ATARI ST

GFA BASIC Version 3

Interpreter

GFA BASIC
Version 3

User Manual

GFA Software Technologies, Inc.

July 1988

Warranty

All programs in this manual have written expressly to illustrate specific training points, they are not warranted as being suitable for any particular application. Every care has been taken in the writting and presentation of this manual but no responsibility is assumed by the author or publishers for any errors or omissions contained herein or any consequential loss suffered therefrom.

ISBN 1 85181 186 9

Published in the U.S. by:

GFA Software Technologies, Inc.
27 Congress St.
Salem, MA 01970

CONTENTS

CHAPTER 1
INTRODUCTION

With the release of GFA BASIC 3 is provided with both an extensive programming and powerful language which is easy to use via its quick and friendly Editor. The entire framework and design of the language encourages structured programming and provides the user with the necessary tools.

The Interpreter provides for simple debugging by means of the special commands provided. The Editor supports structured programming by automatic indentation of loops and conditions. Additionally, subroutines can be represented in the program listing by name only, and 'opened' to their full length by pressing a key.

In the area of conditional commands, various additions have been made to the commands IF-ELSE-ENDIF, which already existed in earlier versions of GFA BASIC, e.g. ELSE-IF, SELECT-CASE. Both the values of variables and the variables themselves can be passed to procedures. Furthermore, the loop types FOR-NEXT, REPEAT-UNTIL, WHILE-WEND and DO-LOOP available in earlier versions of GFA BASIC have been extended with DO-UNTIL, DO-WHILE, LOOP-UNTIL and LOOP-WHILE.

System level programming is achieved through the ability to call operating system routines (GEMDOS, BIOS, XBIOS). Many of these routines are also available in form of simple instructions. The programming of Interrupts with EVERY and AFTER is also possible. Assembler and C routines may be interleaved into programs with instructions like RCALL, C:MERGE and MONITOR. The most important VDI-routines and all AES-functions, e.g. menu and form management, are available as built-in functions.

GFA BASIC provides genuine Integer arithmetic, which offers high computing speed, as well as floating-point arithmetic with high computational accuracy - 13 decimal places, while further variable types - BYTE, WORD - and bit operations - BCLR, BSET, BTST, BCHG, SHL, SHR, ROL, ROR, etc. - have been included. Graphics programs can easily make use of the LINE-A routines which are implemented as instructions.

About This Manual

The manual begins with a short description of the GFA BASIC 3 programming language, followed by an explanation of the manual structure and an introduction to the use of the GFA BASIC Interpreter. It ends with a description of the points which one must take note of when using programs which were written in older versions of GFA BASIC.

The next chapter describes the commands for the operation of the Editor. The remaining chapters of the manual detail the instructions and facilities of GFA BASIC 3.0. These are arranged according to criteria concerning the contents, with joint explanations of related terms, e.g. MIN and MAX. In the appendix an alphabetical overview of the instructions and functions is given, with references to the appropriate pages for full explanation.

Descriptions of Commands, etc, are laid out as follows:

- The Syntax
- Description of the permissible parameter types
- Explanatory text
- Example

In the Syntax description optional parameters are indicated with square brackets, e.g.

```
LEFT$ (a$ [, x] )
```

In GFA BASIC there are Commands and Functions. Commands do not return a value, e.g.

```
LINE 100,100,200,200
```

Functions, on the other hand, do return a value and can display this value with PRINT, assign it to variables, etc. Here are some examples:

```
PRINT ASC("65")
PRINT ASC("a")
a=ASC("a")
b=ASC("a")+32
```

In this manual the fact that a Function returns a value is not noted in the Syntax section, but is made clear in the explanation. In the Syntax section, in the case of the function ASC for example, only ASC(a$) is specified and the fact that this would be assigned to a numeric variable is not stated.

Where many optional parameters may be given, e.g. with the DATA statement, this is shown by three full stops:

```
DATA [x, y, ...]
```

The description of permitted parameter types follows the Syntax section. For these types the following abbreviations are used:

avar Arithmetic variable

This a numeric variable, which can be of any form, e.g. floating point, integer, boolean, etc.

aexp Arithmetic expression

This is any simple or complex expression which produces a number and can include variables. Examples of arithmetic expressions are:

```
a%
3
2+a%+ASC("a")
```

svar String variable

This is a character string variable ending with $, e.g.

```
a$
```

sexp String expression

This expression can be of arbitrary complexity but must finally result in a String. Examples of string expressions are:

```
a$
"Test"
a$+"GFA BASIC"+LEFT$("MANUAL",4)
```

ivar Integer variable

This can be any integer variable.

iexp Integer expression

This is any simple or complex expression but the result must be an integer.

bexp Boolean (logical) expression

Any expression from which the result is a Boolean number.

It is important for some numerical expressions that particular variable types must be used. The most important example of this is with memory addresses. Addresses must be specified as at least a four byte variable and Boolean, Byte or Word variables are thus not allowed.

After the description of the parameter types comes the explanation of the command. Here the meaning of the command and its individual parameters are explained. The discussion of a command is terminated with one or more examples. These examples are designed to be entered into the Editor and started with RUN (shift+F10 or clicking RUN in the menu bar). After each example the effect it produces is given.

This form of command description is modified only in the section dealing with the AES library routines. Here the name of the appropriate command is given, followed by the explanation of its function, then the syntax of the command with the description of the individual parameters. At the end of the chapter concerning the libraries several longer example programs are provided. The reasoning behind this structure is that many commands in the chapter must be used in correct conjunction with other commands, so that example programs tend to contain many AES calls and tend to be somewhat more extensive.

The manual concludes with a collection of tables and an alphabetical list of all commands, with their page numbers.

NOTE: Where a program line is longer than the width of the page it has been wrapped round on the page and right justified. This will not happen when you are typing in the line, it will just extend to the right of the screen as far as necessasary (up to 255 characters).

Using GFA BASIC 3 For The First Time

This section is for those who have not used GFA BASIC before. Those who have already worked with the older versions can just scan this section or, if feeling very confident, jump to Chapter 2.

The GFA BASIC 3 Program disk is not copy-protected, so first make a back-up copy of the original disk. You will find a description of how to do this in your computer manual. Put the copy of the program disk in your disk drive and start GFA BASIC by clicking on the icon.

After a short while, the Editor screen will appear, which is where you can write and debug your programs. For now, enter the following program lines, pressing RETURN at the end of each line. Do not omit the spaces, which serve to separate adjacent keywords. The indenting of instructions within loops is done automatically by the Editor, as is the capitalisation (or lack of it) when RETURN is pressed.

```
DEFFILL 1,2,8
REPEAT
  WHILE MOUSEK=1
    PBOX MOUSEX,MOUSEY,MOUSEX+30,MOUSEY+30
  WEND
UNTIL MOUSEK=2
```

In the upper right corner of the screen you will find the word 'Run'. Point with the mouse arrow to this word and press the left mouse button to start the program.

Now a white screen appears, on which the mouse pointer is visible. If you now press the left mouse button and move the mouse about, you can draw on the screen. A rectangle is used as a 'brush'. Pressing the right mouse button terminates the program and a box appears containing the message 'Program end'. Point to the word 'Return' in this box and press the left mouse button to return you to the Editor.

How does this program work? Drawing takes place with the instruction in the centre of the program. It draws filled rectangles, with the four parameters specifying the corner points. The first instruction of the program (DEFFILL 1,2,8) determines what pattern the rectangle is to be filled with.

Information about the mouse is contained in the built-in variables MOUSEK, MOUSEX and MOUSEY.

MOUSEK determines which mouse button is being pressed. MOUSEK=1 means that the left button was pressed and MOUSEK=2 that the right button was pressed. MOUSEX and MOUSEY supply the x and y positions on the screen of the mouse arrow point.

The remaining commands (REPEAT, WHILE, WEND, UNTIL) are loop instructions. The loop formed with the commands WHILE MOUSEK=1 and WEND means 'repeat, as long as the left mouse button is pressed'. The outside loop, made from the instructions REPEAT and UNTIL MOUSEK=2, means repeat, until the right mouse button is pressed'.

Since there are no further instructions after the UNTIL, the program terminates after leaving this loop. Move the cursor (with the arrow keys) to the beginning of the line after the last and enter the following intentionally wrong program line, in which the 'i' of the command word PRINT is missing:

```
prnt "test"
```

When you press the RETURN key to leave the line and confirm the instruction, a bell sounds and on the second screen line the message Syntax error' appears.

The Editor checks during program entry whether the instructions entered are syntactically correct. Now, with the cursor on the letter 'r', press the Delete key three times. The program line should now contain only:

```
p "test"
```

If the RETURN key is now pressed, then you can leave the line, as the letter p is recognized automatically as an abbreviation for the command PRINT.

The Editor

Fundamentals

The GFA BASIC 3.0 Editor is no usual text editor, but was specially designed for program development. Syntactically wrong instructions are recognised at the typing-in stage. In addition, commands are indented automatically in loops or conditional sections and command names expanded from their abbreviations, e.g. p expands to PRINT.

When writing a program a syntax check is carried out whenever the cursor leaves the current line. If the line is not syntactically correct, then the message 'Syntax error' appears on the second line of the screen. It will only be possible to move the cursor from the line by correcting the syntax or putting an apostrophe at the beginning, thus converting it into a Comment.

Only one instruction per program line is allowed, but a comment may be added to the end of a line by beginning the comment with an exclamation mark.

Program lines may be up to 255 characters long. When a line exceeds 80 characters in length it is scrolled horizontally to the left. When the cursor leaves an instruction line a syntax check is done, correct indentation of the line is carried out and the line is formatted. In addition, redundant spaces are removed, e.g. '2 + 2' becomes '2+2', and the letters belonging to command words and variable names are adjusted according to the current DEFLIST setting. The preset parameter DEFLIST 0 causes all command words to be written in upper case and all variable names in lower case.

The Cursor Keypad

Cursor control is by means of the block of arrow keys. The key settings are:

Left arrow - cursor moves one character left
Right arrow - cursor moves one character to the right
Up arrow - cursor moves up one line
Down arrow - cursor moves down one line

The movements of the cursor are subject to certain restrictions. It can be moved a maximum of one character beyond the last character in a line, and a maximum of one line beyond the last line in a program. If the cursor enters a line which is too short for the current cursor position, the cursor jumps to the end of that line, otherwise it maintains its current column position (this is different to the behaviour of the cursor in earlier versions of GFA BASIC). It also is possible to position the cursor with the mouse. To do this point the mouse arrow to the desired place and click the left mouse button.

The Insert key inserts a blank line between the line the cursor is on and the line above. The cursor is set to the beginning of this blank line.

Clr/Home moves the cursor to the upper left corner under the menu bar. Control-Clr/Home jumps to the beginning of the program listing.

The Undo key will cancel any changes made in an edited line, provided that the changes have not been confirmed by moving the cursor off that line.

With the Help key, procedures in the program listing can be shortened to just their name ('folded'). This is done by moving the cursor to the line with the word 'Procedure' on it and then pressing Help. A '>' character is placed at the beginning of the line to indicate that folding has occurred. The procedure can be unfolded in the same way, or by deleting the '>'. Of course this does not affect the procedure in the program itself, only as listed. This folding up of subroutines enables one to create short clear listings in which one 'opens' only the subroutine on which one is working.

A program with folded up procedures can look as follows. Note, that to call a procedure in GFA BASIC 3 only the procedure name is used.

```
init
main_menu
'
> PROCEDURE init
> PROCEDURE main_menu
> PROCEDURE menu_list
> PROCEDURE load
> PROCEDURE store
> PROCEDURE do_work
> PROCEDURE fetch_info
> PROCEDURE show_info
```

With an opened procedure the program then appears as:

```
init
main_menu
'
> PROCEDURE init
> PROCEDURE main_menu
> PROCEDURE menu_list
PROCEDURE load
  FILESELECT "\ *.RSC","",rsc_file$
  IF NOT EXIST(rsc_file$)
    ALERT 1,"File does not exist !",1,"abort",r%
  ELSE
    IF RSRC_LOAD(rsc_file$)=0
       ALERT 1,"Fault when loading the file!",1,
                                          "abort",r%
      END
    ENDIF
  ENDIF
RETURN
> PROCEDURE store
> PROCEDURE do_work
> PROCEDURE fetch_info
> PROCEDURE show_info
```

The Numeric Pad

The numeric pad is normally used to input numbers and some other characters. However, in addition it can, in combination with the Control key, perform the following functions:

Control and 4	Cursor one character to the left
Control and 6	Cursor one character to the right
Control and 8	Cursor one line up
Control and 2	Cursor one line down
Control and 7	Jump to start of program
Control and 1	Jump to end of program
Control and 9	Move one page up
Control and 3	Move one page down
Control and 0	Equivalent to pressing Insert
Control and .	Equivalent to pressing Delete

The functions are similar to those obtained on a PC when NUMLOCK is not on.The numeric pad can also be switched to a mode in which the keys can be used without having to press Control as well. The switch is made by pressing Control and '-' on the numeric pad, and indicated by a circumflex to the left of the menu bar. This key combination acts as a toggle switching between modes each time it is pressed.

Further Editing Commands

The Delete key deletes the character at the cursor position, the remainder of the line being pulled to the left.

The Backspace key deletes the character to the left of the cursor.

The Tab key moves the cursor eight character positions to the right.

Control-Tab moves the cursor eight character positions to the left.

The Return or Enter keys move the cursor to the beginning of the next line.

The Escape key enters Direct Mode, in which you can type in commands.

Further Control Commands

Control-Delete	Deletes the line on which the cursor appears
Control-U	Restores a line deleted as above, or below (the line may be restored several times - like copying)
Control-Y	Deletes the line on which the cursor appears
Control-N	Inserts a blank line as with the Insert key
Control-Q	Call block menu (like function key F4)
Control-B	Mark beginning of block (Block start)
Control-K	Mark end of block (Block end)
Control-R	Page up
Control-C	Page down
Control-E	Replace text
Shift-control-E	Find/Replace inputted text
Control-F	Find text
Shift-control-F	Input text and find it
Control-left arrow	Cursor jumps to beginning of the line
Control-right arrow	Cursor jumps to the end of the line
Control-up arrow	Page up
Control-down arrow	Page down
Control-Clr/Home	Jump to beginning of the program
Control-P	Deletes everthing to the right of the cursor
Control-O	Brings back the string deleted with Control-P and inserts it at the cursor position. Perhaps it is difficult to remeber these combinations; we thought of: P = Put line end into buffer O = Output line from buffer
Control-Z	Jump to the end of the program
Control-Tab	Cursor jumps one tab position to the left
Control-G (goto)	Move to Line Number display (top right on Menu bar), then enter line number

A special group of Control commands makes the setting of 'Editor Marks' possible. These marks are only meaningful to the Editor and have no effect on the actual program. These marks can be set by Control-n, where n is a number on the main keyboard. The cursor can be made to jump to a given mark by simultaneously pressing Alternate and the appropriate number.

The key combinations of Alternate and the numbers 7-9 and 0 are pre-allocated. Pressing Alternate and 7 makes the cursor to jump to the last cursor position before changing modes or before a program was last Run. With Alternate and 8 the cursor jumps to its position at the start of the Editor. Alternate and 0 moves the cursor to the last cursor position at which a change was made, and Alternate and 9 moves it to the position at which the last search procedure was started.

The Menu Bar and Function Keys

The items in the menu bar on the second and first screen lines are also available via the function keys and Shift-function keys respectively.

On the far left of the menu bar is the Atari symbol (), clicking which generates another menu containing the menu titles, Atari symbol and GFA BASIC. Clicking the Atari symbol causes an Alert box to appear, giving the title and version number of GFA BASIC 3.xx and the two options Editor and Menu. Clicking the Editor button returns you to the Editor, or clicking Menu brings you back to the same menu.

The GFA BASIC Menu Bar contains the following items:

Save

A File-Select box appears, by means of which the current program can be stored by entering a program nameand clicking on OK.

Load

A program can be similarly loaded into the Editor by clicking on the program name and then clicking on OK.

Deflist

Makes possible the adjustment of the appearance of the program listing (see DEFLIST later on for details, or experiment).

New names
By means of this item a mode can be set in which variable names are queried as you introduce them into a program. This is useful as GFA BASIC allows long variable names, making typing errors possibile. If you are editing a program and do not mean to introduce new variables and you mis-type an existing name, this feature will warn you.

On the right-hand side of the menu bar is a clock and the line number display. The use of these two items is explained later on.

Under the Atari symbol there is a space for two further indicators, namely an up-arrow character to show when Caps-Lock is active on the keyboard, and the circumflex showing when the numeric pad is used for cursor movements, etc. These two modes can be entered and exited by clicking the mouse on the space for the indicator (or the indicator itself if present).

Load (F1)

This command load is used to load a GFA BASIC 3 program. This will be in tokenised format, which enables programs to be loaded and saved quickly, using less disk space. The extension .GFA will be looked for by default.

> Earlier versions of GFA BASIC saved their programs using a different tokenisation process. For these programs to be loaded into version 3 they must first be saved, using the old GFA BASIC, in ASCII format (Save,A') and then loaded into Version 3 with Merge. After that they can be stored in version 3 format with Save, and then Loaded as normal.

Save (Shift+F1)

A File-Select box appears, in which the desired program name can be specified. The extension .GFA will be added to the filename if no other is given. If a file of that name already exists on the disk it will be renamed to .BAK.

Merge (F2)

With this command a file in ASCII format can be inserted into the current program, starting at the line above the cursor position. The default file extension will be .LST. Syntax checking is also carried out during the Merging process, but instead of interrupting with a Syntax error when an uninterpretable line is encountered, the line will be prefaced with the characters ==>', and must be fixed before the program can be Run.

Save,A (Shift+F2)

The current program will be stored in ASCII format. The extension .LST will be added to the filename if no other is specified. If a file with that name already exists on the disc, it will be renamed to .BAK.

Llist (F3)

This command causes the program to be printed out in the manner specified by so-called Dot Commands within the program. They have no effect other than on the printout, and are as follows, where 'x' represents a digit:

.ll xx	Maximum line length
.pl xx	Maximum page length
.ff xxx	Form feed character (for printers, which have other values than &H0C (Decimal 12). (.ff 012 is the default)
.he head	Text to be put on the first line of each page
.fo foot	Text to be put on the last line of each page
.lr xx	Left margin
.l-	Conditional printing: This instruction causes the following lines not to be printed
.l+	Printing starts again if it was stopped by a previous .l-
.n1 to .n9	Switches on line numbering adjusted to occupy up to nine character positions
.n0	Switches off line numbering
.PA	Forces a form feed
.P-	The point commands are NOT listed
.P+	The point commands ARE listed, as usual .P+ and .P- influence the whole listing (like .Nx), the last command gives the effect.

In the header and footer text the following can also be inserted:

\ xxx	The ASCII character xxx
\ d	Date
\ t	Time
#	Page number

(To print the symbols \ and #, use \\ and \# respectively.)

Quit (Shift-F3)

After querying this command to make sure you are sure, GFA BASIC is exited completely and normally you will be returned to the Desktop.

Block (F4)

If no block has been marked, then the message 'Block ???' appears on the second line in order to indicate that a block command would be unreasonable. If, however, a block has been marked, then the Block menu appears on the top line of the screen. Items can then be chosen with the mouse, or by pressing the key corresponding to the first letter of the command (except for block Delete, which for safety is called from the keyboard by Control-D).

Copy

Copies the block to the current cursor position. The block remains marked.

Move

Moves the block to the current cursor position. The block is then 'forgotten'and no further block commands can be used until another is marked out.

Write Stores the block as an ASCII file (to be read via Merge)

Llist Prints the block out

Start Moves the cursor to the beginning of the block

End Moves the cursor to the end of the block

Del Deletes the block (Control-D)

Hide Removes the block markers

Clicking the mouse outside the Block menu or pressing a key removes it, and the original menu is restored.

New (Shift-F4)

The program currently in the Editor is deleted

BlkEnd (F5)

The line before the cursor is marked as the end of a block. If the start of block marker is located before this line, the block is shown in a different colour (or with a dotted background if a monochrome monitor is being used). The end of a block can also be marked by means of Control-K, without recourse to the Block menu.

BlkSta (Shift-F5)

Marks the beginning of a block as above. Control-B from the keyboard has the same effect.

Find (F6)

A string of text can be input, which then becomes the object of a search starting from the current cursor position. If the string is found, the search can be continued for another occurrence of the string with Control-F or Control-L. When a string is found, the cursor is put at its position, or, if not, at the end of the program. If Find is called again, the previous search string is presented, which may be accepted with Return, deleted with Escape, or edited with the cursor, delete and backspace keys. The command can also be called directly by Shift-Control-F or Shift-Control-L.

Procedures which have been 'folded' (see above) are not searched.

Replace (Shift-F6)

This command is for replacing one string of text by another. First the user is asked for the text to be replaced, then the replacement text. If the text to be replaced is found, the cursor moves to the beginning of that line. The actual replacement can be effected with Control-E, whereupon another search is made for further occurrences of the string to be replaced. The command can also be called from the keyboard using Shift-Control-E. The editing facilities for the strings are available as above.

Again, 'folded' procedures are not searched.

Pg down (F7)

Scrolls the screen one page down. Also called via Control-C.

Pg UP (Shift-F7)

Scrolls the screen one page upwards. Also called via Control-R.

INSERT/Overwr (F8)

Switches between insert and overwrite modes.

Txt 16/text 8 (Shift-F8)

This feature is available only if a monochrome monitor is being used.It refers to the size of characters displayed on the screen. 16-pixel high characters (default) allow 23 lines to be displayed at once on the screen. Using 8-pixel high characters, 48 lines are displayed.

Flip (F9)

This command switches between the Edit and Output screens. Pressing any key or a mouse button also switches from the Output to the Edit screen.

Direct (Shift-F9)

Switches to Direct Mode, in which commands are responded to immediately, e.g. `PRINT "hello"`. Some commands, e.g. loop commands, are not however available in Direct Mode. This mode can also be reached by pressing the Escape key. In addition, by use of the up- and down-arrow keys, the last eight commands entered in Direct Mode can be recalled, edited, entered, etc. The Undo key recalls the last command. Direct Mode can only be entered when the syntax of the current line in the Editor is correct.

Several lines of instructions can be executed from Direct Mode by writing them as a procedure in Edit mode and then calling it from Direct Mode.

Test (F10)

Issuing this instruction causes all loops, subroutines and conditional instructions to be checked for consistency without actually running the program.

Run (Shift-F10)

The program currently in the Editor is started. If this program contains a structural fault, of the kind that could have been found with Test above, then an appropriate error message is displayed and the program is not started. A running program can be interrupted by the Break key' combination, Control-Shift-Alternate.

The Line Number & Clock Displays

The Line number display and the Clock on the right of the menu bar both react when clicked with the mouse. In the case of the clock , the cursor moves to the first hour digit, and the time can be set by typing in the new time and ending with Return. The input can be edited with the Cursor and Backspace keys, or pressing Escape aborts the process, leaving the time display as it was.

With the Line number display, the cursor moves there, and a number can be typed in. Pressing Return causes the cursor to jump to that line number. Instead of clicking on the Line number display, Control-G could be typed to produce the same effect. Only numbers are accepted as the input and they can be edited with the Escape, Cursor and Backspace keys as before.

Special Commands

DEFBIT f$
DEFBYT f$
DEFINT f$
DEFWRD f$
DEFFLT f$
DEFSTR f$

f$: string constant

DEFxxx f$

The instruction DEFxxx facilitates variable declaration, where xxx is the variable type, specified as follows:

DEFBIT "b" !

All variables beginning with the letter 'b' or post-fixed with '!' are declared as Boolean (logical) variables.

DEFBYT "by" |

All variables beginning with the two letters 'by' or post-fixed with '|' ('rule' character) are declared as 1-byte integers.

DEFWRD "w" &

All variables beginning with the letter 'w' or post-fixed with '&' are declared as 2-byte signed integers.

DEFINT "i-k,m-p" %

All variables beginning with letters from i to k and m to p, or post-fixed with '%' are declared as 4- byte signed integers.

DEFFLT "x-z" #

All variables beginning with letters from x to z, or post-fixed with '#' are declared as 8-byte floating point (the default variable type).

DEFSTR "s,t" $

All variables beginning with the letters 's' or 't' or ending with '$' are declared character strings.

Appearances of DEFSNG or DEFDBL will be replaced automatically by the Editor with DEFFLT.

These definitions will normally be declared at the beginning of a program, but may be changed at any time. The post-fixed symbols will always override any previous global definitions. The default variable type is floating point.

The above definitions will have no effect on variable which are already in the program, they only affect those which are still to be type in.

DEFLIST n

n : iexp

DEFLIST determines the format of the program listing. The numerical expression n can take a value between 0 and 3 (inclusive). The effect of the instruction on the listing is as follows:

n	Command	variable
0	PRINT	abc
1	Print	Abc
2	PRINT	abc#
3	Print	Abc#

The default mode is DEFLIST 0.

DEFLIST 0

Instructions and functions are represented in capitals. Variables, procedure and function names are in lower case.

DEFLIST 1

Instructions, functions, and procedure and variable names are represented with the first letter in upper case, and the remainder in lower case.

DEFLIST 2

As DEFLIST 0, except that the variable-type post-fix is added.

DEFLIST 3

As DEFLIST 1, except that the variable-type post-fix is added.

$ text

The command $, which is treated by the Interpreter like a REM, is used for the control of the compiler.

CHAPTER 2
VARIABLES AND MEMORY MANAGEMENT

Variable Types

GFA BASIC 3 allows the following variable types:

Name	Postfix	Memory requirements
Boolean	!	1 Byte (1 bit in arrays)
Byte		1 Byte
Word	&	2 Byte
Integer	%	4 Byte
Float	#	8 Byte
String	$	(Dependent on the length)

Boolean (logical) variables can accept only the values 0 (FALSE) or -1 (TRUE). If a non-zero value is assigned, then this value is taken as -1. This variable type is designated by the postfix '!', and occupies '1' byte of memory. In Boolean arrays an array element requires only one bit.

