139
GEMDOS(47)	f_getdta	139
GEMDOS(48)	s_version	139
GEMDOS(49,L:size%,ret%)	p_termres	139

GEMDOS(54,L:adr%,d%)	d_free	139
GEMDOS(57,L:adr%)	d_create	140
GEMDOS(58,L:adr%)	d_delete	140
GEMDOS(59,L:adr%)	d_setpat	140
GEMDOS(60,L:adr%,attr%)	f_create	140
GEMDOS(61,L:adr%,mode%)	f_open	140
GEMDOS(62,h%)	f_close	140
GEMDOS(63,h%,L:len%,L:adr%)	f_read	141
GEMDOS(64,h%,L:len%,L:adr%)	f_write	141
GEMDOS(65,L:adr%)	f_delete	141
GEMDOS(66,L:n%,h%,mode%)	f_seek	141
GEMDOS(67,L:adr%,flg%,attr%)	f_attrib	141
GEMDOS(69,h%)	f_dup	142
GEMDOS(70,n%,s%)	f_force	142
GEMDOS(71,L:adr%,d%)	d_getpath	142
GEMDOS(72,L:size%)	m_malloc	142
GEMDOS(73,L:adr%)	m_free	142
GEMDOS(74,0,L:adr%,L:size%)	m_shrink	142
GEMDOS(75,f%,L:nam%,L:cmd%,L:env%)	p_exec	143
GEMDOS(76,ret%)	p_term	143
GEMDOS(78,L:nam%,attr%)	f_sfirst	143
GEMDOS(79)	f_snext	143
GEMDOS(86,0,L:old%,L:neu%)	f_rename	143
GEMDOS(87,L:tdbuf%,h%,flg%)	f_datime	143

APPENDIX D: GEMSYS

INDEX

Index

GFA BASIC BOOK

by GFA Systemtechnik

You've seen the program, now it's time to read the Book: The GFA BASIC BOOK. The Book that will transform you into the programmer you were meant to be. This easy to comprehend intermediate manual, written by Frank Ostrowski – author of the GFA BASIC Interpreter, is designed for the person with a casual knowledge of BASIC who wants to gain an understanding of the more complicated aspects of the programmers art. It guides you step by step through many intricate facets of BASIC and GEM programming.

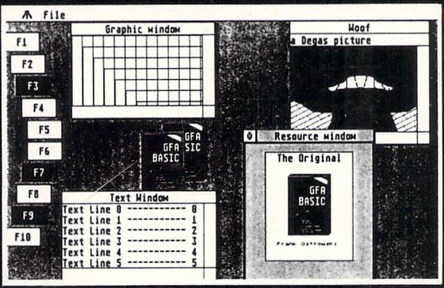
Best of all, there is a disk enclosed with over 75 programs and files. Virtually every program and principle discussed in the manual is present as .LST or .BAS files, so you will not have to engage in lot of bothersome typing before you see results. Many of these files and procedures can be imbedded directly into your own programs.

Chapter topics include:

OPTIMIZATION: a list of routines that will make your programs run faster and more efficiently.

GRAPHICS: which contains sections on Clipping, Raster Graphics Commands, Flicker Free Graphics, and more.

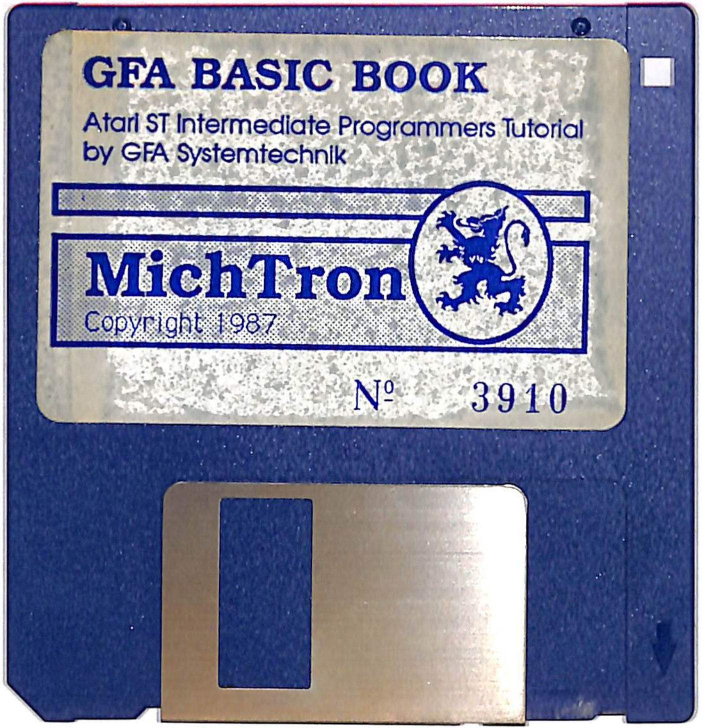
TIPS & PROGRAMS: a potpourri of features dealing with everything from Sound and Magnify techniques, to Procedures for Copying files, Formatting disks, and obtaining Directories from within BASIC programs, as well as the use of Recursion, Scan Codes, different Fonts and the EXEC command.



Design multiple Windows easily: Just one of many programs included

This brief summary doesn't begin to cover the wealth of tips and information you'll find in the pages of this book, and on the accompanying disk. It is an essential teaching guide and a remarkable reference for anyone who uses GFA BASIC – or plans to, and who wants to do so with more skill and expertise.

MichTron
576 S. Telegraph, Pontiac, MI 48053
Orders and Information (313) 334-5700


GFA BASIC BOOK
Atari ST Intermediate Programmers Tutorial
by GFA Systemtechnik
MichTron
Copyright 1987
№ 3910


GFA BASIC 3.0
The Atari ST Programming Language
by GFA Systemtechnik
MichTron
Copyright 1988
№ 4611