Examples:

```
b!=TRUE
c!=x>y
```

Byte variables can accept values between 0 and 255. Larger values will provoke a "Number not byte" error. The postfix of this variable type is the vertical rule character '|'. As the name implies, this variable type occupies one byte.

Example:

```
x|=128
```

Word variables are signed 2-byte integers. The postfix of this type is '&'. Numbers in the range -32768 to 32767 can be represented. Outside this range, a "Number not word" error occurs.

Example:

```
x&=32767
```

Integer variables are signed, occupying 4-bytes, with the postfix '%'. Numbers can be represented in the range from -2147483648 to 2147483647.

Example:

```
x%=2000000000
```

Float is a floating-point variable type occupying 8 bytes of memory. As this is the default type, no postfix is necessary, but the type may be made explicit with the postfix '#'. The range of representable numbers extends from 2.225073858507E-308 to 3.59538626972E+308.

Example:

```
x=123456789e123
```

Character strings (Strings) are designated by the postfix '$'. They can have a maximum length of 32767 characters. Strings are administered by means of a so-called descriptor, six bytes in length. The first four bytes contain the address of the character string, the last two bytes the length of the Strings. If a string contains an odd number of characters a zero filler byte is added. The address of the descriptor (Backtrailer) is also added.

Example:

```
a$="qwertyuiop"
```

The addresses of all variable types can be determined with the help of the functions VARPTR (V:) and ARRPTR (*). With Strings VARPTR returns the address of the first byte of the character string, and ARRPTR returns the address of the descriptor. With arrays one can determine the addresses of the individual array elements with VARPTR/V:

Example:

```
adr%=V:x%(5)
```

By means of ARRPTR/* the address of the Array Descriptor can be found

Example:

```
Arr_des%=ARRPTR(x%())
```

Arrays

DIM, DIM?
OPTION BASE
ARRAYFILL

All variable types can be used in arrays, or 'fields'. An array can be dimensioned with DIM, or the size of an array can be determined with DIM?.

The management of arrays in memory is effected by means of Descriptors. A descriptor is a structure six bytes in length, the first four bytes of which contain the address of the array. The next two bytes specify the number of dimensions. The array field itself begins with sets of four bytes, giving the number of elements in each dimension (beginning with the last dimension) followed by the actual contents of the dimensions. With string arrays, instead of the contents, the descriptors of the character strings follow the four number-of-elements bytes. For example, after

```
DIM a%(2,3)
```

*a%() gives the address of the array descriptor. The number of dimensions of the array is in the last two bytes of the descriptor, therefore

```
PRINT DPEEK(*a%()+4)
```

returns the value 2.

The array itself begins with the number of elements in the second dimension, where the zeroth element is taken into account (assuming OPTION BASE 0 is valid), followed by the number of elements in the first dimension. Then

```
PRINT LPEEK(V:a%(0,0)-8)
```

returns the value 4.

```
PRINT LPEEK(V:a%(0,0)-4)
```

returns the value 3

Following this, the actual contents in the order:

a%(0,0), a%(1,0), a%(2,0), a%(0,1), a%(1,1), a%(2,1), etc.

DIM x(d1, [d2,...]) [,y(d1, [d2,...])]
DIM?(x())

x,y : Variable name (arbitrary variable type)
d1,d2 : iexp

With the instruction DIM numerical and character string arrays can be declared. The structure of such an array was explained in the introduction of this section.

The possible number of the dimensions of the array is only limited by:

a) the last dimension must be smaller than 65535
b) the product of the number of the field elements must be smaller than 65535

So DIM a%(100,10,10) is permitted, as the last dimension (10) and the product of the number of field elements (100*10*10 = 10000) are both less than 65535.

In an array, only variables of the same type may exist. The type of array takes precedence over the type of number put into it.

The function DIM? determines the total number of elements in an array.

Example:

```
DIM x(10)
x(4)=3
PRINT x(LEN("test"))
PRINT DIM?(x())
'
DIM y%(2,3)
PRINT DIM?(y%())
```

--> Two arrays are declared. The output on the screen consists of the numbers 3, 11 and 12.

OPTION BASE 0 (default)
OPTION BASE 1

With help of the command OPTION BASE it can be decided whether a n array is to contain a zeroth element or not. With OPTION BASE 0 a zeroth element is allocated, with OPTION BASE 1 it is not, and the array starts with element number 1.

The contents of fields are not changed by the OPTION BASE- instruction, however the indices may be, as in the following:

Example:

```
DIM x%(3)
FOR i%=3 DOWNTO 0
   x%(i%)=i%
   PRINT i%,x%(i%)
NEXT i%

OPTION BASE 1
FOR i%=3 DOWNTO 0
   PRINT i%,x%(i%)
NEXT i%
```

--> x%() is declared, by default including the zeroth element, and the elements are given the values 0-3 and printed. After OPTION BASE 1 , there is no longer a zeroth element, and accessing the element 3 actually accesses what was element 2, etc.

The program ends with an error message on attempting to access the no longer valid zeroth element of the array.

ARRAYFILL x(),y

x: Name of an array with numerical variable type
y: aexp

The instruction ARRAYFILL sets all elements of the array x() equal to the value of the numerical expression y.

Example:

```
DIM x(10)
PRINT x(4)
ARRAYFILL x(),5+1
PRINT x(4)
```

--> The first number printed is zero, because when an array is dimensioned all elements are automatically set to zero. After filling the array with 5+1's, the number 6 is printed on the screen.

Note: It is not possible to use the ARRAYFILL command with string arrays.

Type Transformation

TYPE (x)

x: iexp

With the function TYPE the type of a variable can be determined. The function operates on the pointer to the variable, and returns a value corresponding to the variable type as follows:

Floating-point	0
String	1
Integer	2
Boolean	3
Floating-point array	4
String array	5
Integer array	6
Boolean array	7
Word	8
Byte	9
Word array	12
Byte array	13

Note: an invalid pointer returns the value -1.

Example:

```
a$="test"
x%=4
DIM y(3)
PRINT TYPE(*a$),TYPE(*x%),TYPE(*y())
```

--> The numbers 1, 2 and 4 appear on the screen. (*a$, etc. represent the pointer to a variable.)

ASC(a$)
CHR$(x)

a$: sexp
x : aexp

ASC and CHR$ are complementary functions.

The function ASC supplies the ASCII code of the first character in the subject string. If the string is of zero length (""), zero is returned.

CHR$ gives a character, from a specified ASCII code. Only the lowest 8 bits of x (the low-byte) are relevant, as ASCII codes only go up to 255.

Example:

```
PRINT ASC("TEST")
code|=ASC(CHR$(65))        ! CHR$(65) is 'A'
PRINT code|,CHR$(189)
```

--> The numbers 84 and 65 and the copyright sign appear on the screen.

STR$(x [,y] [,z])
BIN$(x [,y])
OCT$(x [,y])
HEX$(x, y)

The functions STR$, BIN$, OCT$ and HEX$ convert a numerical expression into a character string.

The length of the required output string can be specified by a second parameter. If necessary, it is padded at the front by blanks (STR$) or zeros (BIN$, OCT$ and HEX$). If the length specified is too short, only that number of characters will be returned.

STR$(x)
STR$(x,y)
STR$(x,y,z)

x, y, z: aexp

STR$ produces a string from the number 'x', with 'y' specifying the required length.

A further variant of STR$ is provided with the third parameter 'z'. The number is formatted and rounded with 'y' characters and 'z' decimal places.

Example:

```
a=123.4567
PRINT STR$(a,6,2)
PRINT STR$(PI,5,3)
PRINT STR$(PI,2,2)
```

--> Prints the numbers 123.46, 3.142 and 14 on the screen.

BIN$(x[,y])
OCT$(x[,y])
HEX$(x[,y])

x, y: iexp

BIN$ converts an integer to binary (base 2) representation. The optional parameter 'y' specifies the number of character positions (1 to 32) to be used.

OCT$ converts an integer to Octal (base 8) representation. The optional parameter 'y' specifies the number of character positions (1 to 11) to be used.

HEX$ converts an integer to hexadecimal (base 16) representation. The optional parameter 'y' specifies the number of character positions (1 to 8) to be used.

Example:

```
x=32+15
a$=OCT$(16+7,4)
PRINT HEX$(x),a$,BIN$(1+4+16+64,8)
```

--> Prints 2F, 0027 and 01010101 on the screen.

VAL(a$)
VAL?(a$)

a$: sexp

VAL() turns a character string into a number. If VAL() encounters a character that cannot be interpreted as part of a number, the conversion stops at that point with the characters successfully converted returned as the result. If no number is found at the beginning of the string, zero is returned.

By adding the prefix &H (hex) or &X (bin) or &O (oct) numbers in hexadecimal, binary and octal notation can be recognized.

The prefixes '$' and '%' may also be used to identify hexadecimal and binary notation respectively.

With **VAL?** one can determine the number of characters convertible with VAL, returning zero if none can be converted.

Examples:

```
a$=STR$(12345)
PRINT VAL(a$),VAL("-.123abc123"),VAL?("3.00 km")
```

--> The numbers 12345, -0.123 and 4 are printed on the screen.

```
PRINT VAL("&H"+"AF")
```

--> '175' is displayed.

```
PRINT VAL("$AA")
PRINT VAL("%10101010")
```

--> '170' is displayed twice.

CVI(a$) CVL(a$) CVS(a$) CVF(a$) CVD(a$) MKI$(x) MKL(x) MKS(x) MKF(x) MKD (x)

a$: **sexp**
x: **aexp**

The functions CVI, CVL, CVS, CVF and CVD convert character strings into numbers, but, as opposed to VAL/STR$, it is the internal representation which is set.

The individual CVx functions have the following effects:

CVI changes a 2-byte string into an integer.

CVL changes a 4-byte string into an integer.

CVS changes a 4-byte string which contains a valid BASIC-compatible number into GFA BASIC floating-point format.

CVF changes a 6-byte string into GFA BASIC 1 or 2 floating- point format.

CVD converts 8 byte string into GFA BASIC 3 floating-point format.

MKI$, MKL$, MKS$, MKF$ and MKD$

are the inverse of the CVx functions above. Thus:

```
MKI$(x%)=CHR$(SHR(x%,8))+CHR$(x%)
MKL$(x%)=CHR$(SHR(x%,24))+CHR$(SHR(x%,16))
                          +CHR$(SHR(x%,8))+CHR$(x%)
```

(SHR is the shift-right function. Note that the high-byte comes first.)

Uses might be reading the number formats of other programs or in saving space when storing numbers in random-access files.

Example:

```
a$=MKL$(1000)
PRINT CVL(a$),LEN(a$)
b$=MKD$(100.1)
PRINT CVD(b$),LEN(b$)
```

--> Prints the numbers 1000 and 4, and 100.1 and 8 on the screen.

CINT(x)
CFLOAT(y)

x: **aexp**
y: **iexp**

The function CINT changes a floating-point format number 'x' into a rounded integer value.

Similarly CFLOAT changes a integer 'y' into a floating-point number. This function is not normally required and is specified only for the sake of completeness. With compiled programs, however, it has a use.

Example:

```
a=1.2345
a%=10000
b%=CINT(a)
b=CFLOAT(a%)
PRINT b%,b
```

--> Prints 1 and 10000 on the screen.

Pointer Operations

*****
xPEEK, xPOKE
BYTE{}, CARD{}, INT{}, LONG{}, {}, FLOAT{}
SINGLE{}, DOUBLE{}, CHAR{}
V:, VARPTR, ARRPTR, ABSOLUTE

***x**

x: svar or an array name followed by ()

The multiply sign also serves as a pointer symbol. In this case *x gives the address of the variable x in memory. With character strings *x$ gives the address of the String Descriptor (as with ARRPTR(x$)).

*x is synonymous with ARRPTR(x). This usage has special meaning with the indirect passing of arrays and variables to subroutines. In addition one can use, in Version 3, the instruction VAR.

Examples:

Version 2

```
' indirect array passing
DIM a(3)
change(*a())
PRINT a(2)
'
PROCEDURE change(ptr%)
  SWAP *ptr%,x()
  ARRAYFILL x(),1
  SWAP *ptr%,x()
RETURN
```

Version 3

```
DIM a(3)
change(a())
PRINT a(2)
'
PROCEDURE change(VAR x())
  ARRAYFILL x(),1
RETURN
```

--> The contents of the array a() are changed, without its name being used in the procedure CHANGE. The number 1 is printed on the screen (see also SWAP).

PEEK(x) DPEEK(x) LPEEK(x)
POKE x,y DPOKE x,y LPOKE x,y
SPOKE x,y SDPOKE x,y SLPOKE x,y

x, y: iexp

With the function PEEK and the instruction POKE and their variants one can read from specified memory locations and write to them. The individual variants are:

PEEK(x)	Reads a byte from the address x.
DPEEK(x)	Reads two bytes (a word) starting at address x.
LPEEK(x)	Reads four bytes (a long word) starting at address x.
POKE x,y	Writes the value y as a byte to the address x.
DPOKE x,y	Writes y as a 2-byte word to the address x.
LPOKE x,y	Writes y as a 4-byte word to the address x.

It is important that when using DPEEK, LPEEK, DPOKE and LPEEK only even addresses are given.

POKE instructions have variants which work in the supervisor mode. So protected addresses, e.g. 0 to 2047, can be modified. The appropriate instructions are SPOKE, SDPOKE and SLPOKE. Caution is advised, particularly in the supervisor mode, since modifications to protected addresses can have serious consequences. The functions always work in the supervisor mode.

Example:

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

-> Sets the head- and tail-pointers to the keyboard buffer to the buffer start, thus effectively erasing the buffer.

Or alternatively:

```
REPEAT
UNTIL INKEY$=""
```

--> Deletes the keyboard buffer by character by character.

BYTE{x} CARD{x} INT{x} LONG{x} {x} FLOAT{x} SINGLE{x} DOUBLE{x} CHAR{x}

x: iexp

By means of these commands one can read certain variable types starting from a given address, or write them to an address.

As a function, e.g. y=BYTE{x}, one can read, starting from the address x, and as an instruction, e.g. BYTE{x}=y, one can write, again starting from address x.

It is important when using INT{}, CARD{}, LONG{}, {}, FLOAT{}, SINGLE{} and DOUBLE{} that only even numbered addresses are specified, since otherwise an address error occurs and three bombs are released.

Type	Meaning
BYTE{x}	Reads/writes a byte
CARD{x}	Reads/writes a 2-byte unsigned integer
INT{x}	Reads/writes a 2-byte signed integer Instead of INT{} you may use WORD{} which has the same effects.
LONG{x}	Reads/writes a 4-byte integer
{x}	Reads/writes a 4-byte integer
FLOAT{x}	Reads/writes an 8-byte variable in BASIC v3 floating-point format
SINGLE{x}	Reads/writes a 4-byte floating-point variable in IEEE single-precision format

DOUBLE{x} Reads/writes aa 8-byte floating-point variable in IEEE double-precision format

CHAR{x} Reads a string of bytes until a null byte (zero) is encountered, or writes the specified string of bytes and appends a null byte. Particularly important for communication with C - routines and GEMDOS

With x%=LONG{adr%} the variable x% is assigned the long-word value found at the address adr%, and with LONG{adr%}=x% the value of x% is written as a long-word to the address adr%.

With SINGLE and DOUBLE it is possible to read or write in IEEE format, used by some 'C' compilers. So, with GFA BASIC, a number in the SINGLE or DOUBLE format can be converted and displayed in hexadecimal:

```
a$=SPACE$(4)
SINGLE{V:a$}=1.2345
PRINT HEX$(CVL(a$),8)
```

or,

```
a$=SPACE$(8)
DOUBLE{V:a$}=1.2345
PRINT HEX$({V:a$},8)
PRINT HEX$({V:a$+4},8)
```

Some functions mentioned above approximately correspond: LONG{x} (or just {x}) for instance corresponds to LPEEK(x). However, {x} is quicker than LPEEK, although it will not work in the Supervisor mode. Attempting to access protected storage locations (0 to 2047) with {x} results in a bus error, with the release of two bombs.

Examples:

```
adr%=XBIOS(2)
t%=TIMER
FOR i%=1 TO 4000
  VOID LPEEK(adr%)
NEXT i%
PRINT (TIMER-t%)/200
'
t%=TIMER
FOR i%=1 TO 4000
  VOID LPEEK(adr%)
NEXT i%
PRINT (TIMER-t%)/200
'
'
PRINT x
FLOAT{*x}=PI
PRINT x
```

--> The first part of the example shows that { } works more quickly than LPEEK, and the second part demonstrates writing a floating-point number indirectly to a variable.

```
BYTE{XBIOS(2)+100*160}=&HFF
CARD{XBIOS(2)+102*160}=&HFFFF
LONG{XBIOS(2)+104*160}=&HFFFFFFFF
'
a$="test"+CHR$(0)
PRINT CHAR{v:a$};
'
b$=SPACE$(5)
CHAR{v:b$}="word"
PRINT b$,ASC(RIGHT$(b$))
```

--> First, some values are written directly to the screen memory and appear as lines. Then a$ is assigned conventionally, with a zero byte added, whereupon it is read with CHAR and printed. Then the word "word" is written into the space made for it by assigning b$ as five spaces. The reassigned b$ is then printed, together with the ASCII code of its last character, to prove that a null byte was indeed added to "word". The output is thus 'test','word','0'.

VARPTR(x) V:x
ARRPTR(y) *y

x: Arbitrary-type variable name
y: Arbitrary-type variable name, or array name with empty brackets

VARPTR(x) or V:x returns the address of variables or strings or particular elements of arrays.

ARRPTR(y) or *y returns the addresses of variables, but for strings or arrays the address of the Descriptor is returned.

VARPTR and V: are synonymous, as are **ARRPTR** and *.

Example:

```
DIM x%(10)
a$="test"
PRINT VARPTR(x%(0)),V:x%(1),ARRPTR(x%())
PRINT ARRPTR(a$),*a$,VARPTR(a$)
```

--> The third line prints the addresses of the first two elements of x%(), together with the address of the Array Descriptor. The fourth line prints the address of the String Descriptor twice, followed by the address of the first byte of a$.

ABSOLUTE x,y

x: a variable of arbitrary type
y: iexp

With the instruction ABSOLUTE the address y is given to the variable x.

Example:

```
ABSOLUTE x,*y
x=13
y=7
PRINT x,y,*x,*y
```

--> Here the variable x is assigned to the address of variable y, so that at the end both variables have the same value (7) and same address.

Deleting and Exchanging

CLEAR, CLR, ERASE
SWAP
SSORT, QSORT
INSERT, DELETE

CLEAR
CLR x [,y,...]
ERASE z1() [,z2(),...]

x,y : svar or avar
z1, z2 : name of arbitrary arrays

With CLEAR all variables and arrays are emptied. The instruction cannot be used within FOR-NEXT loops or subroutines. When a program is Run it is implemented automatically, otherwise it is really only applicable for dealing with serious faults before using RESUME x appropriately.

The instruction CLR deletes the variables specified in the list following it. Arrays cannot be deleted with CLR however.

ERASE deletes complete arrays, which can then be redimensioned if required. In contrast to Version 2, several arrays can be deleted at once, for example ERASE x(),y().

Example:

```
x=2
y=3
CLEAR
PRINT x,y
'
x=2 y=3
CLR x
PRINT x,y
'
DIM x(10)
PRINT FRE(0)
ERASE x()
PRINT FRE(0)
```

--> The above program prints three zeros and a '3' on the screen. After that the amount of free memory is displayed both before and after the erasure of x(), showing that the memory occupied by the array is released by ERASE.

SWAP a,b
SWAP e(),f()
SWAP *c,d()

a, b : **avar or svar**
c : **Pointer to an Array Descriptor**
d, e, f : **Names of arrays**

In its most simple variant the instruction SWAP serves to exchange two variables of the same type (SWAP a,b). In addition, it can be used for exchanging two arrays. The process is fast, since in fact only the associated descriptors are exchanged, having the effect of also exchanging the dimensioning of the two arrays. Arrays do not need to be dimensioned for this. The third variant, where one of the descriptors is addressed directly, is mainly useful for the indirect passing of arrays to subroutines (see second example).

NOTE! The instruction SWAP should be clearly differentiated from the function SWAP, which is discussed in the section on bit operations.

Examples:

```
x=1
y=2
PRINT x,y
SWAP x,y
PRINT x,y
```

--> The numbers 1 and 2, and then 2 and 1 are displayed

```
DIM x(3)
change(*x())
PRINT x(2)
'
PROCEDURE change(adr%)
  SWAP *adr%,a()
  ARRAYFILL a(),1
  SWAP *adr%,a()
RETURN
```

--> The array x() is filled with 1's in the procedure CHANGE without reference to the name x(), so the subroutine is of general use. The descriptor address is handed over to the subroutine and by means of SWAP the array is addressed under the name a().

In Version 3 the array name itself can be passed:

```
DIM x(3)
change(x())
PRINT x(2)
'
PROCEDURE change(VAR a())
ARRAYFILL a(),1
RETURN
```

QSORT a(s) [OFFSET o] [WITH i()] [,n [,j%()]]
QSORT x$(s) WITH i() [,n [,j%()]]

SSORT a(s) [OFFSET o] [WITH i()] [,n [,j%()]]
SSORT x$(s) WITH i() [,n [,j%()]]

a() : **arbitrary array, or string array**
i() : **Integer-array (|, & or %)**
j%() : **4-byte integer array**
x$() : **String array**
n : **iexp**
o : **iexp**
s : **+, - or no sign**

You can SORT string fields with an OFFSET o from version 3.02 onwards. The OFFSET determines how many characters off the beginning shall not be considered; e.g.

```
DIM a$(256)
FILES "*.*" TO "LISTE"
OPEN "I", #1, "LISTE"
RECALL #1, a$(), -1, x%
CLOSE #1
QSORT a$()
OFFSET 13, x%
OPEN "O", #1, "CON:"
STORE #1, a$(), x%
```

This program gives the directory as a file "LISTE", with RECALL the array a$ gets the contents, then the directory is sorted and STOREd to "CON:". By giving the OFFSET 13 it is not sorted by name but by the length of the files. " " and "*" and "12345678.123" is skipped.

The instructions SSORT and QSORT allow the elements of an array to be sorted according to their size. SSORT utilises the Shellsort and QSORT the Quicksort. In the brackets of the array name a plus or minus sign may be inserted, meaning that the sort is to be done in ascending or descending order respectively. If neither is specified, the sort by default will be done in ascending order, as with '+'.

The parameter 'n' specifies that only the first 'n' elements of the array are to be sorted. If OPTION BASE 0 is active (the default) these are the elements from 0 to n-1; if OPTION BASE 1 is active the elements from 1 to n are sorted. If n=-1 then the whole array will be sorted.

When a further integer array is specified as the third parameter, that array will be sorted along with the first array. Each exchange of elements in the first array is also carried out in second. This facility can be used, for example, if one array contains a sort code, e.g. a post code, and another array contains related information.

During the sorting of string arrays a sorting criterion can be specified in form of an array with at least 256 elements by means of WITH. Without indication of WITH the normal ASCII order is used as a sorting criterion (see second example).

Examples:

```
DIM x%(20)
PRINT "Unsorted: ";
FOR i%=0 TO 10
  x%(i%)=RAND(9)+1
  PRINT x%(i%);" ";
NEXT i%
PRINT
'
QSORT x%(),11
DIM index%(210)
PRINT "Descending sort: ";
FOR i%=0 TO 10
  PRINT x%(i%);" ";
  index%(i%)=i%
NEXT i%
PRINT
'
SSORT x%(-),11,index%()
PRINT "Ascending sorts: ";
FOR i%=0 TO 10
  PRINT x%(i%);" ";
NEXT i%
PRINT
PRINT "Sort WITH array: ";
FOR i%=0 TO 10
  PRINT index%(i%);" ";
NEXT i%
```

--> Gives a unsorted field and two sorted series from random numbers. In a fourth row the values of a second field that was sorted alongside the first one displayed.

INSERT x(i)=y
DELETE x(i)

x: Name of an array
i: iexp
y: aexp or sexp, according to variable type of the array

With the instructions INSERT and DELETE an array element can be inserted or deleted. INSERT inserts the value of the expression y in the array x at the position i. All elements of the array which have an index larger than i are shifted up one position. Thus if an element stood at position 3 before, then it will be found at position 4 after the INSERT instruction. The last element of the array is deleted.

DELETE removes the i-th element of the array x(). All array elements which have an index larger than i are shifted down one position. The last element of the array is made zero (or a null string with character string arrays).

These two instructions are highly suitable for the management of lists, in which elements are constantly to be inserted and deleted.

Examples:

```
DIM x%(5)
FOR i%=1 TO 5
  x%(i%)=i%
NEXT i%
INSERT x%(3)=33
FOR i%=0 TO 5
  PRINT x%(i%)
NEXT i%
```

--> The numbers 0, 1, 2, 33, 3 and 4 are printed on the screen.

```
DIM x%(5)
FOR i%=1 TO 5
  x%(i%)=i%
NEXT i%
DELETE x%(3)
FOR i%=0 TO 5
  PRINT x%(i%)
NEXT i%
```

--> The numbers 0, 1, 2, 4, 5 and 0 are displayed.

Reserved Variable Names

FALSE, TRUE, PI
DATE$, TIME$, SETTIME
TIMER

FALSE
TRUE
PI

The two logical constants FALSE and TRUE contain the values 0 and -1 respectively. The constant PI contains the value of the transcendental number pi.

Example:

```
PRINT FALSE
IF TRUE
  PRINT PI
ENDIF
```

--> Prints the numbers 0 and 3.14159265359 on the screen.

DATE$
TIME$
SETTIME time$, date$
DATE$=date$
TIME$=time$

time$, date$: **sexp**

The function **DATE$** sets the system date in the format:

```
DD.MM.YYYY          (Day.Month.Year)
```

or

```
MM/DD/YYYY          (US-FORMAT, see MODE)
```

Only years between 1980 and 2079 are allowed.

TIME$ sets the system time. The format is:

```
HH:MM:SS            (Hours:Minutes:Seconds)
```

The time is then updated every two seconds.

With the instruction **SETTIME** both time and date can be set. The strings must have the same format as for TIME$ and DATE$ above. If SETTIME is given strings in the wrong format, then the current values are not changed.

The date and time can also be set individually with **DATE$=** and **TIME$=**.

Example:

```
PRINT DATE$,TIME$
SETTIME "20:15:30","27.2.1988"
PRINT DATE$,TIME$
```

--> The system date and time are printed, re-set, and printed again.

TIMER supplies the elapsed time in 1/200 seconds since the system was powered up.

Example:

```
t%=TIMER
FOR i%=1 TO 2500
NEXT i%
PRINT (TIMER-t%)/200
```

--> The time in seconds required for the FOR-NEXT loop is displayed.

Special Commands

LET, VOID

With LET, values can be assigned to variables whose names are keywords. VOID invokes a function but ignores the returned value.

LET x=y

x: avar or svar
y: aexp or sexp

By means of LET one can transfer the value of an expression to a variable. The expression and the variable must either be both numerical or both character strings. Normally LET is not necessary: it served in older BASIC's to allow the use of keywords as variable names. However, GFA BASIC usually recognizes automatically when a keyword is used in this way and allows it.

Example:

```
LET print=3
PRINT print
```

--> '3' is displayed on the screen.

VOID fx
~fi

fx: aexp
fi: iexp

Programming languages normally differentiate between commands and functions. Both cause some activity to be carried out, but with functions this activity 'returns' a value, which may be used as an element of an expression, displayed with PRINT or assigned to a variable etc.

Often, however, the programmer is not interested in the returned value, but only in the activity carried out. For example, the function INP(2) returns the ASCII code of a pressed key. If the program is only supposed to wait for a key, any key, to be pressed, then the code of the key is irrelevant.

With GFA BASIC, in such a case VOID can be used to implement the function, 'forgetting' about the returned value. The alternative form (with a tilde ('~') instead of VOID) calculates an integer value before forgetting it, making it faster than VOID, which calculates a floating-point value.

Example:

```
VOID INP(2)
```

or

```
~INP(2)
```

--> Waits for a key to be pressed. Which key it was, however, will be unknown to the program, as the returned code is lost.

Memory Management

FRE
BMOVE
BASEPAGE, HIMEM
RESERVE
INLINE
MALLOC, MSHRINK, MFREE

FRE()
FRE(x)

x: aexp

This function computes the amount of free available memory. The parameter x is ignored, but if it is present a 'Garbage Collection' is carried out first (non-current versions of strings are deleted and the memory occupied by them freed). FRE() results in the free memory being calculated without the Garbage Collection.

Example:

```
free%=FRE(0)
max%=free%/3/4
DIM x%(max%)
PRINT free%,max%
```

--> An array is dimensioned so that it occupies for instance a third of the free memory space. An integer array occupies 4 bytes per element, thus the extra division by 4.

BMOVE source,destination,length

source, destination, length: iexp

BMOVE copies a block of memory from one area to another. 'Source' is the address of the first byte of the block to be copied, 'destination' is the address of the first byte of the area to which the block is to be copied, and 'length' is the length of the block.

The instruction works noticeably faster with even parameters than with odd. It also works if the source and destination areas overlap.

Example:

```
DIM screen2%(64000/4)
adr%=VARPTR(screen2%(0))
FOR i%=0 TO 300 STEP 100
  PBOX 0,i%,639,i%+50
NEXT i%
PRINT "hi there!"
REM XBIOS(2)=the start-of-screen address
BMOVE XBIOS(2),adr%,32000
BMOVE XBIOS(2),adr%+32000,32000
REPEAT
  IF MOUSEY<>my%
    BMOVE adr%+my%*80,XBIOS(2),32000
    my%=MOUSEY
  ENDIF
UNTIL MOUSEK=2
```

--> Vertical movements of the mouse scroll the screen up and down.

BASEPAGE
HIMEM

In the variable BASEPAGE is the address of the Basepage of the GFA Interpreter. BASEPAGE is a 256-byte long storage area as follows:

Bytes	Contents
0 to 3	Address of the start of TPA (Transient Program Area)
4 to 7	Address of the end of TPA plus 1
8 to 11	Address of text segment of the program
12 to 15	Length of the text segment
16 to 19	Address of the data segment
20 to 23	Length of the data segment
24 to 27	Address of the BSS (Block Storage Segment)
28 to 31	Length of the BSS
32 to 35	Address of the DTA (Disk Transfer Address)
36 to 39	Address of the Basepage of the calling program
40 to 43	Reserved
44 to 47	Address of the Environment Strings
48 to 127	Reserved
128 to 255	Command line (first byte specifies length of command text)

The variable **HIMEM** gives the address of the first free memory location not used by BASIC. This is normally 16384 bytes below the screen area.

Example:

```
a%={BASEPAGE+&H2C}
DO
  a$=CHAR{a%}
  EXIT IF LEN(a$)=0
  PRINT a$
  ADD a%,SUCC(LEN(a$))   ! SUCC= next higher integer
LOOP
```

--> Here the complete BASIC Environment is displayed.

RESERVE [n]

n: iexp

The size of the storage area used by GFA BASIC can be specified. If n is positive, then n bytes are reserved for the Interpreter and the remainder is released. If n is negative, then the whole of the free memory less n bytes is reserved.

RESERVE

If no parameter is specified, the state when the Interpreter was started is reestablished.

Memory can be reserved only in blocks of 256 bytes.

The instruction can be used, for instance, in order to release a storage area for data or Resource files. If the storage area for GFA BASIC is reduced with RESERVE, then one should not forget to enlarge it again later since otherwise the available space becomes smaller with each execution of the program.

Example:

```
RESERVE 2560
EXEC 0,"\PROGRAM.PRG","",""
RESERVE
```

--> 2560 bytes are reserved for the Interpreter, and PROGRAM.PRG (if available) is loaded and started. After exiting from PROGRAM.PRG the reserved space is restored.

INLINE addr,length

addr : 4-byte integer variable, (not an array variable)
length: Integer constant, less than 32700

This instruction reserves an area of memory within a program.

(No comment is possible on the same line as this instruction, since memory is internally reserved where otherwise the comment would be.)

The reserved area always begins at an even address and it is initially filled with zeros. When implementing INLINE this address is written to the integer variable addr. When the program is loaded or saved, the reserved memory area is also.

Positioning the cursor on the program line which contains the INLINE instruction and pressing the Help key causes a menu line with the entries LOAD, SAVE, DUMP and CLEAR to appear on the top line of the editor.

LOAD is used to load a machine code program or data into the reserved area. SAVE saves the contents of the reserved area to disk. DUMP causes a hexadecimal printout on the printer of the reserved area. CLEAR fills the reserved area with zeros. If the INLINE instruction is deleted, so also is the reserved area.

The default file extension is .INL.

Into this storage area, for instance, could be put pictures, tables or machine code programs.

Example:

See the example for C: in the system routines section.

MALLOC(x)
MFREE(y)
MSHRINK(y,z)

x, y, z: iexp

The function **MALLOC** (GEMDOS 72) is used to reserve (ALLOCate) areas of memory.

If its parameter x is equal to -1, then the function returns the length of the largest contiguous free area. If x is a positive number, then this means that x bytes are to be reserved. In this case MALLOC returns the start address of the reserved area. If a fault occurred with the reservation attempt, then the value 0 is returned.

If larger areas are to be allocated, then first some of GFA BASIC's memory must be freed with RESERVE.

Allocated storage areas must always be released before the end of the program, although this is accomplished automatically when leaving the Interpreter.

MFREE (GEMDOS 73) releases the storage location reserved with MALLOC again. The parameter y specifies the start address of the memory area to be released (which was returned by MALLOC). If the release occurred without problems, the value 0 is returned, otherwise a negative number.

MSHRINK (GEMDOS 74) will reduce the size of a storage area previously allocated with MALLOC. The parameter y specifies the address of the reserved storage area (which was returned by MALLOC). The second parameter z gives the required (shrunk) size.

The function MSHRINK returns 0 if the reduction was made without difficulty, -40 if an incorrect address was given as y, or -67, if the new desired size was larger than the current size. It is important with MFREE and MSHRINK that an incorrect address is never given.

Example:

```
RESERVE 1000
PRINT MALLOC(-1)
adr%=MALLOC(60000)
PRINT adr%
IF adr%>0
  x%=MSHRINK(adr%,30000)
  y%=MFREE(adr%)
ENDIF
RESERVE
```

--> GFA BASIC's usable memory is reduced to 1000 bytes, and then the size of the largest free memory area is printed. Then an attempt is made with MALLOC to reserve 60000 bytes. If the reservation was successful, i.e. adr% is not zero, MSHRINK is used to reduce the size of the reserved area to 30000 bytes. finally MFREE releases the memory again.

CHAPTER 3
OPERATORS

Operators in a programming language serve to link and compare elements in numeric and logical expressions.

In this chapter the operators available in GFA BASIC V3 are introduced in in the following five sections.

In the first section the numerical operators +, -, *, /, ^, DIV, \ and MOD are discussed. They link two numerical expressions and produce a number, which can then be assigned to variable with the equals sign (=). The operators + and - have a secondary function as unary operators, giving a sign to a number (+1, -2 etc.).

The second section deals with the logical operators AND, OR, XOR, NOT, IMP and EQV. They link two logical expressions and produce a logical result (TRUE or FALSE). The operator NOT has a special role in that it works on only one logical expression and reverses its logical value (TRUE becomes FALSE and vice versa).

The third section deals with the concatenation (joining) of string expressions using the plus sign (+).

The next section describes the comparison operators, which compare two numerical or two string expressions and produce a logical result (TRUE or FALSE).

The last section details the order in which operators are processed if several occur in one expression, and the use of the bracket symbols (and) , by means of which the order of processing can be governed.

Arithmetic Operators

+ - * / ^
DIV \ MOD
+ -

These arithmetic operators link two numerical expressions and produce a number. This number can then be used as part of another expression, assigned to a variable, printed out with PRINT, etc. The operators + and - are also used as signs to indicate whether numbers are positive or negative (-1, +2, etc.).

x+y	Produces the sum of the numbers x and y (addition)
x-y	Produces the difference of the numbers x and y (subtraction)
x*y	Produces the product of the numbers x and y (multiplication)
x/y	Produces the quotient of x divided by y (division)
x^y	Produces x raised to the power y (exponentiation).
x DIV y	Results in a fast integer division of x by y.
x\y	The backslash '\' character is an alternative form of DIV
x MOD y	Produces the remainder of the division of x by y.

The following relations are equivalent:

```
x DIV y = TRUNC(x/y).
x MOD y = x-y*TRUNC(x/y)
```

(TRUNC is the Truncate function.)

Example:

```
13 DIV 4    produces 3
13 MOD 4    produces 1.
```

+ provides the value x with a positive sign, unless it was negative, in which case it is left as it was.

- provides the value x with a negative sign. If it was positive, it is treated as negative and vice versa.

Logical Operators

AND OR XOR
NOT IMP EQV

These logical operators work on bit level for 32-bit integer values. Logical operators link two logical expressions and produce a logical result (TRUE or FALSE). The operator NOT is an exception, in that it negates the value of a given expression.

The numerical value for FALSE is 0, and for TRUE is -1.

(For those interested in why these values should be chosen, the reason is that TRUE is considered to be all ones in a 32-bit integer, and FALSE to be all zeros . Thus:

```
11111111111111111111111111111111 = TRUE
```

and

```
00000000000000000000000000000000 = FALSE
```

Those familiar with two's complement arithmetic will recognize that the former is the two's complement notation for -1.)

All logical operators can also be applied to numerical expressions. In this case the logical operations are implemented bit by bit. The effects of logical operators will be described with so-called truth tables. In these tables the logical values of the linked expressions are given in the first columns, and the result in the last column.

NOT x

x: iexp

The operator **NOT** negates a given logical expression. It is the only logical operator, which has a single argument. Each individual bit of the argument is modified.

x	NOT x
T	F
F	T

Examples:

```
PRINT NOT FALSE
PRINT NOT TRUE
PRINT NOT 0
```

--> The numbers -1, 0 and -1 appear on the screen.

```
x=1
PRINT BIN$(x,2)
PRINT BIN$(NOT x,2)
```

--> 01 and 10 appear on the screen.

```
x%=17
PRINT BIN$(x%,8),x%
PRINT BIN$(NOT x%,8),NOT x%
```

--> Displays: 00010001 17, and 11101110 -18

x AND y

x, y: iexp

The logical operator **AND** (conjunction) checks whether two logical expressions x and y are both true. Only in this case it produces the value TRUE (-1) . If one or both logical expressions are wrong, then AND produces a logical FALSE. With AND each pair of the 32 bits is tested independently.

x	y	x **AND** y
T	T	T
T	F	F
F	T	F
F	F	F

Examples:

```
PRINT TRUE AND -1
PRINT FALSE AND TRUE
```

--> On the screen the numbers -1 and 0 appear.

```
x=3
y=10
PRINT BIN$(x,4)
PRINT BIN$(y,4)
PRINT BIN$(x AND y,4),x AND y
```

--> Displays: 0011, 1010, 0010 and 2

x OR y

x,y: iexp

The command **OR** (disjunction) checks whether at least one of two logical expressions x and y is TRUE. Only if x and y are both FALSE will the result FALSE be produced. Unlike XOR, (see below) a TRUE result from OR means that one or both arguments are TRUE. OR also works on bit level with numbers.

x	y	x **OR** y
T	T	T
T	F	T
F	T	T
F	F	F

Examples:

```
PRINT TRUE OR -1
PRINT FALSE OR TRUE
PRINT 0 OR FALSE
```

--> The numbers -1, -1 and 0 are displayed.

```
x=3
y=10
PRINT BIN$(x,4)
PRINT BIN$(y,4)
PRINT BIN$(x OR y,4),x OR y
```

--> 0011, 1010, 1011 and the number 11 appear on the screen.

x XOR y

x, y: iexp

The **XOR** operator produces the value TRUE if one, but not both, of the arguments is TRUE. If they are both TRUE, or both FALSE, the result is FALSE.

Unlike OR, where TRUE OR TRUE = TRUE, TRUE XOR TRUE = FALSE. (XOR is the Boolean Exclusive Disjunction operation.)

Again, XOR works individually on each of the 32 bits of numeric arguments.

x	y	x **XOR** y
T	T	F
T	F	T
F	T	T
F	F	F

Examples:

```
PRINT FALSE XOR -1
PRINT -1 XOR 1
PRINT 0 XOR FALSE
```

--> The numbers -1, -2 and 0 are printed on the screen.

```
x=3
y=10
PRINT BIN$(x,4)
PRINT BIN$(y,4)
PRINT BIN$(x XOR y,4),x XOR y
```

--> Displays: 0011, 1010, 1001 and 9.

x IMP y

x,y: iexp

The operator **IMP** (Implication) corresponds to a logical consequence. The result is only FALSE if a FALSE expression follows a TRUE one. IMP also works on bit level. Unlike AND, OR, XOR and EQV, the sequence of the arguments is important.

x	y	x **IMP** y
T	T	T
T	F	F
F	T	T
F	F	T

Examples:

```
PRINT TRUE IMP -1
PRINT 0 IMP FALSE
PRINT TRUE IMP 0
```

--> The numbers -1, -1 and 0 appear on the screen.

```
x=3
y=10
PRINT BIN$(x,4)
PRINT BIN$(y,4)
PRINT BIN$(x IMP y,4)
```

--> Displays: 0011, 1010 and 1110.

x EQV y

x,y: iexp

The operator **EQV** (Equivalence) produces a TRUE result only if the arguments are both TRUE or both FALSE. EQV works on bit level and sets the bits which are same in both arguments. This is exactly the opposite of XOR, so x EQV y is the same as NOT(x XOR y).

x	y	x **EQV** y
T	T	T
T	F	F
F	T	F
F	F	T

Example:

```
PRINT TRUE EQV FALSE
PRINT FALSE EQV FALSE
```

--> On the screen the numbers 0 and -1 appear.

```
x=3
y=10
PRINT BIN$(x,4)
PRINT BIN$(y,4)
PRINT BIN$(x EQV y,4)
```

--> Displays: 0011, 1010 and 0110.

Concatenation Operator +

a$+b$

a$,b$: sexp

The operator '+' also serves to link character strings together. The result is a character string, composed of a$ and b$.

Example:

```
a$="GFA-"
PRINT a$+"BASIC"
```

--> The text 'GFA-BASIC' appears on the screen.

Comparison Operators

= == >= <= <>

With these numerical, logical and string expressions can be compared with each other. The result of this comparison is always one of the logical values TRUE (-1) or FALSE (0). The operator '==' is an exception: string expressions cannot be compared with it.

x=y

x,y: exp

The operator '=' compares two numerical or string expressions for equality. If the two expressions are equal, then the result is logically TRUE, otherwise it is FALSE.

Example:

```
x=6
IF 2=x/3
  PRINT "Ok"
ENDIF
PRINT 2=x/3
```

--> 'Ok' and -1 appear on the screen.

x==y

x,y: aexp

The operator '==' compares two numerical expressions for approximate equality. With this operator only eight and one half decimal places (28 bits of the mantissas of floating-point numbers) are compared. ' ==' is useful for comparing floating-point numbers, where inaccuracies due to rounding can occur.

Examples:

```
PRINT 1.0000000001=1
PRINT 1.0000000001==1
```

--> The numbers 0 and -1 are displayed.

```
a=SINQ(77)        ! SINQ(degrees) is a Quick,
'                 ! lesser accuracy, SIN function
b=SIN(RAD(77))    ! RAD converts from degrees
'                 ! to radians
PRINT a=b
PRINT a==b
```

--> The numbers 0 (logically FALSE) and -1 (TRUE) appear.

x<y x>y x<=y x>=y

x,y: exp

These operators serve to compare the size of numerical and string expressions. With numerical expressions the values are compared, and with string expressions the comparison depends on the ASCII codes of the characters. The string "ABC" is treated as the number sequence 65, 66, 67.

In the comparison "ABC">"AAA", the first characters are compared, which both have the ASCII code 65. Then the next characters are compared. There B has a higher code value than "A", and "B" is then taken as the 'larger'.

Here the comparison of the two strings is broken off, and the statement '"ABC" is larger than "AAA"' is deemed logically TRUE.

A special case of the comparison crops up if the two strings end before an inequality is discovered. An example would be: "AA">"A". This statement is rated logically TRUE, as is "A"+CHR$(0)>"A".

Now to the individual operators:

x>y	TRUE if x is greater than y.
x<y	TRUE if x is less than y.
x>=y	TRUE if x is greater than, or equal to, y.
x<=y	TRUE if x is less than, or equal to, y.

The following are equivalent ways of writing the same expressions:

```
x<=y, x=<y        x>=y, x=>y
```

Example:

```
PRINT "AAA">"aaa"
PRINT -1<=4-5
```

--> The numbers -1 and -1 are printed.

x<>y

x, y: exp

This operator checks whether two numerical or string expressions are unequal. If this is the case, then the statement x<>y is logically TRUE. If x and y are the same, then the result of x<>y is logically FALSE.

The expressions x<>y and x><y are equivalent.

Example:

```
PRINT "test"<>"test"
PRINT -1<>4-5
```

--> 0 and 1 appear on the screen.

Assignment Operator =

x=y

x: var
y: exp

The equals sign '=' can be used not only as a comparison operator, but also to assign a value to a numeric or string variable. The value of the expression y on the right of the equals sign is determined and assigned to the variable on the left.

Numerical expressions can only be assigned to numerical variables, and character string expressions can only be assigned to string variables.

Optionally, the command LET may be used, which also permits the assignment of values to variables which have the same names as keywords.

Example:

```
x=LEN("TEST")+3
a$="GF"+CHR$(65)
PRINT x,a$
```

--> On the screen 7 and 'GFA' appear.

Operator Hierarchy

If several operators are used in an expression, they are processed in a certain order, which depends on the operator's place in the so-called operator hierarchy. The operators at the top of this hierarchy are processed first. The hierarchy is as follows:

()	Brackets
+	Character string addition
= < > => <= <>	Character string comparison
+ -	Signing
^	Exponentiation
* /	Multiplication, division
DIV MOD	Integral and modulo division
+ -	Addition, subtraction
= == < <= >= > <>	Numeric and logical comparison
AND OR XOR IMP EQV	Logical operators
NOT	Negation

By means of the brackets it is possible to force lower-precedence operators to be processed before those higher up the hierarchy.

Example:

```
PRINT 2+4*3
PRINT (2+4)*3
PRINT 2+(4*3)
PRINT 3*2^2
```

--> The numbers 14, 18, 14 and 12 appear on the screen.

CHAPTER 4
NUMERICAL FUNCTIONS

Mathematical Functions

ABS, SGN
ODD, EVEN
INT, TRUNC, FIX, FRAC, ROUND
MAX, MIN
SQR
EXP, LOG, LOG10
SIN, COS, TAN, ASIN, ACOS, ATN, DEG, RAD
SINQ, COSQ

There are numerical functions for the following tasks: the functions ABS and SGN supply the absolute value and the sign of a numerical expression. ODD and EVEN check whether a number is odd or even. INT, TRUNC, FIX and FRAC deal with separate manipulation of the parts of a number to the left and to the right of the decimal point. ROUND rounds an expression. MAX and MIN supply the largest or the smallest of several numerical expressions and SQR the square root of an expression. The trigonometrical functions are SIN, COS, TAN, ASIN, ACOS and ATN. SINQ and COSQ are faster, less accurate alternatives for SIN and COS. EXP and LOG compute powers and logarithms to base e. LOG10 computes logarithms to base 10. DEG and RAD transform values from radians to degrees and vice versa. RND, RANDOM, RAND and RANDOMIZE deal with the generation of random numbers.

ABS(x)
SGN(x)

x: aexp

The function ABS returns the absolute value of a numerical expression. The returned value is the same as the given value, except that its sign is ignored. The sign of the returned value is always made positive.

With the function SGN one can determine the sign of a numerical expression thus:

x	SGN(x)
Negative	-1
Equal to 0	0
Positive	1

Example:

```
x=-2
y=ABS (x)
PRINT SGN(x),ABS(5-3),SGN(ABS(x*3))
```

--> Displays the numbers -1, 2 and 1.

ODD (x)
EVEN (x)

x: aexp

These two functions check whether the numerical expression x is odd or even. ODD supplies the value -1 (TRUE), if x is odd and 0 (FALSE) if x is even. EVEN results in TRUE for an even x and FALSE for an odd x. Zero (0) is treated as an even number.

x	ODD(x)	EVEN(x)
Even	0 (FALSE)	-1 (TRUE)
Odd	-1 (TRUE)	0 (FALSE)
Zero	0 (FALSE)	-1 (TRUE)

Example:

```
x=2
PRINT ODD(x),EVEN(-2),ODD(3*5),EVEN(-3*x)
```

--> The numbers 0, -1, -1 and -1 are displayed.

INT(x)
TRUNC(x)
FIX(x)
FRAC(x)

x: aexp

These functions allow independent manipulation of the parts of a numerical expression to the left and right of the decimal point: **INT**, **TRUNC** and **FIX** (TRUNC and FIX are identical) return a whole number. The function TRUNC simply cuts off the digits to the right of the decimal point. INT returns the largest whole number which is less than or equal to x. There is no difference between TRUNC (or FIX) and INT for positive x-values, however, with a negative, non-integer x a difference arises. So TRUNC(-1.2) removes the decimal places and returns -1 as the result. INT(-1.2), on the other hand, returns the next smaller whole number, namely -2. FRAC returns only the fractional part of x, in other words just the decimal places, with the same sign as x had. FRAC is complementary to TRUNC and not to INT. It is always true that x = TRUNC(x) + FRAC(x), but INT(x) + FRAC(x) is not equal to x for negative values.

Example:

```
x=-1.4
y=TRUNC(1.3)
PRINT y,INT(x),FIX(3*x),FRAC(x-3)
```

--> The numbers 1, -2, -4 and -0.4 are displayed.

ROUND(x [,n])

x: aexp
n: iexp

The function ROUND(x) returns x rounded to the nearest whole number. The variant ROUND(x,n) rounds the expression x to n decimal places. If n is zero, the effect is the same as for ROUND(x). If n is negative, the rounding is done before the decimal point, so ROUND(155, -1) results in the number 160. (NB: CINT() also rounds with an Integer result.)

Examples:

```
y=ROUND(-1.2)
PRINT y,ROUND(1.7)
```

--> Displays -1 and 2 on the screen.

```
FOR i%=-5 TO 5
  PRINT i%,ROUND(PI*100,i%)
NEXT i%
```

--> Displays the loop variable and the associated formatted expression on the screen: PI is multiplied by 100 and then displayed rounded to i% decimal places. Where i% is negative, the rounding is done to the left of the decimal point.

MIN(x [,y,z,...]) MIN(x$ [,y$,z$,...])
MAX(x [,y,z,...]) MAX(x$ [,y$,z$,...])

x,y,z : aexp
x$,y$,z$: sexp

The function MAX returns the largest of the numerical expressions x,y,z,..., specified in the parameter list. Similarly, MIN returns the smallest value. MIN and MAX can also be applied to string expressions.

Example:

```
x=3
y=MAX(3,5.5-4)
PRINT MIN(x,y),MAX(-1,x*2)
```

--> Displays the numbers 3 and 6.

SQR(x)

x: aexp

Produces the square root of the numerical expression x. If the expression x is negative, an error message is invoked.

Examples:

```
x=9
y=SQR(x)
PRINT y,SQR(4*4)
```

--> The numbers 3 and 4 appear on the screen.

```
PRINT SQR(SQR(16))
PRINT SQR(-2)
```

--> The number 2 appears, followed by an error message.

EXP(x)
LOG(x)
LOG10(x)

x: aexp

EXP computes **e** to the power **x** (Euler's number e=2.1782818284...), and **LOG** forms the inverse function, that is the natural logarithm of x. Similarly, **LOG10** returns the logarithm of x to the base 10. The numerical expression x must be larger than zero with the logarithm function, otherwise an error message results.

Example:

```
x=2
y=EXP(2)
PRINT y,LOG10(2*5),LOG(x)
```

--> On the screen 7.389056098931, 1 and 0.6931471805599 appear.

SIN(x) COS(x) TAN(x)
ASIN(x) ACOS(x) ATN(x)
DEG(x) RAD(degrees)
SINQ(degrees) COSQ(degrees)

x,degrees : aexp

These are the trigonometrical functions. The numerical expression x is assumed to specify radians. They compute the following:

SIN	Sine
COS	Cosine
TAN	Tangent
ASIN	Arc-sine
ACOS	Arc-cosine
ATN	Arc-tangent

To convert between radians and degrees one uses the functions DEG() (radians to degrees) or the inverse, RAD(). DEG(x) and RAD(x) are thus equivalent to (x * 180/PI) and (x * PI/180) respectively.

The functions SINQ and COSQ supply sine and cosine values, interpolated in steps of one sixteenth of a degree from an internal table. For graphics work on the screen this accuracy is not distinguishable from the values computed with SIN or COS, however SINQ and COSQ work up to 10 times as fast. Unlike SIN and COS, degrees are expected as the arguments for SINQ and COSQ. SINQ(degrees) corresponds to SIN(RAD(degrees)) and COSQ(degrees) corresponds to COS(RAD(degrees)).

Examples:

```
x=90
y=COSQ(x*PI/180)
z=270*PI/180
PRINT y,SIN(z),TAN(45),ATN(1/2)
```

--> The numbers 1, -1, 1.619775190544, 0.4636476090008 appear.

```
alpha%=30
PRINT SINQ(alpha%)
```

--> The number 0.5 is displayed.

Random Number Generation

RND [(x)]
RANDOM(x)
RAND(y)
RANDOMIZE y

x: aexp
y: iexp

This group of commands deals with the generation of random numbers:

RND Produces a random number between 0 and 1 (including 0, excluding 1). The optional parameter x has no meaning.

RANDOM(x) Produces a random integer between 0 and x (including 0, excluding x). The numerical expression x need not be integer if not all numbers are required to have the same probability.

RAND(y) Produces a 16-bit random integer in the range 0 to y-1. Where y is an integer with a maximum value of 65535 (&HFFFF hex). If y exceeds 65535, then its low-order word is taken as the argument.

RANDOMIZE y Initialises the random number generator with the value y. If the random number generator is initialised several times with the same y, the same sequence of random numbers will be produced. Without the use of RANDOMIZE at the beginning of a program, different sequences will be generated each time the program is run, which may be undesirable in some cases. For initialising the random number generator one can also use RANDOMIZE (without parameters) or RANDOMIZE 0, which initialize the random number generator with a random number.

Examples:

```
x=RND
PRINT x,RND(2)
```

--> Two random numbers between 0 and 1 appear on the screen.

```
x=RANDOM(2)
y=RAND(4)
PRINT x,y,RAND(x),RANDOM(3*x)
```

--> Four integer random numbers are displayed.

```
RANDOMIZE 3
x=RND
RANDOMIZE 3
PRINT x,RND
```

--> The same 'random number' is printed twice.

Integer Arithmetic

Commands and Functions

DEC, INC
ADD, SUB, MUL, DIV
PRED(), SUCC()
ADD(), SUB(), MUL(), DIV(), MOD()

DEC, INC
ADD, SUB, MUL, DIV

These instructions are shorter forms of the following frequently used statements:

```
DEC x       corresponds to    x=x-1
INC x       corresponds to    x=x+1

ADD x,y     corresponds to    x=x+y
SUB x,y     corresponds to    x=x-y
MUL x,y     corresponds to    x=x*y
DIV x,y     corresponds to    x=x/y
```

The statements on the left require less time for their execution than those on the right, particularly in the case of integer variables. It is important to note that INC, DEC, ADD, SUB, MUL and DIV do not carry out an overflow check with integer variables (types %,&,!). If overflow occurs, only the appropriate number of low-order bits (8 for byte, 16 for word, etc) are used for the result. Thus for byte-sized variables, if x!=16 and y!=17, MUL x!,y! gives the result 16 in x!. The extra '256' in the 9th bit is ignored.

DEC i
INC i

i: avar

The instructions DEC and INC decrement or increment respectively the value of the specified variable, in other words subtracting or adding one (1) to its value. These instructions can work with floating-point variables, but are substantially faster with integers.

Examples:

```
x%=4
y%=7
DEC x%
INC y%
PRINT x%,y%
```

--> The numbers 3 and 8 appear.

```
a|=255
INC a|
INC a|
PRINT a|
```

--> Results in '1' being displayed, as there was no overflow check.

ADD x,y
SUB x,y
MUL x,y
DIV x,y

x: avar
y: aexp

The instruction ADD increases the variable x by the value y, while SUB reduces x by y. MUL multiplies x by y and assigns the result to x. DIV divides x by y and puts the result in x. With these instructions x must be a numeric variable, whilst y must be a numerical expression. These instructions can work with floating point variables, but are substantially faster with integers.

DIViding a variable by zero will result in an error.

Example:

```
x%=1
y%=2
z%=3

ADD x%,y%          ! now x is equal to 3
SUB z%,(x%-1)/2    ! the numerical expression
'                  ! (x%-1)/2 results in 1
PRINT x%,z%
```

--> The numbers 3 and 2 are displayed.

PRED(), SUCC()
ADD(), SUB(), MUL(), DIV(), MOD()

PRED and SUCC determine the next higher or next lower number. ADD(), SUB(), MUL(), DIV() and MOD() offer quick integer arithmetic with Polish notation.

PRED(i)
SUCC(i)

i: iexp

PRED returns the next lower and SUCC the next higher number than the argument. So PRED and SUCC supply the predecessor or the successor of a numerical expression. Note that these functions operate on integer expressions, so that any decimal places are ignored. This gives rise to the effect that, for instance, PRED(2.1) returns the result 1, not 2. These two functions will also operate on string expressions (see the section on Character String Manipulation).

Example:

```
i%=6
j%=PRED(i%)
PRINT j%,SUCC(2),PRED(3*i%)
```

--> On the screen, the numbers 5, 3 and 17 appear.

ADD(x,y)
SUB(x,y)
MUL(x,y)
DIV(x,y)
MOD(x,y)

x,y: iexp

These functions can replace more usual expressions thus:

```
ADD(x,y)  corresponds to x+y
SUB(x,y)  corresponds to x-y
MUL(x,y)  corresponds to x*y
DIV(x,y)  corresponds to x/y  or x DIV y
MOD(x,y)  corresponds to x MOD y
```

Since these functions use integer arithmetic, any decimal places are ignored. After the following program segment is run ...

```
x%=5 y%=4
ADD y%,3
z%=SUB(x%,3)
```

... x% will have the value 5, y% the value 7 and z% the value 2. The functions ADD, SUB, MUL, DIV and MOD can be interleaved at will. This method of notation is known as 'Polish'.

Examples:

```
DEFINT "a-z"
x=4
y=ADD(x,x)                    ! y becomes equal to 8
z=SUB(x,2)
PRINT y,z,ADD(x,MUL(y,2))
```

--> The numbers 8, 2 and 20 appear on the screen.

```
DEFINT "a-z"
x=2
y=MUL(x,3)                    !y becomes equal to 6
PRINT y,DIV(8,x),MOD(11,4)  !MOD(11,4) is equal to 3
```

--> The numbers 6, 4 and 3 are printed.

```
DEFINT "a-z"
x=5
y=ADD(SUB(x,2),MUL(3,4))
PRINT y,DIV(8,MOD(14,4))
```

--> Displays the numbers 15 and 4.

Bit Operations

BCLR, BSET, BTST, BCHG
SHL, SHR, ROL, ROR
AND(), OR(), XOR(), IMP(), EQV()
SWAP()
BYTE(), CARD(), WORD()

These Bit Operations affect numerical expressions at bit level. The commands BCLR, BSET, BTST and BCHG are direct implementations of the 68000 instructions. Note, however, that they are used as functions, not commands.

They reset, set, test and negate individual bits. SHL, SHR, ROL and ROR shift or rotate. AND, OR, XOR, IMP and EQV are logical functions. In the following explanations the convention is used that bit 0 is the least significant bit. With 4-byte integers bit 31 is the most significant bit and is also the sign bit (if the sign bit is set, i.e. =1, then a negative number is represented in two's complement form, otherwise it is a positive number).

SWAP exchanges the high- and low-order words of a 4-byte value. BYTE reads the lower 8 bits and CARD the lower 16 bits of an expression. WORD extends a word to form a long word, i.e. bit 15 is copied into positions 16 to 31.

BCLR(x,y)
BSET(x,y)
BCHG(x,y)
BTST(x,y)

x,y: iexp

These functions permit the resetting, setting, negating and testing of bits. The bit numbers are counted starting from 0 on the 'right' and are internally ANDed with 31, so that they are always taken as being between 0 and 31.

The function BCLR sets the y-th bit of the numerical expression x to zero. Similarly, BSET sets the y-th bit of x to 1. BCHG sets bit y of x to 1 if it was 0, or resets it to 0 if it was 1. The function BTST results in -1 (TRUE) if bit y of x is equal to 1 and 0 (FALSE) if it is equal to 0.

Examples:

```
x=BSET(0,3)
PRINT x,BSET(0,5)
```

--> The numbers 8 (2^3) and 32 (2^5) appear on the screen.

```
REPEAT
  t|=Inp(2)
  PRINT CHR$(t|),CHR$(BCLR(t|,5))
UNTIL CHR$(t|)="x"
```

--> If Caps Lock is off, this program prints the letter corresponding to the key pressed, in both lower and upper case. (With lower case letters, bit 5 is always set; resetting this bit forces the transformation to upper case).

```
s$="TESTcase"
FOR i%=1 TO LEN(s$)
  PRINT CHR$(BCHG(ASC(MID$(s$,i%)),5));
NEXT i%
```

--> Displays testCASE on the screen. Each lower case letter is changed to upper case, and vice versa. (This will not work with umlauted characters).

SHL(x,y) SHL&(x,y) SHL|(x y)
SHR(x,y) SHR&(x,y) SHR|(x,y)
ROL(x,y) ROL&(x,y) ROL|(x,y)
ROR(x,y) ROR&(x,y) ROR|(x,y)

x,y: iexp

These instructions **SH**ift or **RO**tate the numerical expression x by y bits. If the variable type is not specified this will occur over long word length (4 bytes). If word length is specified (with &), the operation takes place over 2 bytes, and with byte length (|) over one byte.

The third letter of the function name specifies the direction for the shift or rotation, 'L' standing for left, and 'R' for right.

With word-length operations (with &) bit 15 is copied to bits 16 to 31, thus preserving the sign. With byte-length operations bits 8 to 31 are set to 0.

The following tables show the effect of the shift instructions:

x%	SHL|(x%,1)	BIN$(x%,16)	BIN$(SHL|(x%,1),16)
18	36	00000000 00010010	00000000 00100100
642	4	00000010 10000010	00000000 00000100

x%	SHL&(x%,1)	BIN$(x%,16)	BIN$(SHL&(x%,1),16)
18	36	00000000 00010010	00000000 00100100
130	260	00000000 10000010	00000001 00000100

x%	SHR&(x%,2)	BIN$(x%,16)	BIN$(SHR&(x%,2),16)
24	6	00000000 00011000	00000000 00000110
4162	1040	00010000 01000010	00000100 00010000

(**Note:** the bit representations have been grouped as two sets of 8 bits for the sake of clarity. BIN$ does not do this.)

The next examples concern rotation. Here, bits which leave one end of the argument are pushed in again at the other end. For instance, if only the highest bit of a byte is set, and the byte is then shifted one bit to the left (ROL|(128,1)), the set bit which disappears off the left end is re-introduced on the right as bit 0. With a shift to the left, (say SHL|(128,1)) the set bit would have been lost, with bit 0 being made zero.

Examples of rotation are:

x|	ROL|(x|,1)	BIN$(x|,1)	BIN$(ROL|(x|,1),8)
6	12	00000110	00001100
130	5	10000010	00000101

x|	ROR|(x|,3)	BIN$(x|,8)	BIN$(ROR|(x|,3),8)
66	72	01000010	01001000
2	64	00000010	01000000

Example:

```
x|=128+1                              ! bits 7 and 0 set
y%=ROR|(x|,1)                         ! y becomes 192
PRINT SHL(y%,4),y%*2^4
PRINT SHL(ROR|(128+1,1),4)
```

--> The number 3072 is printed three times. The function SHL(a,b) is equivalent to a*2^b, if no bit is lost off the left end of the four-byte number a. The bit-shift formulation, however, is clearly quicker.

AND(x,y)
OR(x,y)
XOR(x,y)
IMP(x,y)
EQV(x,y)

x,y: iexp

These functions operate logically on two numerical expressions. The function AND returns a result in which only those bits are set which are set in both x and y. The result of OR contains bits set in the places in which bits are set in either x or y or both. XOR sets only those bits which are set in x or y but not both (or, to put it another way: XOR sets those bits which are different in x and y). IMP resets a bit to zero if the appropriate bit in x is set and in y is reset, otherwise the bit is set. EQV sets a bit of the result if the appropriate bits in x and y are both set, or both reset. (See the section on Logical Operators for the truth tables of these functions.)

Example:

```
x=3
y=2
z=AND(x,y)
PRINT z,OR(2,7),XOR(x,1+4+8)
```

--> On the screen, the numbers 2, 7 and 14 appear.

```
PRINT BIN$(15,4),15
PRINT BIN$(6,4),6
PRINT BIN$(IMP(15,6),4),"IMP(15,6)"
PRINT BIN$(EQV(15,6),4),"EQV(15,6)"
```

--> Displays:

```
1111            15
0110            6
0110            IMP(15,6)
0110            EQV(15,6)
```

SWAP(x)

x: iexp

The function SWAP re-formulates (if necessary) the numerical expression x as a long word (4 bytes) and exchanges its upper and lower words (2 bytes in each case). This could be useful when passing long word parameters to to an operating system routine which requires the words in reverse order.

(**NOTE** that this function has nothing to do with the instruction of the same name, which swaps values of variables.)

Example:

```
x=1044480
PRINT BIN$(x,32)
y=SWAP(x)
PRINT BIN$(y,32)
```

--> The following appears on the screen:

```
00000000000011111111000000000000
11110000000000000000000000001111
```

An example of SWAP in use might be:

```
~WIND_SET(0,13,SWAP(t%),t%,0,0)
```

BYTE(x)
CARD(x)
WORD(x)

x: iexp

BYTE returns the lower 8 bits of the numerical expression x, and CARD the lower 16 bits. WORD extends a word to long word length (32 bits) by copying bit 15 to bit positions 16 to 31, thus preserving the sign.

Examples:

```
PRINT BYTE(255),BYTE(1+255)
PRINT HEX$(CARD(&H1234ABCD))
```

--> 255, 0 and ABCD are printed.

```
x%=&HFFFF
PRINT BIN$(x%,32)
x%=WORD(x%)
PRINT BIN$(x%,32)
```

--> The screen displays:

```
00000000000000001111111111111111
11111111111111111111111111111111
```

CHAPTER 5
STRING MANIPULATION

LEFT$, RIGHT$
MID$ (as a function)
PRED, SUCC
LEN, TRIM$
INSTR, RINSTR
STRING$, SPACE$, SPC
UPPER$
LSET, RSET, MID$ (as an instruction)

These functions allow the manipulation of strings by selectively altering and concatenating their contents. The functions LEFT$ and RIGHT$ return the left or right part of a character string. MID$, used as a function, returns a section from the middle of a string, or, used as an command, allows the replacement of part of one string with all or part of another. PRED and SUCC return the character with the ASCII code one lower or one higher than the first character of the specified string, while LEN determines the length of a character string. INSTR and RINSTR search a particular string for the occurrence of another string. STRING$, SPACE$ and SPC generate strings composed of several identical strings, and UPPER$ transforms all lower case letters in a string to upper case. Left- and right-justified insertion of one string into another is accomplished with LSET and RSET.

LEFT$(a$ [,x])
RIGHT$(a$ [,x])

a$: sexp
x : iexp

LEFT$ returns the first x characters from the character string a$. If x is larger than the number of characters in a$, then the whole of a$ is returned. If x is not specified, the first character of a$ is returned.

RIGHT$ works similarly, except that it returns the last x characters of a$ and when x is not specified, the last character is returned.

Examples:

```
a$="Right-minded people use GFA BASIC"
b$=LEFT$("GFA Systemtechnik",4)
PRINT b$;RIGHT$(a$,5)
```

--> 'GFA BASIC' is printed on the screen.

```
a$="Oh, don't"
b$=LEFT$(a$)+RIGHT$("look")
PRINT b$
```

--> 'Ok' appears on the screen.

MID$(a$,x [,y]) (as a function)

a$: sexp
x,y: iexp

The function MID$ returns y characters starting from position x of the string a$. If x is larger than the length of a$ then a null string ("") is returned. If y is omitted, then the function returns the whole of the string from position x onwards.

Example:

```
a$="This is the GFA BASIC manual"
b$=MID$(a$,13,9)+MID$("version 3",8)
PRINT b$
```

--> On the screen, the text 'GFA-BASIC 3' appears.

PRED(a$)
SUCC(a$)

a$: sexp

PRED supplies the character with the ASCII code one less than that of the **first** character of the specified string. (In other words, its **PRED**ecessor in the ASCII table.)

Similarly **SUCC** returns the character with the ASCII code one greater than that of the **first** character of the string specified. (In other words, its **SUCC**essor.)

(These functions are also effective with numbers: see the section on Integer Arithmetic.)

PRED(a$) corresponds to the expression CHR$(ASC(a$)-1), and SUCC(a$) to CHR$(ASC(a$)+1).

Example:

```
character$="blue moon"
predecessor$=PRED(character$)
successor$=SUCC(character$)
PRINT predecessor$;SUCC(predecessor$);successor$
```

--> 'abc' is displayed on the screen.

LEN(a$)
TRIM$(a$)

a$: sexp

LEN determines the number of characters contained in a$.

TRIM$ removes spaces from the left and right ends of a string.

Examples:

```
a$="test"
x=LEN(a$)+1
PRINT x,LEN("word")
```

> Displays: the numbers 5 and 4.

```
b$="   test  "
PRINT LEN(b$)
PRINT TRIM$(b$)
PRINT LEN(TRIM$(b$))
```

> Displays 9, the word 'test' (without spaces) and 4.

INSTR(a$,b$)
INSTR(a$,b$,[x])
INSTR([x],a$,b$)

a$, b$: sexp
x : iexp

The function INSTR searches the character string a$ for an occurrence of the string b$. If x is specified, then the search begins at character position x in a$, otherwise the whole string is searched.

If b$ is found within a$, the character position at which it begins is returned, or, if it is not found, the function returns zero (0). This value may be assigned to a variable or used to determine whether a particular string was present in a$. If a$ and b$ are both null strings (""), one (1) is returned.

Example:

```
a$="GFA-Systemtechnik"
x=INSTR(a$,"System")
PRINT x,INSTR("GFA-BASIC","BASIC",6)
```

--> The numbers 5 and 0 appear on the screen.

```
REPEAT
a$="123456"
b$=INKEY$
UNTIL INSTR(a$,b$)
```

--> Waits until one of the specified keys has been pressed.

RINSTR(a$,b$)
RINSTR(a$,b$,[x])
RINSTR([x],a$,b$)

a$, b$: **sexp**
x: **iexp**

RINSTR() operates in the same way as INSTR except that the search for b$ begins at the 'right-hand' end of a$. (One could say that this is the Reverse of INSTR.)

Example:

```
PRINT RINSTR("a:\FOLDER\*.GFA","\")
```

> The string "\" is sought in the string "a:\FOLDER*.GFA", starting from the end. Its first occurrence is found at position 10.

STRING$(x,a$)
STRING$(x,code)
SPACE$(x)
SPC(x)

x,code : iexp
a$: sexp

The function STRING$ produces a string which reproduces the expression 'a$' (or the ASCII character whose number is 'code') x times, where x has a maximum value of 32767.

SPACE$(x) produces a string consisting of x spaces.

SPC inserts x spaces in a PRINT statement without the need to create a string variable or to explicitly specify the spaces by " ".

Example:

```
a$="GFA "
b$=SPACE$(5)
PRINT b$;STRING$(3,a$);SPC(4);STRING$(5,65)
```

> ' GFA GFA GFA AAAAA' appears on the screen.

UPPER$(a$)

a$: sexp

All lower case letters in a$ are converted to upper case. (This also happens with characters containing umlauts.)

Example:

```
a$=" test"
b$=UPPER$(a$)+UPPER$("ware")
PRINT UPPER$("Gfa basic 3");b$
```

> On the screen 'GFA-BASIC 3 TESTWARE' appears.

LSET a$=b$
RSET a$=b$

a$: svar
b$: sexp
x,y: iexp

LSET and RSET will set the string expression b$ into a$, justified either to the left or to the right. If b$ is shorter than a$ (the normal situation) spaces will be inserted to make up the original length of a$. Note that the actual content of a$ is irrelevant: only its length is significant.

Example:

```
a$="xxxxx"
FOR i%=1 TO 128
  LSET a$=STR$(i%)    ! left-justified format
  PRINT a$;
NEXT i%
PRINT
'
FOR i%=1 TO 128
  RSET a$=STR$(i%)   ! right-justified format
  PRINT a$;
NEXT i%
~INP (2)
```

--> Displays left- and right-justified formatted columns of numbers, then waits for a key to be pressed.

(a$=STR$(i%,5,0) could also be used in place of RSET a$=STR$(i%) .)

MID$(a$,x [,y]) (as an instruction)

a$: svar
x,y: iexp

MID$ used as an instruction makes possible the replacement of part of a string variable a$ with the string expression b$. So with MID$(a$,x,y)=b$, characters from b$ will overwrite those in a$, starting at the x-th character position of a$. The optional parameter y determines how many characters of b$ are used. If y is omitted, then as many characters as possible of a$ are replaced with those from b$. The length of a$ is unchanged, so that no characters will be written beyond the end of a$.

Example:

```
a$="GFA SYSTEMTECHNIK"
MID$(a$,5)="BASIC "
'
b$="Testword"
MID$(b$,6,10)="are you serious?"
PRINT a$,b$
```

--> 'GFA BASIC TECHNIK' and 'Testware' are printed.

CHAPTER 6
INPUT AND OUTPUT

Keyboard and Screen Handling

This first section begins with an examination of the command INKEY$, which takes a single character from the keyboard. Next, the family of INPUT commands is examined, along with the related commands LINE INPUT, FORM INPUT and FORM INPUT AS. The discussion of output capabilities begins with the most simple instruction, PRINT, and goes on to extended versions such as PRINT AT and PRINT USING. CRSCOL, CRSLIN and POS are used to report on the current cursor position, and TAB, HTAB and VTAB are used to control it. At the end of this section, the KEYxxx commands are considered, these being a family of commands for the interrogation of the keyboard, and its configuration while a program is running.

INKEY$

INKEY$ reads a character from the keyboard. This function will, however, not detect depressions of the Shift, Alternate or Control keys alone. INKEY$ does not wait for a key to be pressed, but scans the keyboard to check whether a key has been pressed since the last scan. If it has, the command accepts that pressed key, otherwise a null string ("") is returned. When the key pressed has no ASCII code, for example a Function key or the HELP or UNDO keys, then INKEY$ returns the scan code of the pressed key as a two-character long string containing CHR$(0) as its first character, and the identification code of the special key as the second character.

The following example demonstrates how the values returned by INKEY$ can be examined:

```
DO
  t$=INKEY$
  IF t$<>""
    IF LEN(t$)=1
      PRINT "Character: ";t$,"ASCII code:";ASC(t$)
    ELSE
      PRINT "CHR$(0)+Scan code ";CVI(t$)
    ENDIF
  ENDIF
LOOP
```

--> Displays the ASCII or Scan code of each key pressed.

Note: CVI turns a 2-byte string into an integer. As the first byte of the string is zero, in this case it returns the value of the second byte.

INPUT ["text",] x [,y,...]
INPUT ["text";] x [,y,...]

x,y: avar or svar

The command INPUT can be used in several ways. It accepts the input of variables or variable lists with or without a text message being displayed on the screen. For INPUT, the cursor will normally retain its last screen position; however, by means of PRINT AT followed by a semicolon, or by using LOCATE, VTAB, or HTAB, the cursor can be put at a desired screen position.

A text string may follow the INPUT command, separated from the following variables by a comma or a semicolon. If a semicolon is used, then a question mark and a space are printed on the screen and the cursor placed at the succeeding character position. When a comma is used, the question mark and space are omitted, and the cursor is placed directly after the last character of the text string.

If no text is to be displayed, then the question mark and space are printed, and the cursor is put directly after the space (as if a null string ("") had been used as text, followed by a semi-colon).

When only one variable is to be input, the user types in a number or a character string and terminates it by pressing either the RETURN or the ENTER key. When several values are to be entered (if there was a list of variables after the INPUT statement in the program line), each individual variable can be terminated by pressing RETURN or ENTER, or they may be typed in separated by commas and all confirmed together with a single press of the RETURN or ENTER key.

If a string is to be entered which may contain commas, the instruction LINE INPUT must be used.

If a numeric variable was expected by the INPUT statement, and a non-numeric string typed in instead, a bell signal sounds, and the input must be repeated.

Prior to the RETURN or ENTER key being pressed the input can be edited by means of the Backspace, Delete, and Left and Right arrow keys. Pressing the INSERT key switches between 'insert' and 'overwrite' modes during editing. The maximum length of the input is 255 characters.

Special symbols can be entered in three different ways:

- By holding down the ALTERNATE key and typing in the ASCII code of the desired character using the numeric pad. When the ALTERNATE key is released, the appropriate character is displayed. For instance, with 64 as the ASCII code, the character '@' appears. This also works with INKEY$, INP(2), GEM Dialog Boxes etc., if it is not switched off by the command KEYPAD.

- By typing Control-S followed by another character, for example 'Control-S C' for the Pi character. (Press the Control and S key at the same time, then press C.) This feature only works with the INPUT statement or in Edit mode, when a program is actually being typed in.

- By typing Control-A followed by the ASCII code of the desired character, e.g.: 'Control-A 64' for the '@' character.

Example:

```
INPUT a$
INPUT "",b$
INPUT "Enter two numbers: ";x,y
PRINT a$,b$,x,y
```

--> Reads in two strings and two numeric variables. The first input command generates a '? ', the second appears with no text and the third issues the message 'Enter two numbers: ? '

LINE INPUT ["text",] a$ [,b$...]
LINE INPUT ["text";] a$ [,b$...]

a$,b$: svar

LINE INPUT is a variant of the INPUT command. Unlike INPUT, it allows commas to be accepted as part of the input. The preceding description of the input of variables or a variable list, and the facilities for correction of the input prior to the pressing of RETURN or ENTER also apply to LINE INPUT. LINE INPUT, however, may only be used with string variables.

Example:

```
LINE INPUT a$
INPUT b$
PRINT a$,b$
```

--> When the program is Run, and the text 'com,ma' inputted twice, a$ is taken as 'com,ma' whilst b$ contains 'com', as the 'ma' is taken as an separate input and ignored. See also LINE INPUT #.

FORM INPUT n,a$
FORM INPUT n AS a$

n : iexp
a$: svar

FORM INPUT and FORM INPUT AS are both used to input string variables. The value of n specifies the number of characters to be entered, up to a maximum of 255.

Additionally, FORM INPUT AS displays the current value of a$, which can then be edited, or taken as a 'default' input by pressing RETURN or ENTER immediately. The editing facilities are the same as those for the INPUT command.

Example:

```
FORM INPUT 10,a$
b$="test"
FORM INPUT 5 AS b$
PRINT a$,b$
```

--> This asks for two strings to be entered. On the entry of the second string, the word 'test' is displayed as a preselected value of b$. This may then be edited, or accepted by pressing RETURN.

PRINT
PRINT expression
PRINT AT(column,row);expression
WRITE expression
LOCATE row,column

expression : arbitrary combination of sexp or aexp
column,line : iexp

The instruction PRINT, without parameters, causes a blank line to be printed. If the cursor happens to be on the last line, the entire screen is scrolled up one line. PRINT followed by an expression causes the expression to be printed at the current cursor position. Text strings must be enclosed in quotes. If the expression to be printed consists of several elements (constants, variables or expressions), the individual parts must separated by a semicolon, comma or apostrophe, with the following effects:

- A comma causes the cursor to move to one position past the next column number which is divisible by 16. In other words, the columns 1, 17, 33, etc. are used. If this takes the cursor beyond the end of the line, the cursor moves to column 1 of the next line.

- A semicolon causes the output of the corresponding elements, one after the other without any spaces being added.

- An apostrophe inserts one space between the appropriate elements.

PRINT AT makes it possible for the expression to be displayed on a specified row, starting at a specified column. Depending on the current screen resolution, up to 80 columns and 25 rows are available.

Note: Row and Column numbers begin at 1, not 0.

If a window is open, the column and row numbers are relative to the top left-hand corner of the window.

If the output expression (the message) is not terminated with a semicolon, the cursor is placed at the beginning of the next row. If it was already at the bottom of the screen, then the screen is scrolled up by one row.

If print control characters (ASCII characters from 0 to 31) are specified, then these are processed by the VT-52 Emulator (see Appendix). Similarly to PRINT AT, LOCATE places the cursor at the specified **column and row**, i.e. in the **reverse** order to PRINT AT. It will not, however, actually display anything. (See VTAB and HTAB.)

The **WRITE** command can be used to send output to the screen. The command is followed by the numerical or string expressions to be sent, separated by commas. The output INCLUDES THE COMMAS and, with string expressions, the quotation marks are also included. A semi-colon may be placed after the last item to be output, in which case the usual Carriage Return/Line Feed will be suppressed.

Examples:

```
a$="GFA Systemtechnik"
PRINT LEFT$(a$,4)+"BASIC"'1+2;
PRINT ".0","GFA ";UPPER$(MID$(a$,5))
```

-->Displays 'GFA BASIC 3.0' and 'GFA SYSTEMTECHNIK' .

```
PRINT AT(5,8);"Fifth column, eighth row"
```

--> Prints the message starting at the position x=5, y=8.

```
LOCATE 8,5
PRINT "Eighth row, fifth column"
```

--> Positions the cursor at the fifth column of the eighth row and displays the message, again x=5 and y=8.

```
WRITE 1+1,"Hello",3*4
```

--> The output is: ' 2,"Hello",12 '

PRINT USING format$,expression [;]
PRINT AT(column,line);USING format$,expression [;]

format$: **sexp**
expression : **as many aexp or sexp as wanted, separated by commas**
column,row : **iexp**

PRINT USING and its variant PRINT AT USING serve to format data output on the screen. These commands work in the same general way as PRINT or PRINT AT. However, the data in the expression to be displayed are formatted according to the contents of format$.

The following symbols may be used to format a numerical expression:

- # Reserves a character position for a number (digit).
- . A full-stop specifies the position of the decimal point within several '#' symbols
- , A comma in the appropriate place between '#' signs causes the insertion of a comma, e.g. as a 'thousands' separator in 10,000.
- - The minus sign reserves a character position for a minus sign, in case the number to be printed is negative. For a positive number a space will be printed instead. (GFA BASIC normally prints nothing preceding a positive number.)
- + Reserves a space for the plus sign of a positive number. If the number is positive, the '+' sign will be printed, but if it is negative a minus sign will appear as usual. (Cf. '-' above.)
- * Replacement for # above, with the difference that leading zeros are replaced by asterisks, instead of blanks.
- $ Causes one dollar '$' character to be printed before a number, provided it is put directly before the first '#' symbol.

^ Specifies the length of the exponent (including the 'E+') when a number is to be printed in exponential format (e.g. E+123). If the number of digits before the decimal point is specified using several '#' symbols, then the exponent is adjusted to take account of this. for a negative exponent, the possibility of a '-' must be allowed for by '-' above.

In contrast to GFA BASIC Version 2, the principle 'Formatting First' is adhered to. That is, in the case of overflows only the digits with character positions reserved for them will be displayed. Thus, great care must be taken to ensure that the string to be printed can actually be accommodated within the format specified.

For formatting strings the following symbols are available:

& Causes the output of the entire character string.

! The output is limited to the first character of the string.

\...\ Specifies the number of characters of the string to be printed. (The counting includes the two '\' symbols).

_ Causes the output of the character following the underline ('_') character. A sequence of characters can be output by placing it between two '_' characters.

Examples:

```
PRINT USING "#.####",PI
PRINT AT(4,4);USING "PI_._._. #.###",PI;
```

--> Displays '3.1416' and 'PI... 3.142'.

```
FOR i%=1 TO 14
PRINT USING "###.##^^^^ ",2^i%;
NEXT i%
```

--> Displays:

```
1.00E+00  2.00E+00   4.00E+00   8.00E+00  16.00E+00
32.00E+00 64.00E+00 128.00E+00 256.00E+00 512.00E+00
1.02E+03  2.05E+03   4.10E+03   8.19E+03  16.38E+03
```

(The functions of commas and full stops can be exchanged using the MODE command: see overleaf.)

MODE n

n: iexp

With MODE the representations of the decimal point and the 'thousands comma', as interpreted by PRINT USING (and also by STR$ with 3 parameters) can be reversed. This allows the use of the continental method of representing numbers.

In addition, MODE selects the format of the date representation used by DATE$, SETTIME, DATE$= and FILES. The parameter n can be between 0 and 3. The following may then be used:

MODE	USING	DATE$
0	#,###.##	16.05.1988
1	#,###.##	05/16/1988
2	#.###,##	16.05.1988
3	#.###,##	05/16/1988

DEFNUM n

n: iexp

DEFNUM affects the output of numbers by the PRINT command and its variants. All numbers following the DEFNUM instruction are outputted to occupy n character positions, not counting the decimal point. Rounding takes place based on the value of the (n+1)th digit. The internal computational accuracy is not affected.

Example:

```
PRINT 100/6
DEFNUM 5
PRINT 100/6
```

--> The numbers 16.66666666667 and 16.667 appear on the screen.

CRSCOL
CRSLIN
POS(x)
TABA(n)
HTAB column
VTAB line

n,column,row: iexp
x : aexp

The command group comprising CRSCOL, CRSLIN, POS and TABA serve to return the location of the cursor, or to place the cursor at a particular position.

CRSCOL returns the current column position and CRSLIN the current line position of the cursor.

POS supplies the number of characters displayed on the screen (ANDed with 255 to give a maximum of 255) since the last Carriage Return. The expression x is ignored. The value returned by POS may, however, not agree with CRSCOL e.g. if a string 120 characters long is displayed, POS(0) in this case will return the value 120, whereas CRSCOL will return 41, the column number of the next character to be printed counting from the left edge. With the output of control characters POS(0) has even less to do with the current cursor column reported by CRSCOL, since POS will keep track of non-printed characters, e.g. ESCape = CHR$(27). POS will, however, ignore Line Feeds - Chr$(10) -, be reduced by one by the printing of a Backspace - Chr$(8) - or be reset to zero by a Carriage Return - Chr$(13).

TAB(n) prints spaces until POS(0) reaches n. If POS(0) already exceeds n, then a Line Feed/Carriage Return is executed first. The value of n is limited to a maximum of 255 by ANDing it internally with 255.

The instructions HTAB and VTAB position the cursor to the specified column or line number. Note that cursor columns and lines are counted from 1, not 0.

Examples:

```
PRINT AT(38,12);"Test";
PRINT CRSCOL'CRSLIN
PRINT TAB(37);"Test";
PRINT POS(0)
INP(2)
```

--> On the screen, 'Test42 12' appears and, underneath it, 'Test41'. (POS(0) returns 41, not 42, as it is counted from zero rather than 1 as with CRSCOL).

```
PRINT AT(4,3);"Word 1"
HTAB 4
VTAB 2
PRINT "Word 2"
```

--> 'Word 1' appears starting at row 3, column 1, and 'Word 2' at row 2, column 4.

KEYxxx

This group of functions makes it possible to read the status of the keyboard shift keys while a program is running as well as assigning freely definable character strings (max. 31 characters each) to the Function Keys, which will also be available when using the GFA BASIC Editor.

KEYPAD n

n : iexp

The numerical expression n is evaluated bit by bit and has the following meaning:

Bit	Meaning	0	1
0	NUMLOCK	On	Off
1	NUMLOCK	Not switchable	Switchable
2	CTRL-KEYPAD	Normal	Cursor
3	ALT-KEYPAD	Normal	ASCII
4	KEYDEF without ALT	Off	On
5	KEYDEF with ALT	Off	On

With bit 0 set the keypad will act as a 'PC' keypad with NUMLOCK off, i.e. it responds with cursor movements.

With bit 1 set the 'PC' NUMLOCK mode can be toggled with Alternate and '-', otherwise it cannot.

With bit 2 set, NUMLOCK is effectively switched off while the CONTROL key is held down. Thus CONTROL-4 (on the keypad) produces cursor movements.

With bit 3 set ASCII values for characters can be typed in with the ALTernate key held down. When ALT is released, the character appears. With bit 4 set, the character strings assigned with KEYDEF to the keys F1 to F10 and Shift-F1 to Shift-F10 are output when the key is pressed. With bit 5 set the ALTernate key must also be held down.

When turned on, the ATARI ST is effectively configured to KEYPAD 0. With GFA BASIC in operation, the default keypad mode is decimal 46, i.e. bits 1, 2, 3 and 5 are set.

KEYTEST n
KEYGET n
KEYLOOK n

n: ivar

KEYTEST is similar to INKEY$ and reads a character from the keyboard. If no key was pressed since the last input (apart from Alternate, Control, Shift and Caps Lock) the returned value is zero, otherwise its value corresponds to the key in the fashion shown below for KEYGET.

KEYGET waits for a key to be pressed and then returns a long word value corresponding to the key. This 32-bit long word is constructed as follows:

Bits 0-7 : the ASCII code
Bits 8-15 : Zero
Bits 16-23 : the Scan code
Bits 24-31 : Status of Shift, Control, Alternate, Caps Lock as follows:

Bit	Key
0	Right shift
1	Left shift
2	Control
3	Alternate
4	Caps lock

KEYLOOK allows a character to be read from the keyboard buffer, without changing the buffers contents, as with KEYGET or INKEY$.

Using KEYTEST, KEYGET and KEYLOOK with byte- or word-sized variables results in conversion from an integer value taking place automatically. The result wiil be placed in the variable n.

Examples:

```
PRINT "Please press ESCape"
REPEAT                          ! The program runs
'                               ! through the loop
UNTIL INKEY$=CHR$(27)           ! until the ESCape key
'                               ! is pressed.
'
PRINT "Please press ESCape again"
REPEAT                          ! Again runs through
  '                             ! the loop but
  KEYTEST n|                    ! this time checks the
  '                             ! keyboard here for
UNTIL n|=27                     ! character number 27.
```

--> Waits twice for the ESCape key to be pressed. Note that in the second example bits 0 to 7 are masked off automatically by using a byte-sized (8-bit) variable.

```
PRINT "Please press a key"
key_1|=INP(2)                   ! Waits for a
'                               ! key-press.
PRINT "Press another key please"
KEYGET key_2|                   ! Waits again.
PRINT "INP(2) : ";key_1|        ! The ASCII codes
'                               ! of the depressed
PRINT "KEYGET : ";key_2|        ! keys are printed
```

--> Waits twice for keys to be pressed, and displays their ASCII codes.

```
REPEAT
  KEYLOOK n%
UNTIL n%
INPUT "Type in something ";a$
```

--> Waits in a loop until a key is pressed, then begins the input of a$ without losing the initial character.

```
DO
  KEYGET a%
  PRINT HEX$(a%,8),BIN$(a%,32),
  OUT 5,a%
  PRINT
LOOP
```

--> Waits for a key to be pressed and displays the value returned by KEYGET in both hexadecimal and binary.

(Note: OUT 5,a% allows the characters associated with control codes 0-32 to be displayed.)

KEYPRESS n

n: iexp

The command KEYPRESS simulates the pressing of a key. That is, the character with the ASCII code contained in the lowest-order 8 bits of n is added to the keyboard buffer. Additionally, the state of the Control, Shift and Alternate keys may be passed in the highest-order byte, as defined by KEYGET. If the ASCII code given is zero, a scan code may be passed in bits 16 to 23, e.g. KEYPRESS &H3B0000 presses F1.

Examples:

```
FOR i&=65 TO 90        ! Simulates the pressing
  KEYPRESS i&          ! of the keys A-Z ....
NEXT i&
KEYPRESS 13           ! .... followed by a Carriage
Return
INPUT a$               ! Characters are taken up to
the first Carriage Return
PRINT a$
```

--> The letters from A to Z are printed.

```
KEYDEF 1,"Hello"+CHR$(13)
KEYPRESS &H83B0000
PAUSE 1
LINE INPUT a$
PRINT a$
```

--> A string to be produced by pressing ALT-F1 is defined, then a press of this key is simulated, followed by a slight pause to give the associated interrupt routine time to process it. The word "Hello" is then taken as the inputted a$.

KEYDEF n,s$

n : iexp
s$: sexp

The command KEYDEF makes it possible to assign an arbitrary character string (with a maximum length of 255 characters) to the Function Keys. The arithmetic expression n (with a value from 1 to 20 inclusive) determines the key. A value of n from 1 to 10 refers to the keys F1 to F10, and from 11 to 20, to the keys Shift-F1 to Shift-F10. The defined string is available both during the running of a program and from the GFA BASIC Editor. However, in the Editor the ALTernate key must also be pressed, as otherwise the GFA BASIC commands would not be available.

Example:

```
KEYPAD 16              ! Sets ALT key as not required
DO
  KEYDEF 1,"F1"              ! String for F1 key
  KEYDEF 11,"Shift+F1"    ! String for Shift-F1 key
  INPUT x$
LOOP
```

--> Instead of having to type in a string as the input, pressing F1 automatically supplies the string 'F1', and pressing Shift-F1 supplies the string 'Shift+F1'.

Data Input and Output

This section first describes the storing and recalling of constants with DATA, READ and RESTORE. Then file management is dealt with, starting with the indexing commands DIR$, CHDIR, DIR, FILES, MKDIR and RMDIR, and explaining the structure of the hierarchical filing system. Next comes the opening, closing, renaming etc. of files with EXIST, KILL, NAME, OPEN and CLOSE followed by the facilities for storage of areas of memory with BLOAD, BSAVE, BGET and BPUT. Then sequential input and output with INPUT$, INPUT# and PRINT# are covered, together with indexed sequential accessing of files (SEEK, RELSEEK), as well as random accessing (FIELD, GET#, PUT#, SEEK#, RELSEEK#). The section on peripheral devices describes byte by byte input and output with INP, OUT and their 'query' counterparts INP? and OUT?, and input from the serial and MIDI interfaces (INPAUX$, INPMID$).

Finally, the commands are given for the handling of the mouse and joysticks (MOUSE, MOUSEX, MOUSEY, MOUSEK, HIDEM, SHOWM, STICK, STRIG), and printer output (LPRINT, LPOS, HARDCOPY).

Data Commands

DATA const [,const1,const2,...]
READ var [,var1,var2,...]
RESTORE [label]

const,const1,const2 : numerical or string constants
var,var1,var2 : avar or svar
label : user-defined label

DATA statements are used to store numeric or string constants which can then be accessed with READ. Numeric values can be specified in hexadecimal, octal or binary form, but string constants must be enclosed in inverted commas if they contain commas, as commas are normally used to separate items in the DATA statement.

Internally associated with DATA and READ is the so-called data pointer, which always points to the next item to be READ. When a program is Run, this is the first item in the first DATA statement. The RESTORE instruction allows the data pointer to be moved so as to point to any DATA statement, provided the DATA statement is preceded by a label. If no label is specified by RESTORE, the data pointer is moved to the first DATA statement in the program.

The label can consist of numbers, letters, underline characters ('_') or full stops, and, unlike variable names, can begin with a number. The label occupies a line by itself and must end with a colon, although the colon is not used when the label is referred to elsewhere in the program.

Example:

```
FOR i=1 TO 3
  READ a
  PRINT a'
NEXT i
'
RESTORE roman_numbers
READ a$,b$,c$,d$
PRINT
PRINT a$'b$'c$'d$
'
DATA 1,2,3,4
DATA a,b,c,d
'
roman_numbers:
DATA I,II,III,IV
```

--> In the loop the numeric values 1,2 and 3 are read in and printed on the screen. After that, by means of RESTORE, the data pointer is moved to the line containing the roman numerals as data. They are also read in and printed.

```
DATA 10,&A,$A,&HA,&O12,&X1010,%1010
FOR i%=1 to 7
  READ a%
  PRINT a%
NEXT i%
```

The number 10 is read in seven times. Like INPUT, VAL etc., READ can interpret hexadecimal numbers if they are prefixed with '$' or '&H', and binary numbers if they are prefixed with '%' or '&X'.

File Management

In the following section, instructions related to file organisation are explained.

First, however, it is important to know the structure of a file specification and the rules of the hierarchical filing system. A file specification consists of three parts: the drive specification, the file name and the filename extension. The drive specification contains the disk drive identification in the range A to O, followed by a colon. The file name is up to 8 characters in length, with, optionally, an extension comprising a full stop and up to 3 characters. Groups of files may be gathered into directories (also called folders), which may themselves be gathered into different sub-directories (or folders) and so on. The lowest level of grouping (which contains all the files on the disc in their respective directories and sub-directories) is known as the root directory. Starting from this root directory, directories may be accessed, followed by sub-directories, etc. Therefore, for a file to be accessed, the following information must be given:

- Drive specification
- Name of directory if any, sub-directory if any, etc.
- The actual name of the file and its extension, if any

These parts are separated by reverse diagonal strokes "\" (backslashes). Names for directories have the same format as file names. The access path for a file is a combination of these elements.

For example:

```
A:\TEXT.DOC\MANUAL\CHAPTER_1.DOC
```

This means that the file to be accessed, named CHAPTER_1, has the extension .DOC and is in a sub-sub-directory called MANUAL, which itself is in a sub-directory called TEXT.DOC. TEXT.DOC is in the root directory of disk drive A:.

Two special symbols are available to make file selection easier, which can be used within file names and their extensions. These are the question mark ('?') and the asterisk ('*'). The question mark acts like a 'wild card' and will be accepted as any character whose ASCII value is greater than 32. The asterisk is similar, except that it can be taken as any sequence of characters in the file specification.

The command DIR will be explained in a moment, but for now, from the Direct Mode of GFA BASIC (reached by pressing ESCape), you can type in 'DIR', which will cause all the files on drive A: to be displayed.

Typing 'DIR "*.GFA"' lists those files which have the extension '.GFA'.

Typing 'DIR "?AB?.*A"' lists those files which have four-letter file names with the middle two letters 'AB', and an extension ending in 'A'.

Typing 'DIR "*.*"' lists all files.

Directory Handling

DFREE(n)
CHDRIVE n or n$
DIR$(n)
CHDIR name$

n : **iexp**
name$: **sexp**

DFREE (disk free) returns the amount of space free for storage on drive n in bytes. This can take a few seconds, or longer with a partitioned hard disk drive. See CHDRIVE below for the meaning of the parameter n.

CHDRIVE (Change Drive) sets the default disk drive. This is the drive which is used by DIR etc. if no other drive is specified. Any drive can be made the default drive with CHDRIVE followed by the drive number from 0 to 16. Drive 0 corresponds to the current default drive, drive 1 corresponds to A:, drive 2 to B:, etc. The argument of CHDRIVE may also be a string, in which case the first character (from A to P) identifies the drive.

DIR$(n) returns as a string the current access path for drive n, as set with CHDIR below. See CHDRIVE above for the meaning of the parameter n.

CHDIR sets the current directory. Since the default drive cannot be changed with CHDIR, the specified directory must be on the current default drive or a specified drive.

So, after CHDIR "B:\TEST", the directory TEST on drive B: becomes the current directory, which may be accessed without the need to specify the path 'B:\TEST'. With CHDIR "\", one can return to the root directory from any sub-directory. The directory required is always looked for as a sub-directory of the current directory. Thus if, after making B:TEST the current

directory as above, one issued the command CHDIR "TEST2", the effective path from the root directory would be B:\TEST\TEST2.

There are two special folder names: '.' and '..', which are shorthand ways of referring to the current directory path and the 'parent' directory path respectively. The parent path is that which leads up to, but does not include, the current directory. For instance, with the path 'B:\TEST\TEST2' above, the parent path is 'B:\TEST'. Thus to change the current directory to another on the same level, perhaps TEST2A, it is only necessary to type 'CHDIR "..\TEST2A"'.

Examples:

```
CHDRIVE 1
PRINT DFREE(0)
PRINT DIR$(2)
CHDRIVE "C:\"
```

--> Disk drive A: is selected as the current default drive, and its free storage capacity is printed. Then the current access path for drive B: is printed, and finally drive C: is made the current default drive.

```
CHDIR "\"
CHDIR "TEXT.DOC\MANUAL"
CHDIR "APPENDIX"
```

--> CHDIR "\" sets the current directory to be the root directory of the current drive. In the second line MANUAL, which is a sub-directory of the sub-directory TEXT.DOC, is made the current directory. In the third line a further sub-directory APPENDIX is made the current directory, so that the access path is now \TEXT.DOC\MANUAL\APPENDIX.

DIR p$ [TO name$]
FILES p$ [TO name$]

p$,name$: sexp

The instructions DIR and FILES allow directories to be printed out, or sent to a specified device or file with TO.. If no device is specified, the output goes to the screen. The desired access path and file mask (e.g. *.*) is specified in the expression p$. FILES is very similar to DIR, with the difference that it also provides information on the length, time and date of the listed files, and will also list folder names, identified by a '*' prefix, if they fulfill the conditions of the file mask.

If p$ ends in '\', with no file mask, the mask '*.*' will be added automatically by the BASIC.

With the optional extra 'TO name$' the directory information can be sent to a peripheral device or to a file. In this case name$ must contain the device identifier (e.g. "LST:") or a file specification (e.g. "A:\CONTENTS.LST").

Examples:

```
DIR "A:\BOOKS\*.DOC"
```

--> All files with the extension .DOC in the folder BOOKS on drive A: are listed on the screen.

```
DIR "A:\BOOKS\MANUAL\*.DOC" TO "B:\MANUAL\CONTENTS.ASC"
```

--> The list of files ending in .DOC in the folder MANUAL, which itself is found in the folder BOOKS, is sent to the file CONTENTS.ASC in the folder MANUAL on drive B:.

```
DIR "A:\*.*" TO "PRN:"
```

--> All the files on drive A: are listed on the printer.

```
FILES "A:\*.DOC" TO "LST:"
```

--> All the files on drive A: which have the extension .DOC are listed on the printer.

FGETDTA()
FSETDTA(addr)

addr: iexp

The DTA (disk transfer address) can be read with the function FGETDTA() or set with the instruction FSETDTA. The default address of the DTA is BASEPAGE+128, but it is changed when a File Select Box is used. This address is used by DIR, FILES and EXIST.

The DTA has the following structure:

Offset	Bytes	Meaning
0	21	Reserved for GEMDOS
21	1	File attributes (see below)
22	2	Time
24	2	Date
26	4	File length
30	14	File name terminated with null byte, without blanks

The meaning of the attribute bits is:

Bit	**Meaning** (if the bit is =1)
0	The file is write protected
1	The file is hidden (excluded from DIR search)
2	The file is a System file (excluded from DIR search)
3	Disk label
4	Folders
5	Archive bit

FSFIRST(p$,attr)
FSNEXT()

p$: sexp
attr: iexp

The function FSFIRST searches for the first file on a disc to fulfill the criteria specified in p$ (e.g. "C:*.GFA"). If such a file is found, the filename, together with other information, is written into the DTA (see FGETDTA and FSETDTA). The parameter attr contains the file attributes which the file to be found may have; for instance, it is possible to search for hidden files (bit 1) or folders (bit 4).

The function FSNEXT() searches for the next file which fulfills the conditions of FSFIRST.

Example:

```
~FSETDTA(BASEPAGE+128)              ! Set the DTA
e%=FSFIRST("\*.GFA",-1)             ! Set search criteria
'                                   ! (-1 =all bits set)
DO UNTIL e%
  IF BYTE{BASEPAGE+149} AND 16      ! If it is a folder
    PRINT "*";CHAR{BASEPAGE+158},   ! indicate by a star
  ELSE                              ! otherwise
    PRINT ' CHAR{BASEPAGE+158},     ! a space before
'                                   ! the file name
  ENDIF
  e%=FSNEXT()                       ! Continue search
LOOP
```

--> Displays all files with the extension .GFA from the current directory on the screen, as well as all folder names with that extension, indicated with the prefix '*'.

MKDIR name$
RMDIR name$

name$: sexp

The instruction MKDIR (make directory) puts a new directory (folder) on the disc. The expression name$ contains the associated access path. With RMDIR (remove directory) a directory is deleted, provided, however, that it is empty (i.e. contains no files or sub-directories).

Example:

```
MKDIR "A:\PROGS"
RMDIR "A:\PROGS"
```

--> The directory "PROGS" is created on drive A:, then deleted.

Files

EXIST(name$)

name$: sexp

By means of EXIST one can determine whether a given file exists on a disc. The parameter name$ contains the access path and the filename. The function returns the value TRUE (-1), if the file exists, or FALSE (0) if not.

Example:

```
OPEN "o",#1,"TEST.TXT"
PRINT #1,"EXAMPLE"
CLOSE #1
PRINT EXIST("TEST.TXT")
PRINT EXIST("TEST.DOC")
```

--> The file TEST.TXT is opened, some sample text is put in it, and the file is closed again. The value -1 (TRUE) is then printed, followed by 0 (FALSE), as the file TEST.DOC does not exist.

Also see: FSFIRST, FSNEXT.

OPEN mode$,#n,name$ [,len]

mode$,name$: **sexp**
len,n : **iexp**

OPEN opens a data channel to a file or to a peripheral device. The expression mode$ sets one of the following access modes:

O (output)	A file is opened to receive data. If necessary, the file will be created, or, if the file already exists, its contents will be deleted.
I (input)	A file is opened for reading.
A (append)	An existing file is opened, and the data pointer set to the end of the file. Data output to the file will then be added at that point.
U (update)	An existing file is opened for writing and reading.
R (Random access)	A random access file is opened for reading and writing. This file type is described fully further on.

The numerical expression n contains the channel number and can take a value from 0 to 99. This channel number must be specified when working with the file. The '#' sign before the channel number can be omitted, as the Editor will supply it anyway. The expression name$ contains the access path and the filename of the required file.

Instead of a filename, a peripheral device can be specified. The numerical expression len is used only with the Random Access mode and defines the length of a data record.

Contraction	Meaning	Internally
LST: (list)	Parallel	BIOS 0
AUX: (auxiliary)	Serial (RS232)	BIOS 1
CON: (console)	Keyboard/screen	BIOS 2 or VDI
MID: (musical instr. dig. interface)	MIDI	BIOS 3
IKB: (intelligent keyboard)	Keyboard processor	BIOS 4
VID: (video)	Monitor	BIOS 5 or VDI
PRN: (printer)	Printer	GEMDOS 3

LOF(#n)
LOC(#n)
EOF(#n)
CLOSE [#n]
TOUCH [#] n

n: iexp

The functions LOF (length of file), LOC (location) and EOF (end of file) can be applied only to files previously opened with OPEN. With all three functions the numerical expression n refers to the channel number previously specified with OPEN.

LOF returns the length of a file in bytes.

LOC returns the current position of the data pointer measured in bytes from the beginning of the file (where LOC returns zero). (See also SEEK).

EOF determines whether the data pointer points to the end of a file (or the whole file has been read). If the data pointer does points to the end of the file, TRUE (-1) is returned, otherwise FALSE (0).

CLOSE closes a data channel to a file or peripheral device previously opened with OPEN. The numerical expression n contains the number of the channel to be closed. If the channel number is omitted, all open files are closed.

TOUCH updates the date and time stamps of a file, giving it the current system time and date.

Examples:

```
OPEN "o",#1,"TEXT.TXT"
FOR i%=1 TO 20
  PRINT #1,STR$(i%)
NEXT i%
CLOSE #1
FILES "TEST.TXT"
DELAY 20                ! Wait 20 seconds
OPEN "u",#1,"TEST.TXT"
TOUCH #1
CLOSE #1
FILES "TEST.TXT"
```

--> In the example the file TEST.TXT is opened, written to, and closed Then the file information is displayed, including the time and date. Twenty seconds later the file is opened again, the date and time updated, then closed. The file information is displayed again.

```
OPEN "i",#1,"TEST.TXT"
PRINT "file length: ";LOF(#1)
PRINT
PRINT "data","position of the data pointer"
DO UNTIL EOF(#1)
  INPUT #1,a$
  PRINT "";a$,LOC(#1)
LOOP
CLOSE #1
```

--> The example opens the file from the previous example for reading. First the length of the file is displayed by means of LOF. Then the contents of the file are displayed along with the associated data pointer position until the termination condition of the loop is met when EOF(#1) is TRUE.

NAME old$ AS new$
RENAME old$ AS new$
KILL name$

old$,new$,name$: sexp

NAME gives the file specified in old$ the new name specified in new$. The contents of the file are not changed. Of course, old$ and new$ must refer to the same disc drive in their file specifications. RENAME and NAME are synonymous.

KILL deletes the file specified in the expression "name$".

Example:

```
OPEN "o",#1,"TEST.TXT"
PRINT #1,"example"
CLOSE #1
'
NAME "TEST.TXT" AS "EXAMPLE.TST"
DIR
KILL "EXAMPLE.TST"
DIR
```

--> The file TEST.TXT is opened and some sample data written to it. It is then renamed as EXAMPLE.TST and the directory displayed to show its precedence. EXAMPLE.TST is then deleted and the directory displayed again, this time to show its absence.

BLOAD name$ [,addr]
BSAVE name$,addr,count
BGET #n,addr,count
BPUT #n,addr,count

name$: **sexp**
n,addr,count: **aexp**

With BSAVE an area of memory can be stored on disk (or RAM-disk, hard disk drive etc.), and loaded again e.g. with BLOAD. The numerical expression addr specifies the address of the first byte of the area to be saved, or, with BLOAD, the address where the data from disk is to be put. If no address is specified with BLOAD, the address that was specified with BSAVE when the area was saved will be used. BSAVE must also specify the number of bytes to be saved. The parameter name$ is the file specification of the file to which the area is to be saved, or from which it is to be loaded. See the beginning of 'File Management' for details of the specification.

BSAVE and BLOAD always use a whole file. In contrast, BPUT and BGET access a file via its channel number n, so it is possible to use BGET and BPUT to save or load parts of a file.

Examples:

```
DEFFILL 1,2,4
PBOX 100,100,200,200
BSAVE "RECTANG.PIC",XBIOS(2),32000 !XBIOS(2) gives
'                                  !screen address
CLS
PRINT AT(4,20);"Picture stored. Press a key"
~INP(2)
CLS
BLOAD "RECTANG.PIC"
```

--> Draws a rectangle and stores the screen under the name "RECTANG.PIC". An appropriate message is displayed, and, after a key has been pressed, the file is re-loaded, again to the screen memory area.

```
DEFFILL 1,2,4
PBOX 0,0,639,199
DEFFILL 1,2,2
PBOX 0,200,639,399
DEFTEXT 1,0,0,32
TEXT 10,115,"the upper half"
TEXT 10,315,"the lower half"
'
OPEN "o",#1,"SCREEN.PIC"
BPUT #1,XBIOS(2),32000
CLOSE #1
PAUSE 25
CLS
'
OPEN "i",#1,"SCREEN.PIC"
BGET #1,XBIOS(2)+16000,16000
BGET #1,XBIOS(2),16000
CLOSE #1
```

--> The upper half of the screen is filled with one pattern, the lower half with another, and appropriate messages are put in each half. Then the entire screen is saved into the file SCREEN.PIC. After a pause, the first half of the file is loaded into the bottom half of the screen, and the second half of the file is loaded into the top half of the screen.

Sequential Access

INP(#n)
OUT #n,a [,b,c,...]
INP&(#), INPT%(#), OUT&, OUT%

n,a,b,c,... : iexp

INP(#n) reads a byte from a file which has been previously opened with OPEN. Similarly, OUT #n sends a byte to a file. The numerical expression n is the channel number under which the file was OPENed.

INP and OUT without the '#' can also be used for communication with the screen, keyboard, etc. (e.g. INP(2) takes a character from the keyboard). Only the low-order 8 bits of a,b,c,... are output, thus limiting their values to 255.

The INP-function and the OUT-command now cater for 16 and 32 bit input, e.g. A% = CVL(INPUT$(4, #1)) is replaced by A% = INP(#1).

Example:

```
OPEN "o",#1,"TEST.TXT"
OUT #1,128
CLOSE #1
'
OPEN "i",#1,"TEST.TXT"
a=INP(#1)
CLOSE #1
PRINT a
```

--> In the first part of the example a file is opened for output and a byte is written to it. In the second part this byte is read back in as the variable 'a' which, when printed, is revealed to have the value 128.

INPUT$(count [,#n])

n,count: iexp

INPUT$ reads 'count' characters from the keyboard and assigns them to a string. Optionally, if the channel number n (0 to 99) is specified, the characters are read in as bytes from a file or peripheral device. In both cases the numerical expression 'count' determines the number of characters read.

Example:

```
OPEN "o",#1,"VERSION.DAT"
  PRINT #1,"GFA BASIC, Version 3.0"
  CLOSE #1
  '
  OPEN "i",#1,"VERSION.DAT"
  v$=INPUT$(9,#1)
  CLOSE #1
  PRINT v$
  PRINT "Please type in the Version number :";
  PRINT INPUT$(3)
```

--> In the first part of the example the file "VERSION.DAT" is opened and a message printed to it. The second part reads the first 9 characters of this file into the variable v$ and displays v$ on the screen. Then a message appears, and 3 characters are taken from the keyboard and printed.

INPUT #n,var1 [,var2,var3,...]
LINE INPUT #n,a1$ [,a2$,a2$,...]

n : iexp
a1$,a2$,a3$: sexp
var1,var2,var3: avar or svar

INPUT #n makes it possible to take data from a file or a peripheral device. Individual variables or variable lists (where the variables are separated by commas) can be input. These instructions correspond to INPUT and LINE INPUT, though they only take input from the keyboard.

Example:

```
OPEN "o",#1,TEXT.DOC
WRITE #1,"Goodbye","Hello","Hello, Hello, Hello"
CLOSE #1
'
OPEN "i",#1,"TEXT.DOC"
INPUT #1,a$,b$
LINE INPUT #1,c$
CLOSE #1
PRINT a$
PRINT b$
PRINT c$
```

--> Three strings are written to the file TEXT.DOC. The first two are read back in with INPUT#, the third with LINE INPUT#, as it contains commas. The three strings are printed.

PRINT #n,expression
PRINT #n,USING form$,expression
WRITE#n,expression

n : **iexp**
form$: **sexp**
expression : **aexp or sexp or a combination**

PRINT #n outputs data to a specified channel. PRINT #n USING allows formatted output to a data channel. In both cases n (0 to 99) denotes the required channel. Otherwise the instructions operate like PRINT, PRINT USING and WRITE. PRINT #n AT, however, is not possible. The instruction WRITE # serves primarily for the space-saving storage of data in sequential files in a format suitable for later reading with INPUT #. The expressions are separated by commas and character strings must be enclosed in quotation marks.

Examples:

```
OPEN "o",#1,"TEXT.DOC"
PRINT #1,"test"
CLOSE #1
'
OPEN "i",#1,"TEXT.DOC"
INPUT #1,a$
CLOSE #1
PRINT a$
```

--> The string "test" is printed to the file TEXT.DOC, which is then closed and re-opened for input. INPUT # takes the string from the file as a$, which is then printed on the screen.

```
OPEN "o",#1,"TEST.DAT"
WRITE #1,"Version ",3,".0"
CLOSE #1
'
OPEN "i",#1,"TEST.DAT"
INPUT #1,v1$,v2$,v3$
CLOSE #1
PRINT v1$+v2$+v3$
```

--> Data separated by commas is put into the file TEST.DAT and afterwards read back in again with INPUT # and displayed on the screen.

STORE #i,x$() [,n [TO m]]
RECALL #i,x$(),n [TOm] ,x

i,n,m : iexp
x$(): String array
x : Variable, at least 32-bit

The instruction STORE is used for sending the contents of an array to a file or data channel (with the elements separated by CR/LFs). The optional parameter m specifies how many elements of the array are to be sent to the previously OPENed channel n; if m is omitted, the whole array is transferred.

The instruction RECALL allows the quick inputting of m lines from a text file to the array x$(). If m=-1 all available lines are read. If m is too large for the dimensioning of the array, then the number of lines read in is limited automatically. If the end of file is reached during reading then the inputting is broken off without an error occurring. In each case, after the reading has been completed, the variable x will contain the number of strings actually read.

In spite of mentioning the number of strings to be read/written you can give the range of the strings to be stored, e.g. STORE #1, a$(), 10 TO 20.

--> That stores eleven strings to a$(), starting with number 10 up to 20, counting ALWAYS starts with zero.

Examples:

```
DIM A$(1000)
FOR i%=0 TO 499
  a$(i%)=STR$(RND)              ! anything
NEXT i%
'
OPEN "o",#1,"TESTFILE.TXT"
STORE #1,a$(),500
```

```
CLOSE #1
'
DIM b$(2000)
OPEN "i",#1,"TESTFILE.TXT"
RECALL #1,b$(),-1,x
CLOSE #1
PRINT x
```

--> The number of text lines read is displayed (500).

```
PRINT "Line counter:"
DIM a$(1000)
DO
  FILESELECT "\*.*","",f$
  EXIT IF f$= ""
  lc%=0
  OPEN "i",#1,f$
  DO
    RECALL #1,a$(),-1,x%
    ADDS lc%,x%
  LOOP WHILE x%
  CLOSE #1
  PRINT f$'"contains"'lc%'"lines."
LOOP
```

--> This program counts the lines in text files.

Note: STORE functions with files and peripheral devices, but RECALL only functions with files, since a SEEK (see following) is used internally.

SEEK #n,pos
RELSEEK #n,num

n,num,pos: iexp

The commands SEEK and RELSEEK permit the re-positioning of the data pointer with an accessed file, allowing the realisation of indexed sequential file access. The numerical expression n contains the channel number used when the file was OPENed. Both commands can be used only with files, not with peripheral devices. The data pointer specifies which byte of a file was read or written last. Except for access mode 'A' (used to append data to the end of an an existing file) the data pointer has the value 0 when opening a file. Reading or writing commences with first byte, to which the data pointer subsequently points.

The SEEK command positions the data pointer to a specified byte number in a file. The pointer can, however be moved a specified number of bytes forwards or backwards with RELSEEK (relative seek): the pointer is moved the number of bytes specified in num. RELSEEK is generally faster than SEEK.

SEEK may use positive values of pos up to the relevant file length. RELSEEK accepts positive or negative values of num, an error occurring when an attempt is made to position the data pointer past the end or before the beginning of a file. SEEK #n,0 takes the pointer to the start of a file.

Example:

```
OPEN "o",#1,"X.X"
PRINT #1,STRING$(20,"*")
SEEK #1,10
PRINT #1,"#";
RELSEEK #1,-5
OUT #1,48,49
CLOSE #1
'
OPEN "i",#1,"X.X"
LINE INPUT #1,a$
PRINT a$
CLOSE #1
```

--> Displays: ******01**#**********

Random Access

In the following section the handling of a Random Access file is described. Two terms are particularly important: Data record and data field. A data field is a single item of information, for instance a name or telephone number. A data record consists of a number of data fields, and is analogous to one card in a card index. The length of a record (in bytes) must equal the sum of the lengths of the individual fields.

The difference between a random access file and a sequential file lies in the method of data access. With a sequential file the entire file must be loaded in order to be able to access a particular data record, whereas with a random access file the record may be obtained directly. This is particularly useful with very large files, although this advantage is offset by greater consumption of space on the disk, because a random access file uses a fixed record length, resulting in wastage if some records are shorter than others.

FIELD #n,num AS svar$ [,num AS svar$,num AS svar$, ..]
FIELD #n,num AT(x) [,num AT(x),num AT(x),...]

n,num,x : iexp
svar$: svar (not an array variable)

The command FIELD AS is used to divide data records into fields. The numerical expression n is the number of the data channel (0 to 99) of a file previously opened with OPEN. The integer expression num determines the field length. The variable svar$ contains data for one field of a data record. If the data record is to be divided into several fields, the parts 'num AS svar$' must be separated by commas. The sum of the individual field lengths must equal the data record length, otherwise an appropriate error message is displayed. Thus in order to keep the field lengths to those specified in the FIELD command it is convenient to use the commands LSET and RSET or MID$.

By using AT() instead of AS numeric variables can be written to a random access file. In this case num must contain the length of the variable (1 for byte-type, 2 for word-type etc.) and the brackets must contain a pointer to the variable (see *, VARPTR). Actually an arbitrary area of memory can be transferred by specifying the number of bytes (num) and the address of the first byte (x). Note that there is no space after the 'AT'.

For example:

```
FIELD #1,4 AT(*a%),2 AT(*b&),8 AT(*c#)
```

Arbitrary combinations of AS and AT are possible, e.g.:

```
FIELD #2,4 AS a$,2 AT(*b&),8 AT(*c#),6 AS d$
```

Unlike in Version 2, several successive FIELD commands can be used, on sucessive lines, referring to the same channel number. The effect is the same as that of one long FIELD command. The maximum record length is 32767 bytes, and the maximum number of fields approximately 5000.

GET #n [,r]
PUT #n [,r]
RECORD #n,r

n,r: iexp

GET reads a data record from a random access file. Similarly, PUT stores a data record in such a file. The parameter n (0 to 99) is the channel number under which the file was OPENed, and the optional parameter r contains a value between 1 and the number of data records in the file, specifying the number of the data record to be read or stored. If r is omitted, the next record will be read or stored.

RECORD sets the number of the record next to be read or stored with GET or PUT. (e.g. after RECORD #1,15 record number 15 will be read by GET #1.)

NB.: Only one record at a time can be added to a file. A FOR-NEXT loop etc. must be used for multiple record storage.

Examples:

```
OPEN "r",#1,"PERSONAL.INF",62
FIELD #1,24 AS name$,2 AT(*house&)
FIELD #1,24 AS road$,12 AS town$
FOR i%=1 TO 3
  INPUT "Name         : ";n$
  INPUT "House number: ";house&
  INPUT "Road         : ";r$
  INPUT "Town         : ";t$
  LSET name$=n$
  LSET road$=r$
  LSET town$=t$
  PUT #1,i%
  CLS
NEXT i%
CLOSE #1
```

--> First an random access file (mode 'r') with a record length of 62 bytes is opened. With FIELD a data record is specified as consisting of 4 fields of 24, 2, 24 and 12 bytes respectively (adding up to 62, the record length specified in the OPEN statement).

Then the program asks for three names and addresses. Each time the name, house number etc. are inserted left justified into the appropriate variables and the whole data record is written to the file.

(In older versions of GFA BASIC instead of '2 AT(*house&)' it was necessary to use '2 AS house$' and then 'LSET house$=MKI$(house&)' , etc.)

```
OPEN "r",#1,"PERSONAL.INF",62
FIELD #1,24 AS name$,2 AT(*house&)
FIELD #1,24 AS road$,12 AS town$
FOR i%=1 TO 3
  GET #1,i%
  PRINT "Record number: ";i%
  PRINT
  PRINT "Name         : ";name$
  PRINT "House number : ";house&
  PRINT "Road         : ";road$
  PRINT "Town         : ";town$
  PRINT
  PRINT
NEXT i%
CLOSE #1
```

--> Here the file PERSONAL.INF is opened in Random access mode and the three records are read in and printed out.

Communicating with Peripherals

Byte by Byte input and output

INP(n)
INP?(n)
OUT [#]n,a [,b...]
OUT?(n)

n,a,b: iexp

INP reads a byte from a peripheral device. The numerical expression n can accept values from 0 to 5 (see the following table) or contain a channel number (#n). The command OUT sends a byte to a peripheral device.

Unlike GFA BASIC Version 2, one can now send several bytes with one OUT statement.

INP? and OUT? determine the input or output status of a peripheral device. A non-zero value is returned if the device is ready to send/receive, and zero (logical FALSE) if it is not.

Device table

n	Contraction	Meaning
0	LST: (list)	Printer
1	AUX: (auxiliary)	Serial (RS232)
2	CON: (console)	Keyboard/Screen
3	MID: (MIDI)	MIDI Interface
4	IKB: (intelligent kbd.)	Keyboard processor
5	VID: (video)	Screen

Examples:

```
PRINT AT(4,4);"Press a key please"
~INP(2)
```

--> A message is displayed and the program then waits for a byte from device 2 (the keyboard).

```
OUT 2,27,69,10,10,10
PRINT "Hello"
```

--> This command clears the screen (by means of the VT-52 control sequence ESCape + E, corresponding to the ASCII codes 27 and 69), then issues three Line Feed characters (ASCII code 10). Thus, 'Hello' is printed on the fourth line.

Serial (RS232) and MIDI interfaces

INPAUX$
INPMID$

By means of INPAUX$ and INPMID$ data can be read in very quickly from the serial and MIDI interfaces.

Example:

```
DO
  PRINT INPAUX$;
LOOP UNTIL MOUSEK
```

--> Reads in all the data from the input buffer of the serial interface. This method of data input is very much faster than inputting byte by byte.

```
inp_aux$=""
WHILE INP?(1)
  inp_aux$=inp_aux$+CHR$(INP(1))
WEND
```

--> This alternative method is clearly slower.

Mouse and Joysticks

MOUSEX
MOUSEY
MOUSEK
MOUSE mx,my,mk

mx,my,mk: avar

MOUSEX, MOUSEY and MOUSEK supply the X and Y coordinates of the mouse pointer, and information on the state of the mouse buttons. MOUSE allows the gathering of that information with one statement, giving the current mouse coordinates to mx and my and the mouse button status to mk. MOUSEK (or mk) will give values between 0 and 3, with the following meaning:

mk	Button(s) pressed
0	None
1	Left
2	Right
3	Both left and right

Example:

```
REPEAT
  IF MOUSEK=1
    PLOT MOUSEX,MOUSEY
  ENDIF
UNTIL MOUSEK=2

REPEAT
  MOUSE x%,y%,k%
  IF k%=2
    PLOT x%,y%
  ENDIF
UNTIL k%=1
```

--> In the first REPEAT-UNTIL loop a point is plotted on the screen at the mouse pointer position if the left button is pressed. When the right button is pressed the second REPEAT-UNTIL loop is entered. In the second loop, plotting takes place provided the right button is held down. When the left button is pressed, the loop and the program both terminate.

SETMOUSE mx,my [,mk]

mx,my,mk: iexp

The command SETMOUSE permits the positioning of the mouse cursor under program control. The optional parameter mk can simulate the mouse buttons being pressed or released. This unfortunately is only valid for the VDI, not, (or seldom) with the AES.

Example:

```
FOR i%=0 TO 300
  HIDEM
  SETMOUSE i%,i%
  PLOT MOUSEX,MOUSEY
  SHOWM
  PAUSE 2
NEXT i%
```

--> Moves the mouse pointer diagonally down the screen, plotting points at its current position as it goes.

HIDEM
SHOWM

The commands HIDEM (HIDE Mouse) and SHOWM (SHOW Mouse) cause the mouse pointer to be made invisible or visible respectively. Use of the ALERT command or other AES routines causes the mouse pointer to be switched on, and, if not required, it must be subsequently re-hidden with HIDEM.

During output to the screen with PRINT or the VDI or LINE-A routines the mouse pointer is automatically switched off, but is afterwards returned to its previous state.

Example:

```
REPEAT
  IF MOUSEK=1
    SHOWM
  ENDIF
  IF MOUSEK=2
    HIDEM
  ENDIF
UNTIL MOUSEK=3
```

--> The mouse pointer is activated by pressing the left button, and de-activated by pressing the right button. Pressing both simultaneously ends the program, when the 'Program End' Alert box leaves the pointer visible.

STICK m
STICK(p)
STRIG(p)

m,p: iexp

The ATARI ST is provided with two interfaces (Ports) for the attachment of Mouse and Joysticks. Port 0 can supply mouse or joystick information, but Port 1 can only read joystick data.

STICK 0 causes Port 0 to supply mouse information, and STICK 1 causes it to supply joystick information. Port 1 always reads a joystick. Normally it is not necessary to use STICK, since mouse queries (MOUSE, MOUSEK, etc.) and joystick queries (STICK(), etc.) cause STICK 0 and STICK 1 to be executed automatically. However, before using AES functions (ALERT, etc.) the mouse should be activated if necessary with STICK 0.

The function STICK(p) returns the position of a joystick. For p=0 the joystick at Port 0 is read, and for p=1, the joystick at Port 1. The values returned correspond to the position of the stick as follows:

```
 5  1  9
  \ | /
4 -- 0 -- 8
  / | \
 6  2  10
```

The function STRIG(p) returns the state of the fire button of the joystick attached to Port p as a logical value: TRUE (-1) if it is pressed, or FALSE (0) if not.

Examples:

```
STICK 1  ! Activates joystick if attached to Port 0
REPEAT
  direction%=STICK(0)
  fire!=STRIG(0)
  SELECT direction%
  CASE 4
    PRINT "Left"
  CASE 8
    PRINT "Right"
  CASE 2
    PRINT "Down"
  CASE 1
    PRINT "Up"
  ENDSELECT
UNTIL fire!
WHILE STRIG(0)
WEND          ! Waits for fire button to be released
```

--> With movement of the joystick, appropriate messages are printed until a fire button is pressed. After waiting for it to be released, the program ends.

Note the difference between the REPEAT and WHILE constructs:

```
REPEAT
  ' program segment
UNTIL condition


WHILE condition
  ' program segment
WEND
```

With REPEAT, the program segment will certainly be executed at least once, whereas with WHILE the program segment might not be executed at all if the condition is false.

Printing

LPRINT expression
LPOS(x)
HARDCOPY

expression: **aexp or sexp, arbitrarily mixed**
x: **avar (dummy argument)**

LPRINT is identical to the PRINT command and its variants (LPRINT USING, etc.), except that output goes to the printer instead of the screen, and it is not possible to use a PRINT AT equivalent, as this would involve re-positioning the print head. Similarly to POS, LPOS returns the number of characters printed since the last Carriage Return.

HARDCOPY causes a copy of the screen to be output to a suitable printer, in the same way as by pressing the ALTernate-Help keys. There is a driver for non-Epson-compatible 9 pin printers which, unlike HARDCOPY does not use XBIOS(20). It is activated by SDPOKE &H4EE,0.

Some hard copy drivers don't react on routine XBIOS(20). F Ostrowski made last-minute changes to the HARDCOPY command. Now there is a SPOKE &H4EE,0 and a VSYNC that should help even the worst cases.

Examples:

```
LPRINT
LPRINT "test"
PRINT LPOS(x)
```

--> The word "test" comes out on the printer (if it is attached, switched on, and On-Line) and the current print head position appears on the screen.

```
FOR i%=20 TO 180 STEP 10
  CIRCLE 320,200,i%
NEXT i%
HARDCOPY
```

--> Concentric circles are drawn on the screen, then a copy of the screen is made on the printer.

Sound Generation

SOUND chan,vol,note,octave,del
SOUND chan,vol,#per,del
WAVE voice,env,form,per,del

chan,vol,note,octave,del,per,voice,env,form: **iexp**

SOUND and WAVE serve to control the three-channel tone generator of the ATARI ST, the three channels having nothing to do with the data channels used to communicate with peripheral devices and files. The parameters have the following meanings:

Chan Channel number (0 to 3)

Vol Volume (1 to 15)

Note Note (1 to 12) determines the note as follows:

Note:	1	2	3	4	5	6	7	8	9	10	11	12
Tone:	C	C#	D	D#	E	F	F#	G	G#	A	A#	B

Octave Octave (1 to 8)

Del Delay in 1/50ths second before the next GFA BASIC command is executed.

Per Period of the waveform multiplied by 125000. Thus for a given frequency, per=ROUND(125000/frequency). In the alternative form of the SOUND command, #per can be used to replace the note and octave parameter, e.g. SOUND 1,15,10,4,250 and SOUND 1,15, #284,250 both produce a tone of 440Hz.

Voice Channel combination: with WAVE, any channel or combination of channels may be activated simultaneously. The value of 'voice' is 256 multiplied by the period (0 to 31) of the noise generator, plus the sum of the following:

1 = Channel 1
2 = Channel 2
4 = Channel 3
8 = Noise (channel 1)
16 = Noise (channel 2)
32 = Noise (channel 3)

Env Specifies the channels for which the envelope shaper (see 'form') is to be active. Its value is the sum of the following:

1 = Channel 1
2 = Channel 2
4 = Channel 3

Form Specifies the envelope shape (0 to 15) thus:

0 to 3 = (as 9)

4 to 7 = (as 15)

8 = Falling saw tooth

9 = Falling linearly

10 = Triangle, beginning with fall

11 = Falling linearly, then to max

12 = Rising saw tooth

13 = Rising linearly and holding

14 = Triangle, beginning with rise

15 = Linear rising, then to zero

Tone generation is begun by the SOUND or WAVE command and ended by another SOUND or WAVE command (SOUND 1,0,0,0,0 produces silence). As the operating system uses the sound chip to produce a keyboard click, this also terminates an on-going sound output situation. The keyboard click can be disabled by:

```
SPOKE &H484,BCLR(PEEK(&H484),0)
```

...and enabled by:

```
SPOKE &H484,BSET(PEEK(&H484),0)
```

These two statements have the effect of setting bit 0 of memory location &H484 to zero or one respectively.

The period of the envelope is determined by the parameter 'del'.

With both SOUND and WAVE, the parameters are remembered, so that for subsequent use with similar parameters, it is only necessary to specify parameters up to the one that is to change: after WAVE 7,7,0,10000,100, it is only necessary to type WAVE 1 to change the first parameter, leaving the others the same.

Example:

```
SOUND 1,15,1,4,20
SOUND 2,15,4,4,20
SOUND 3,15,8,4,20
WAVE 7,7,0,65535,300
```

--> A tone is produced with each channel and modulated by means of WAVE.

Chapter 7
Program Structure

In this chapter the commands used for controlling the execution of a program are discussed, starting with 'decision commands', by means of which the execution of certain program sections is carried out only if certain criteria are met. In the case of IF, THEN, ELSE, ENDIF and ELSE-IF, the criterion is a logical one: BASIC checks if a logical expression is true or false. in addition to these are the multiple decision commands SELECT, CASE, DEFAULT, ENDSELECT and CONT. These not only check for true or false, but can accept arbitrary values which can be reacted to selectively.

The next section sets out the loop commands, which make the repeated execution of specified sections of a program possible. GFA BASIC 3 is provided with a very large number of such loop types: FOR-TO-STEP-NEXT, REPEAT-UNTIL, DO-LOOP or ENDLOOP, DO-WHILE, DO-UNTIL, LOOP-WHILE, LOOP-UNTIL, EXIT-IF.

For structured programming the use of sub-routines is of great importance. With the commands PROCEDURE, GOSUB (or @) and RETURN (or ENDSUB) new 'commands' can be created in the form of subroutines.

Functions may also be defined by the user. With the commands DEFFN and FN functions are created like formulas. More flexibility is possible with the commands FUNCTION, ENDFUNC and RESULT, which allow the formation of complete subroutines that return a value.

Also in this section is a detailed description of the the way in which variables are dealt with by subroutines. Local variables may be declared with LOCAL, with the effect that they are only valid in that subroutine. Variables of the same name in other parts of the program are unaffected. Variables can be passed to subroutines as values, or, with the VAR command, the variable itself can be passed (called by reference), and its value changed without the need to refer to it directly by name in the subroutine.

The section on conditional branches also explains the commands which deal with the following two special events: the simultaneous pressing of the 'break keys' Control-Shift-Alternate, which normally stop a program; and the error situation which occurs when, for example, an attempt is made to divide by zero, or to take the square root of a negative number etc. Both of these events can be made to trigger the execution of a user-defined routine, instead of stopping the program.

It is possible in GFA BASIC 3 to call a subroutine when a specified amount of time has elapsed, using the commands EVERY and AFTER. This is explained in this chapter.

Towards the end of the chapter the absolute branch GOTO, the commands PAUSE and DELAY, which temporarily suspend program execution, as well as different methods of ending the program (QUIT, SYSTEM, END, EDIT, NEW and STOP) are presented. The last section covers the commands for monitoring the execution of a program (TRON, TROFF, TRON proc, TRACE$, DUMP, ERR$, ERROR).

Decision Commands

IF condition [THEN]
ELSE
ENDIF

condition : bexp

These commands enable one to specify that a section of a program will only be executed if a logical condition is met. The following example demonstrates this:

```
IF a=1 THEN
  PRINT "a is equal to 1"
  b=2
ENDIF
```

In this case a=1 is the logical condition. The instructions in the lines between IF and ENDIF are processed **only** if this logical condition is MET. If it is untrue, then the program continues with the commands after the ENDIF and the command THEN is not considered. It should be noted that it is sufficient to use the form:

```
IF a=1    instead of    IF a=1 THEN
```

The following construction is somewhat more complex:

```
IF a=1
  PRINT "a is equal to 1"
ELSE
  PRINT "a is not equal to 1,"
  PRINT "it is equal to ";a
ENDIF
```

In this case shown above, the instructions between IF and ELSE are processed if the logical condition after the IF is true. The program then continues execution after the ENDIF. However, if the condition after IF is not fulfilled, then the instructions between ELSE and ENDIF become effective. Again, program execution is continued after the ENDIF.

Note: Any numerical expression, which is not equal to 0, i.e. is FALSE, is considered to be true. The logical value for TRUE is -1 but any non-zero value is considered to be equivalent.

Example:

```
x=1
IF x
  PRINT "x is true"
ENDIF
INPUT y
IF x=9 OR ODD (y)
  PRINT "y is an odd number"
ELSE
  PRINT "y is an even number"
ENDIF
```

--> First the message 'x is true' is displayed and then a numeric input is requested. Since x cannot be 9, the text appears 'y is an odd number', if you have entered an odd number, otherwise 'y is an even' is displayed.

ELSE IF condition

condition : bexp

The command ELSE IF enables nested IF's to be more clearly expressed in a program. The following examples show a simple menu selection made on a single key-press. If l, s or e is pressed then, respectively, a load, save or input routine is called. In all other cases the message 'unknown command' is printed. The normal nested version is as follows:

```
DO
  t$=CHR$(INP(2))
  '
  IF t$="l"
    PRINT "Load text"
  ELSE
    IF t$="s"
      PRINT "Save text"
    ELSE
      IF t$="e"
        PRINT "Enter text"
      ELSE
        PRINT "unknown command"
      ENDIF
    ENDIF
  ENDIF
  '
LOOP
```

The use of ELSE IF produces shorter listing with smaller program indents.

```
DO
  t$=CHR$(INP(2))
  '
  IF t$="l"
    PRINT of "Load text"
  ELSE IF t$="s"
    PRINT "Save text"
  ELSE IF t$="e"
    PRINT "Enter text"
  ELSE
    PRINT "unknown command"
  ENDIF
  '
LOOP
```

The program works in the following way:

If the condition after IF is fulfilled (t$="l"), then the instructions between IF and the next ELSE IF are processed - PRINT "Load text" - and then the program jumps to the command ENDIF. If the condition for the first IF is not true then the other IF's are encountered.

In the second case, if the condition after the ELSE-IF command is met, then all instructions up to the next ELSE, ELSE IF or ENDIF (if no ELSE exists) are processed and the program jumps to the instruction after ENDIF. If neither the condition after the IF or the condition after ELSE-IF is true, then the commands between ELSE and ENDIF are implemented (if an ELSE exists) .

Multiple Branching

ON x GOSUB proc1,proc2, ...

x: **iexp**
proc1, proc2 : **procedure name without parameter**

This command branches the progam to a procedure, which is in the list specified after GOSUB. The x is a numerical (normally integer) expression, whose decimal part (if any) is ignored. If x is smaller than 1 or larger than the number of the procedure names after the GOSUB, then no subroutine is called. After calling the subroutine the program execution is continued immediately after the ON x GOSUB. With this command no parameters can be handed to the procedure.

Example:

```
x=3
ON x GOSUB proc1,proc2,proc3
x=1
ON x+1 GOSUB proc1,proc2,proc3,proc4
```

--> First the procedure proc3 is called, then the procedure proc2.

SELECT x
CASE y [TO z] or CASE y [, z, ...]
CASE TO y
CASE y TO
DEFAULT
ENDSELECT
CONT

x, y, z : iexp or string-constant with a maximum length of 4 characters

The command SELECT makes branching possible using the CASE command, on the basis of the value of the numerical expression x. The following examples give an explanation of the program structure that results:

```
x=0
SELECT x+2
CASE 1
  PRINT "x is equal to 1"
CASE 2 TO 4
  PRINT "x is equal to 2, 3 or 4"
CASE 5,6
  PRINT "x is equal to 5 or 6"
DEFAULT
  PRINT "x is not equal to 1, 2, 3, 4, 5 or 6"
ENDSELECT
```

--> The message 'x is equal to 2, 3 or 4' is displayed.

First the numerical expression after SELECT is evaluated, which is the branch condition, in this case 2. Then the various CASE instructions are gone through and checked as to whether the current value of the branch condition is to be found.

In this example the argument '1' follows the first CASE command. Since the branch condition is 2, the program jumps to the next CASE.

After the second CASE are the arguments '2 TO 4'. As the value of x falls within this range the condition is fulfilled and the message printed on the screen. After that program execution continues after the command ENDSELECT. After the third CASE command is a further variant of the possible arguments for the SELECT command. There the desired values are separated, in a list, by commas.

If none of the current branch criterion after the various CASE commands is met, then the instructions between DEFAULT and ENDSELECT are executed (assuming a DEFAULT is present). In addition, instead of DEFAULT, OTHERWISE can be used (giving compatibility with other BASICs) but the GFA Interpreter replaces it with DEFAULT automatically.

Also, after CASE commands, not only numeric values but also strings may be used as the argument. These may have a maximum length of four characters. If only one character is specified then its ASCII value is used as the branch criterion. If two characters are specified, then this value is calculated as follows:

ASCII value of first character + 255 * ASCII value the second character.

... and so on for the third and fourth characters.

NOTE: The maximum length of CASE is 4 characters, i.e. CASE"A,B,C,D".

Example:

```
exit!=FALSE
REPEAT
  key%=inp(2)
  SELECT key%
  CASE "a" TO "z"
    PRINT "the lower-case letter"+chr$(key%)+" was
entered"
  CASE "A" TO "Z"
    PRINT "the upper-case letter"+chr$(key%)+" was
entered"
  CASE 27
     exit!=TRUE    !program ends when <Esc> key is
pressed
  DEFAULT
    PRINT "an inadmissible key was pressed!"
  ENDSELECT
UNTIL exit!
```

--> Within a loop the keyboard is queried and the the program jumps according to which key was pressed and the appropriate message is displayed. Pressing the Esc key is used as an abort criterion for the loop.

The next example illustrates the meaning of the CONT command, which is effective if it comes before a CASE or DEFAULT command. This CONT command must not be confused with the command of the same name used for resuming execution of an interrupted program run.

Example:

```
x=1
SELECT x
CASE 1
  PRINT "x is equal to 1"
  CONT
CASE 2
  PRINT "x is equal to 2"
CASE 1,3
  PRINT "x is equal to 3"
DEFAULT
  PRINT "x is not equal to 1, 2 or 3"
ENDSELECT
```

--> Displays 'x is equal to 1' and then 'x is equal to 2'.

The CONT command provides a method of jumping over a CASE or DEFAULT command. In this example the value x is equal to the argument after the first CASE, i.e. x=1. Therefore the message "x is equal to 1" is printed. Then comes the CONT command, which jumps over the line CASE 2 and, although the branch condition is not fulfilled, the CONT command causes the instruction PRINT "x being equal to 2" to be implemented. The normal processing continues with processing jumping to the ENDSELECT command.

Example:

```
SELECT INP(2)
CASE "a" TO "g"
  PRINT "a to g"
DEFAULT
  PRINT "default"
ENDSELECT
```

--> If you press one of the a, b, c, d, e, f or g keys the text 'a to g' is displayed, with any other key (except the shift keys) the message 'default' is printed.

With the CASE command a number of different possibilities can be combined, it is not necessary for them to be written separately:

```
SELECT a$
CASE "a" TO "z"
  CONT
CASE "A" TO "Z"
  CONT
CASE "ae","oe","ue","ss","Ae","Oe","Ue"
  PRINT "OK"
ENDSELECT
```

and similarly with ranges:

```
SELECT a$
CASE "a" TO "z" ,"a" TO "z","ae","oe","ue",...
...
...
```

Loops

FOR, TO, STEP, NEXT
REPEAT, UNTIL
WHILE, WEND
DO, LOOP, DO WHILE, DO UNTIL, LOOP WHILE, LOOP UNTIL
EXIT IF

GFA BASIC 3 is provided with an unusually large selection of loop types. Normally one differentiates between 'entry testing' and 'exit testing' loops. 'Entry testing' loops are those, such as WHILE-WEND, in which the abort condition for the loop is checked before entry into the loop. While with 'exit testing loops', such as FOR-NEXT, this condition is examined at the end of the loop, consequently such are gone through at least once. A special loop type, DO-LOOP, is also available in GFA BASIC. This type acts as a continuous loop without an abort condition. In GFA BASIC this command can be very flexibly used. Both after DO and after LOOP the extensions WHILE and UNTIL can be used so that the loops begin or end with a logical condition. In all of the loop types mentioned here, as many as abort conditions as required may be used in the loop body using the EXIT-IF command.

FOR c=b TO e [STEP s]
(Instructions)
NEXT i
DOWNTO

c : avar
b,e,s : aexp

The FOR-NEXT command (also called a counting loop) is used for repeated processing a group of instructions between the commands FOR and NEXT. For this purpose the count variable c is incremented (or decremented), beginning with the initial value b, as far as the end value 'e'.

The instructions in the loop body are gone through, until NEXT is reached. There the count variable c is increased by the value s given after STEP. If no STEP value is given, then an increment of one is assumed. Next c is checked to see whether it has exceeded the value e. If this is the case, the loop is exited and the program continued with the command after the NEXT If not, the program loops back again to the first instruction, this process is repeated, until c is larger than e. A consequence of this exit method is that c, after the loop is left, is equal to to e+s, i.e. the first value which exceeds the abort criterion. Also, the contents of a FOR-NEXT loop will always be gone through at least once.

In place of a STEP value of -1 the command DOWNTO can be used instead of TO. However, with DOWNTO the use of STEP is not possible.

In general, for the count variable c one should use integer variables, because thereby the loop can be processed more quickly than with floating-point variables. This is naturally not possible with non-integer STEP increments. In place of the command NEXT followed by the count variable, the command ENDFOR i% can be used, but the Interpreter automatically replaces it with NEXT i%.

Examples:

```
FOR c=1 TO 10
  PRINT c'
NEXT c
FOR c=-1 DOWNTO -10
  PRINT c'
NEXT c
```

--> Displays on the screen, the numbers:

```
1 2 3 4 5 6 7 8 9 10 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10
```

```
a$="T*e*s*t*w*o*r*d"
FOR j=1 TO LEN(a$) STEP 2
  PRINT MID$(a$,j,1);
NEXT j
```

--> This displays the word 'Testword' on the screen.

REPEAT
(instructions)
UNTIL condition

condition : bexp

The REPEAT-UNTIL command provides an 'exit tested' loop in which a number of instructions are repeated until a logical condition is true.

When the command REPEAT is reached in a program, the group of instructions up to the UNTIL are processed. Then it checks whether the logical condition after UNTIL is true (-1). If this is the case, then the instructions after the UNTIL are implemented. However, if the condition is false (0), then the program execution jumps to the REPEAT. The instructions between REPEAT and UNTIL are processed at least once, as long as the loop is not left by an EXIT IF or GOTO command. ENDREPEAT can be used in place of UNTIL, which the Interpreter automatically replaces by UNTIL.

Examples:

```
REPEAT
UNTIL MOUSEK
```

--> Waits for a mouse key to be depressed.

```
i=1
REPEAT
  INC i
  j=SQR(i)
UNTIL i>10 AND FRAC(j)=0
PRINT i
```

--> Displays the number '16' on the screen.

WHILE condition
(instructions)
WEND

condition : bexp

The commands WHILE and WEND can include a group of commands, which are processed as long as the logical condition is met. If GFA BASIC meets a WHILE command, then the logical condition following it is checked. If it is true, then the instructions between WHILE and WEND implemented. When the WEND is reached the program jumps to the WHILE and the cycle begins again, until condition is false. ENDWHILE can be written instead of WEND, which the Interpreter automatically replaces with WEND.

Example:

```
WHILE INKEY$=""
  PLOT MOUSEX,MOUSEY
WEND
```

--> Enables drawing with the mouse, until a key is pressed. If a character is already in the keyboard buffer, then no point is set.

DO
(instructions)
LOOP

The commands DO LOOP produce a continuous loop. The program processes the instructions between DO and LOOP and returns upon meeting LOOP to the command DO. The loop can only be left by means of EXIT IF, GOTO or other such commands to abandon its execution. In place of LOOP, ENDDO can be written, which the Interpreter replaces with LOOP.

Example:

```
DEFFILL 1,2,4
DO
  MOUSE mx,my,mk
  IF mk
    PBOX mx,my,mx+25,my+25
  ENDIF
LOOP
```

--> Draws filled rectangles at the current mouse pointer position when a mouse key is pressed.

DO WHILE condition
DO UNTIL condition
LOOP WHILE condition
LOOP UNTIL condition

condition : bexp

The commands DO and LOOP can be extended using UNTIL and WHILE. The loop command DO WHILE causes the instructions in the loop to be executed only as long as condition is true. If the loop begins with DO UNTIL, then it is entered only if the condition is not true. LOOP WHILE causes the program to jump back to the DO command as long as condition is true. LOOP UNTIL requires that the condition must be false for the program to loop back. Below, the conditions at DO are testing for true and at LOOP are testing for false.

```
DO WHILE condition                    WHILE condition
.                    corresponds to   .
.                                     .
LOOP                                  WEND

DO                                    REPEAT
.                                     .
.                    corresponds to   .
.                                     .
LOOP UNTIL condition                  UNTIL condition
```

The command variants DO, DO WHILE and DO UNTIL can be combined at will with LOOP, LOOP WHILE and LOOP UNTIL, so forming altogether nine type of loop.

Examples:

```
DO
LOOP UNTIL MOUSEK
```

--> Waits for a mouse button to be pressed.

```
DO UNTIL MOUSEK=2
  DO WHILE MOUSEK=1
    LINE 0,0,MOUSEX,MOUSEY
  LOOP
LOOP UNTIL INKEY$="a"
```

--> Draws lines when left mouse button is held down. If the right mouse button is pressed or the 'a' key is struck the program ends.

```
DO UNTIL EOF(#1)
  INPUT #1,a$
LOOP
```

--> Reads character strings from channel 1 sequentially, until the file end is reached.

```
WHILE NOT EOF(#1)
  INPUT #1,a$
WEND
```

--> Using WHILE WEND is slower, since, additionally, NOT is required.

EXIT IF condition

condition : bexp

By means of EXIT IF loops can be jumped out of, if the Boolian (logical) condition is fulfilled. The actual loop type used is arbitrarily selectable. EXIT IF can be used within IF-ENDIF and SELECT-ENDSELECT.

Example:

```
DO
  EXIT IF MOUSEK
LOOP
REPEAT
  EXIT IF INKEY$="x"
UNTIL FALSE
```

--> The program terminates, if first a mouse button and then the 'x' key are pressed.

Procedures and Functions

In GFA BASIC 3 subroutines, as in most modern programming languages, are given names. These subroutines can have parameters handed over to them and can then work on these parameters. There are two forms of parameter available, one where a value is passed to the sub-routine and the second where the variable itself is passed. In the second case the variable can be changed by the procedure without the necessity of referring to the variable by name. These two forms are known as "call by value" and "call by reference".

Likewise the use of local variables is possible, i.e. no consideration has to be taken on possible name clashes when calling procedures or functions. By means of DEFFN single-line functions to be defined and addressed using FN function name. Multi-line functions are also possible. These are only one special form of procedure and return a result (over RETURN).

GOSUB proc [(par1,par2,...)]
PROCEDURE proc [(var1,var2,...)]
RETURN

proc: **procedure name**
par1,par2: **sexp,aexp**
var1,var2: **svar,avar**

Between the commands PROCEDURE and RETURN are the instructions of a subroutine. After PROCEDURE the name of the subroutine and possibly the list of the variables to be received are placed. Calling a PROCEDURE takes place by giving its name at the beginning of the line, along with a list of appropriate parameters, which are placed in brackets. For the sake of clarity, the option of placing @ or GOSUB in front of the procedure name is available. This also avoids the possibility of confusing procedure names with GFA BASIC commands, e.g. @REM, @STOP. Parameters can be constants, variables and expressions. Not only the values, but also the variables can be passed (see VAR).

The command RETURN is used to end a procedure. When it is reached during the program execution, the program resumes execution from the instruction after the GOSUB. In place of the command PROCEDURE one can write SUB and instead of RETURN, ENDPROC or ENDSUB can be used which the Interpreter replaces itself.

Example:

```
GOSUB slow_print("** Manual for **")
@slow_print("* GFA BASIC 3 *")
slow_print("GFA SYSTEMTECHNIK")
'
PROCEDURE slow_print(t$)
  LOCAL i%
  FOR i%=1 TO LEN(t$)
    PRINT MID$(t$,i%,1);
    PAUSE 3
  NEXT i%
  PRINT
RETURN
```

--> Slowly prints the message character by character.

```
a=8
@cube_root(a)
PRINT a
'
PROCEDURE cube_root(VAR x)
  x=x^(1/3)
RETURN
```

--> Computes the cube root of 8 displaying 2 on the screen.

VAR-parameters

VAR-parametres allow variables themselves to be passed and not just their contents. They make it possible, not only to pass values into a procedure or function but also, to pass variables which can then be changed within the procedure or function. If the command VAR is found in the parameter list of a procedure function, all of the variables that follow will be treated as VAR-parameters.

With VAR parameters, unlike with global variables there is no danger of unintended side-effects. Using VAR-parameters complete arrays can be passed. VAR-parameters are not allowed within the calling line to a procedure or funtion, only within its definition.

VAR-parameters must always appear as the last elements of a parameter list.

The use of VAR-parameters is generally quicker than the use of ordinary parameters or pointers.

Examples:

```
sum(13,12,a)
sum(7,9,b)
PRINT a'b
PROCEDURE sum(x,y,VAR z)
  z=x+y
RETURN
```

--> Two pairs of numbers are added, the values being passed to the variables a and b. Then the numbers 25 and 16 are printed on the screen.

```
DIM a(9)
FOR i%=0 TO 9
  a(i%)=RND
NEXT i%
mean(0,9,a(),m)
PRINT "mean = ";m
PROCEDURE mean(from%,to%,VAR array(),mean)
  mean=0
  FOR i%=from% TO to%
    ADD mean,array(i%)
  NEXT i%
  DIV mean,to%-from%+1
RETURN
```

--> The arithmetic array filled with random numbers is calculated and printed out.

NOTE: In Version 2 it was possible to achieve the effect of passing local variables by passing their pointers. Using this method in Version 3 may result in the error message 'Pointer (*x) error'. It is therefore strongly recommended that VAR-parameters are used to pass local variables.

LOCAL var1 [,var2,var3,...]

var1,var2,var3 : avar, svar

It is possible using the command LOCAL to limit the area of application of variables. The variables specified after LOCAL are only valid in the procedure in which the LOCAL command is used. Also any variables in all subroutines called by this procedure will be local. Thus it is possible for variables specified after LOCAL to have the same name as variables in the main program, i.e. global variables. These global variables cannot be addressed then in the subroutine but are available after leaving the procedure and remain unchanged.

The variables given to a procedure or function are always local.

Example:

```
x=2
GOSUB test
PRINT x,y
'
PROCEDURE test
  LOCAL x,y
  x=3
  y=4
RETURN
```

--> The numbers 2 and 0 are displayed on the screen.

@func [(par1,par2,...)]
FUNCTION func [(var1,var2,...)]
RETURN exp
ENDFUNC

func : **function name**
par1,par2 : **sexp,aexp**
var1,var2 : **svar, avar**
exp : **sexp, aexp**

The commands FUNCTION and ENDFUNC form a subroutine, in a similar manner to PROCEDURE. The name of the subroutine and, optionally, the list of variables are given after FUNCTION command. Calling the subroutine takes place by the use of @ or FN and the function name, followed by a parameter list if necessary. The parameters can be constants, variables or expressions. Not only the values, but also the variables can be handed over by the parameter (see VAR).

If the command RETURN is reached during program execution, then the value given after it or the value of the variable named is returned. However, in a function RETURN can be used several times, with IF or the like. A function cannot be terminated without a RETURN command by ENDFUNC. In a function name ending with the $ character the functions will produce a string result.

Example:

```
fl%=@fac_loop(15)
fr%=@fac_recurs(10)
'
PRINT "loop: fac(15) = ";fl%
PRINT "recursion: fac(10) = ";fr%
'
FUNCTION fac_loop(f%)
  w=1
  FOR j%=1 TO f%
    MUL w,j%
  NEXT j%
  RETURN w
ENDFUNC
'
FUNCTION fac_recurs(f%)
  IF f%<2
    RETURN 1
  ELSE
    RETURN f%*@fac_recurs(PRED(f%))
  ENDIF
ENDFUNC
```

--> The factorials of 15 and 10 are, respectively, computed within a loop and recursively.

DEFFN func [(x1,x2,...)] = expression
FN func [(y1,y2,...)]

func,x1,x2 : var
expression,y1,y2 : exp

The command DEFFN allows the definition of single-line functions. These functions can appear at any point in the program.

The term 'expression' can any numeric or string expression which can include any of the parameters x1, x2, etc listed in the definition. These parameters are local variables to the function, and if they are also globally defined, no reference can be made to them in the expression as the local variables will be used instead. When the function is called these variables will contain the values listed within the brackets of the function call.

The function can be called using either FN func or @func .

Functions can be nested at will, to any depth. However, recursion is not possible and if attempted, the break-key combination will not work.

Examples:

```
DEFFN test(y,a$)=x-y+LEN(a$)
x=2
PRINT @test(4,"abcdef")
```

--> The number 4, i.e. 2-4+6 is displayed on the screen.

```
DEFFN first_last$(a$)=LEFT$(a$)+RIGHT$(a$)
b$=@first_last$("TEST")
PRINT b$
```

--> The text "TT" is displayed on the screen.

```
DEFFN fourth_power(x)=x^4
DEFFN fourth_root(x)=x^(1/4)
PRINT @fourth_root(@fourth_power(1024))
```

--> The number 1024 is displayed on the screen.

Error Handling

In this section two different types of event are considered. Non GFA BASIC specific events such as checking for mouse clicks, selecting pull-down-menus, etc, are dealt with in other chapters (Menu and Window Programming and AES Libraries).

The first of the two events to be discussed is the simultaneous pressing of the Control, Shifts and Alternate keys. The second type of event is the occurrence of an error during program execution.

ON BREAK
ON BREAK CONT
ON BREAK GOSUB proc

proc : name of a procedure

These three commands control the response to the simultaneous pressing of the Control, Shift (left shift-key only) and Alternate keys. Normally, pressing this key combination causes the termination of a program but, it can also be used to call a particular procedure. To do this, the procedure to be called is defined by means of the instruction ON BREAK GOSUB proc.

When an ON BREAK CONT is present in a program, the Control, Shift and Alternate key combination is deactivated. An ON BREAK reactivates this key combination once again.

Example:

```
ON BREAK GOSUB test
PRINT "Press CONTROL, SHIFT (the left one) and
                                        ALTERNATE"
DO
LOOP
'
PROCEDURE test
  PRINT "that was it"
  ON BREAK
RETURN
```

--> The request to press the key combination is displayed. When they are pressed, the procedure 'test' switches the normal BREAK routine on again.

ON ERROR
ON ERROR GOSUB proc
RESUME [NEXT]
RESUME [label]

proc : name of a procedure
label : name of a label

The occurrence of a fault (a TOS or GFA BASIC specific fault) normally causes the output of an error message and an abnormal termination. With ON ERROR GOSUB there is the possibility of branching to a specified procedure, following the detection of an error. In the procedure one can then determine the appropriate reaction to the error.

With the command ON ERROR one switches back to the normal error-trapping, so as to display the appropriate error message and bring about an abnormal termination. Under these conditions, therefore, when an error arises, an ON ERROR command is implemented automatically. In order to be able to react to a number of errors which occur one after the other, the ON ERROR GOSUB proc command must be contained within the error-trapping routine.

The RESUME command allows one to react in different ways when errors have arisen and it is only used in an error-trapping procedure. When used on its own as 'RESUME', this command causes the error to be repeated. RESUME NEXT continues execution with the command which follows the command causing the error. RESUME label causes the program to continue from the point 'label'. The label for RESUME may be locate either in a procedure or in the main program. If a fatal error occurred (see FATAL),then only RESUME label can be used and not RESUME NEXT or RESUME without a label.

Example

```
ON ERROR GOSUB error-trapping
ERROR 5
PRINT "and again..."
ERROR 5
PRINT "is not reached"
'
PROCEDURE error-trapping
  PRINT "Ok, error intercepted"
  RESUME NEXT
RETURN
```

--> The texts 'Ok, error intercepted' is displayed and then 'and again...'. Then the error message 'Square root only for positive numbers', is brought about by ERROR 5. The label for RESUME may be located either in a procedure or in the main program.

ERROR x
ERR
ERR$ (x)
FATAL

x : aexp

With ERROR x, the occurrence of error number x can be simulated. (For the table of error messages, see appendix.) This is command is particularly useful for example, when testing an error processing routine.

The number of the error that has arisen is returned in the variable ERR and, by means of this, one can determine the appropriate reaction to the occurrence of a specific error.

The function ERR$(x) returns, as a string, the GFA BASIC error message with the number x.

The variable FATAL is true if an error in the program generates an unknown address. This can happen, for example, when the error arose from the processing of an operating system routine. When this happens, a RESUME or RESUME NEXT can no longer be correctly executed.

Examples:

```
ON ERROR GOSUB error-trapping
INPUTS "Which error do you want: ",e
ERROR e
'
PROCEDURE error-trapping
  PRINT "That was error no.: ";ERR
  IF FATAL
    PRINT "It was a fatal error"
  ENDIF
RETURN
```

--> The user is asked to select an error by means of its error number and it is displayed on the screen. The program then asks for the next error number.

```
~ FORM_ALERT(1,ERR$(100))
```

--> The error message with the identification 100 is displayed, i.e. the copyright message, as an alert box.

Interrupt Programming

EVERY ticks GOSUB proc
EVERY STOP
EVERY CONT
AFTER ticks GOSUB proc
AFTER STOP
AFTER CONT

ticks : iexp
proc : name of a procedure

By means of the commands EVERY and AFTER, procedures can be called after the expiration of a certain amount of time 'ticks'. The command EVERY causes the procedure 'proc' to be called every 'ticks' clock units. AFTER causes this procedure to be called once only on the expiration of 'ticks' clock units.

The clock unit (tick) is defined as one two-hundredth of a second so ticks=200 sets an elapsed time of one second. However, a branch to the specified procedure can only be called on every fourth clock unit, resulting in an effective time resolution of one fiftieth of a second.

By means of EVERY STOP, the calling of a procedure can be prevented on expiration of the time period. With EVERY CONT, the calling of the procedure is again is allowed. The commands AFTER STOP and AFTER CONT work similarly, being implemented internally by means of the etv_timer vector ($400).

It is only after the complete processing of a command that a check is made to ascertain whether such a procedure is to be implemented. Thus commands which are executed slowly, such as INP (2), QSORT, file operations or the like, can obstruct these routines.

Example:

```
EVERY 4 GOSUB lines
lines! =TRUE
GRAPHMODE 3
DEFFILL 1.0
PLOT MOUSEX,MOUSEY
REPEAT
  IF MOUSEK=1
    EVERY STOP
  ELSE
    EVERY CONT
  ENDIF
  DRAW TO MOUSEX,MOUSEY
UNTIL MOUSEK=2
'
PROCEDURE lines
  INC y%
  LINE 320,y%,639,y%
  IF y%=399
    y%=0
  ENDIF
RETURN
```

--> Draws lines in the right half of the screen from top to bottom and, at the same time, permits the user to draw using the mouse. Pressing the left mouse button switches the moving lines in or out. The program can be terminated by means of the right mouse button.

```
PRINT "Text follows in 3 seconds, "
PRINT "if you do not press a key"
AFTER 600 GOSUB text
REPEAT
UNTIL INKEY$<> "" OR exit!
AFTER STOP
'
PROCEDURE text
  PRINT
  PRINT "here is the text"
  exit! =TRUE
RETURN
```

--> If no key is pressed in the three seconds after the program starts, then the message appears. If a key is pressed, then this terminates the program.

Other Commands

REM, GOTO, PAUSE, DELAY
END, EDIT
STOP
SYSTEM, QUIT

REM x
'x
<commands> !x

x : arbitrary text

A line which begins with a REM or ' command, can contain any text which is placed after these. The text, known as a REMark or comment, is not part of the program proper or subject to the syntax control of the editor and during program execution are not considered. They are usually used to make clearer how the program works. In addition, comments can be added to the end of a command line, other than one with DATA or INLINE commands. This is done by ending the executable portion of the program by means of an exclamation mark '!'.

Example:

```
REM comment
' PRINT "comment"
PRINT "REM" ! comment
```

--> The word 'REM' appears on the screen. Everthing else is ignored.

GOTO label

label : a programmer-defined label

By means of a 'label', specific locations in the program can be defined and a program started from this point with the instruction 'GOTO label'. Program execution is commenced from the label position. The label can consist of letters, numbers, underlines and full-stops. However, unlike with variable names, it may also begin with a number. It must, however, end with a colon. The colon should be left off when refering to a label in a GOTO command.

The use of GOTO is not allowed for jumping out of procedures, functions or FOR-NEXT loops. The use of this command also leads to unclear program structures which are not easy to follow and, it is generally considered, GOTO's should be avoided wherever possible.

Example:

```
PRINT "place 1"
GOTO jump_point
PRINT "place 2"
jump_point:
PRINT "place 3"
```

--> The texts "place 1" and "place 3" appear on the screen.

PAUSE x
DELAY x

x : aexp

The command PAUSE suspends program execution for x/50 seconds. DELAY has a similar effect but the argument x is specified in seconds with a theoretical resolution in milliseconds. DELAY uses the GEM routine EVNT_TIMER and is, therefore, recommended for use in GEM programs

Example:

```
PRINT "start"
BREAK 100
PRINT "a pause"
DELAY 2
PRINT "end"
```

--> The text 'start' appears then, two seconds later the further message 'a pause' is displayed. This is followed after a further two seconds by 'end'.

END
EDIT
STOP

These commands terminate the execution of a program. The command END terminates the program's execution and displays a box with the text 'Program end' on the screen. On clicking on the 'RETURN' button, GFA BASIC 3 returns to the Editor.

EDIT terminates program execution and returns control to the Editor immediately.

STOP causes an Alert box to appear with the choice of STOP and CONT. When the choice CONT is made, program execution continues. When the choice is STOP, GFA BASIC goes into Direct mode. At this stage, one can test and change the values of variables and, by means of CONT, continue program execution.

Example:

```
x=3
STOP
PRINT x
```

--> Select the button 'STOP' when the appropriate Alert-box appears. Now enter, in direct mode, the following commands:

```
PRINT x
```

--> The number 3 appears, next enter:

```
x=4
CONT
```

--> The last command of the program (PRINT x) is now processed and the number 4 appears, demonstrating that the value specified in Direct mode has been assigned to the variable x.

NEW

This command deletes the program currently in memory. In Direct mode, for safety's sake, the wishes of the user are queried. When in the Editor this command can be executed by means of Shift-F4 or a mouse click and, again a safety query is made.

LOAD f$

f$: sexp

This command LOAD loads a GFA BASIC program. The expression f$ must contain the file name along with the full access path for this file. When no extension is specified for the file, the default extension '.GFA' is used.

Example:

```
LOAD "a:\TEST.GFA"
```

--> The program loads TEST.GFA from the root directory of drive A.

SAVE f$
PSAVE f$

f$: sexp

The command SAVE stores a program on disk, under the specified name, f$. By means of the command PSAVE, the specified file can be saved with listing protection and cannot subsequently be listed after re-loading with LOAD as it is run immediately. In both cases, when no extension is specified for the file, the default extension '.GFA' is used.

Example:

```
SAVE "a:\TEST.GFA"
```

--> Saves the current program under the file-name TEST.GFA on drive A.

LIST [f$]
LLIST [f$]

f$: sexp

The command LIST displays the current program on the screen. Optionally, an access path can be specified, by which the program can be stored in ASCII format. Program sections that are to be inserted into other programs by means of 'MERGE', must be saved with LIST or SAVE,A from the Editor-Menu, in ASCII format.

When no extension is specified for the saving of a file, '.LST' is used as the default extension.

By means of the command LLIST, the current program can be output to the printer. The printer listing can be interrupted only by the switching off the printer. Once switched off, the printer will probably continue to print for some seconds before the program resumes or control is returned to the Editor. (See LLIST and the point commands in the section covering the Editor.) In addition, it is also possible to send text to a printer (see OPEN).

Example:

```
LIST "A:\TEST.LST"
```

--> The current program is stored under the name " TEST.LST" in ASCII format on drive A.

```
.11 70
.pl 66
LLIST
```

--> The current program is output to the printer with a line length of 70 characters and a page length of 66 lines.

CHAIN f$

f$: sexp

The command CHAIN loads a GFA BASIC program into memory and starts execution immediately the program is loaded. When no extension is specified, the extension '.GFA' is assumed.

Example:

```
CHAIN "A:\EXAMPLE.GFA"
```

--> The program EXAMPLE.GFA is loaded from disk and is immediately RUN.

RUN [f$]

f$: sexp

The command RUN starts the current program. Additionally, if a complete file name with access path is specified, then the appropriate program is loaded and started, replacing any program currently in memory.

Example:

```
RUN "A:\PART2.GFA"
```

--> The program PART2.GFA is loaded from drive A and RUN.

SYSTEM [n]
QUIT [n]

n : iexp

The commands SYSTEM and QUIT are equivalent in their effect, as they both terminate program execution and leave GFA BASIC. Unlike in Version 2 , SYSTEM and QUIT return a 2 byte integer value to the calling routine (normally the Desktop). This integer has a value of zero if the Interpreter was terminated correctly and left the calling program to hand over to the Desktop.

The convention applied in this case is that zero signals an error free run. A positive 16 bit number signals the occurrence of an internal error or a warning and a negative 16 bit number mostly points to the appropriate operating system error message. However this is not adhered to by all programs.

Example:

```
RESERVE 100
PRINT EXEC(0,"GFABASIC.PRG","","")
```

--> This short program starts by reducing the workspace for BASIC to 100 bytes, it then loads and runs a second copy of the GFA BASIC program.

If the program below is run, the operation of the second copy of GFA BASIC is terminated and returns the value 23 to the original GFA BASIC. This value will then be printed.

```
PRINT "This is the second level GFA BASIC"
QUIT 23
```

Program Tracing

TRON
TRON #n
TROFF

n : iexp

The command TRON (TRace ON) causes each command to be listed on the screen as they are executed. This list can be diverted to a printer or the serial interface by specifying the relevant channel number. The command TROFF turns the TRace OFF again.

Example:

```
PRINT "Start:"
TRON
FOR i%=1 TO 5
  PRINT i%
NEXT i%
TROFF
PRINT "end"
```

--> The word 'Start:'appears on the screen, then the numbers from 1 to 5 are displayed, along with the commands which lead to their display. After that the word 'End' is displayed.

```
OPEN "o", #1,"\tron.lst"
TRON #1
FOR i%=1 TO 10
 PRINT i%
NEXT i%
TROFF
CLOSE #1
'
OPEN "O",#2,"prn:"
TRON #2
FOR i%=10 TO 630 STEP 10
  LINE i%,0,i%,100
NEXT i%
TROFF
CLOSE #2
```

--> The numbers from 1 to 10 are displayed, along with the relevant commands and a row of vertical lines at a distance of 10 pixels. This output is directed to disk or printer.

TRON proc
TRACE$

proc: procedure name

With help of the instruction 'TRON proc', a procedure can be specified which is called before the execution of each individual command. The variable TRACE$ then contains the command which is to be processed next. The command TRON proc makes for very efficient error tracing when used in conjunction with TRACE$. In addition, as well the next command to be processed being displayed, specified variables can be output to screen or printer, allowing changes in the variables to be followed during the course of the program run.

It is important that the TRON procedure should not affect the program itself while running, so while in use, no PRINT commands should be made to the screen (TEXT, ATEXT, ...) and the use of VDI routines should be avoided because of GDOS's use of DEFTEXT, LINE-A, etc.

Example:

```
TRON tr_proc
GRAPHMODE 3
DO UNTIL MOUSEK
  x1%=100+RAND(200)
  y1%=100+RAND(100)
  x2%=200+RAND(200)
  y2%=200+RAND(100)
  PBOX x1%,y1%,x2%,y2%
LOOP
'
PROCEDURE tr_proc
  IF BIOS(11,-1) AND 4   ! Control key
    adr%=XBIOS(2)
    BMOVE adr%+1280,adr%,4*1280
    PRINT AT(1,5);SPACE$(80);
    PRINT AT(1,5);LEFT$(TRACE$,79);
    PAUSE 20
  ENDIF
RETURN
```

--> This program draws randomly-distributed rectangles on the screen. Pressing keys causes the commands just processed to appear on the screen. The program is terminated by the pressing of a mouse button.

DUMP [a$ [TO b$]]

a$, b$: sexp

With the help of the DUMP command, the contents of variables can be displayed during a program run; or labels, procedures and functions listed. In addition the string expression a$ can accept the following:

Examples:

```
DUMP
```

--> Returns all values of variables and the dimensioning of arrays.

```
DUMP "a"
```

--> As above but only operates on variables or arrays which begin with 'a'.

```
DUMP ":"
```

--> Lists all labels and specifies the Editor line number where each label is used. The variation (:b) lists only those labels which begin with 'b'.

```
DUMP "@"
```

--> Lists all procedures and functions and specifies the Editor line number where each is found.

```
proc_name @ 100        (procedure)
func_name FN 200       (function returning a numerical value)
func_name $ FN 300     (function returning a string)
```

Labels, procedures and functions which are no longer defined, are specified without Editor line numbers. If the program listing was saved with the SAVE,A option and then newly reloaded, these undefined names no longer appear. However labels, procedures and function names which are still usable but not defined, are displayed without Editor line numbers.

The Editor line number specified after the name can be jumped to in the Editor by means of Control + G.

When the contents of a string is output, a maximum of 60 characters is displayed. If the character string is longer than this, then the last character displayed is '>'. If a control character, i.e. ASCII value < 32, is to be displayed this is replaced by a full-stop.

The outputs mentioned above can also be directed to a file and, in this case, the filename must be specified in b$.

If none is specified a default extension of .DMP is given.

Chapter 8
Graphics

Three different graphic modes are available on the ATARI ST: one black and white mode and two colour modes. The actual screen coordinates available for the graphic commands and the colours representable with them depend on the current resolution. An overview of the available graphic modes is given below:

	Screen resolution (Coordinates)	**Colours (Colour register)**
Low resolution:	320 x 200 (0 to 319) (0 to 199)	16 (0 to 15) from 512 colours
Medium resolution:	640 x 200 (0 to 639) (0 to 199)	4 (0 to 3) from 512 colours
High resolution:	640 x 400 (0 to 639) (0 to 399)	2 (0 and 1)

In the first section, the commands for the selection of colours (SETCOLOR, COLOR) are covered. After that, the commands for the selection and generation of different types of mouse pointer, display symbols, fill patterns, frames and line-types (DEFMOUSE, DEFMARK, DEFFILL, BOUNDARY, DEFLINE) are explained.

The next section describes the CLIP commands, which are used to trim graphic displays, and the general graphic commands for drawing different geometrical basic forms (PLOT, LINE, BOX, CIRCLE, ELLIPSE). The various options for producing polygons (POLYLINE, POLYMARK, POLYFILL) are described along with the graphic text command TEXT. The section ends with a description of the instruction FILL.

In the last part of this chapter, the treatment of screen sections with SGET, SPUT, GET and PUT is described.

Graphics Definition Commands

SETCOLOR register,red,green,blue
SETCOLOR register,composite
COLOR colour
VSETCOLOR colour,red,green,blue
VSETCOLOR colour,composite

register,red,green,blue,composite,colour : iexp

The first variant of SETCOLOR determines the proportion of the colours red, green and blue in a particular colour register. The intensity of the colour elements is specified on a scale from 0 (low) to 7 (high) and the number of available colour registers depends on the current resolution. In the second variant of SETCOLOR, the colour setting is defined by a single parameter whose value is computed by the following formula:

```
composite = red * 256 + green * 16 + blue * 1
```

and, as with the other setting, the values for red, green and blue are specified on a scale from 0 to 7.

The setting of colour elements in the colour registers is, naturally, only applicable when in the colour modes. However, in the monochrome mode, when a composite value other than 0 or 1 is specified, even numbers have the same effect as 0 and odd numbers the same effect as 1.

The command COLOR determines the text colour. Values between 0 and 15, dependent on the current resolution, are valid.

Example:

```
SETCOLOR 0,0
```

On a mono monitor, the display appears as white on a black background.

VSETCOLOR colour, red, green, blue
VSETCOLOR colour, composite

i,r,g,b,rgb : iexp

Through an apparently unexplained mix-up in the design of the operating system, the colour registers used by SETCOLOR do not correspond directly to the numbers used by COLOR. To overcome this, the command VSETCOLOR is available.

The values of r, g and b are numbers in the range 0 to 7. The term rgb is calculated in the same manner as SETCOLOR, i.e. :

rgb = r * 256 + g * 16 + b * 1

VSETCOLOR 1,2,3,4 is the same as VSETCOLOR 1,&H234

The syntax of the VSETCOLOR command is virtually identical to that of SETCOLOR, the sole difference being in the parameters 'register' and 'colour'. They are related as shown:

For Low Resolution:

```
SETCOLOR   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
VSETCOLOR  0  2  3  6  4  7  5  8  9 10 11 14 12 15 13  1
```

For Medium resolution:

```
SETCOLOR    0  1  2  3
VSETCOLOR   0  2  3  1
```

And in High resolution:

```
SETCOLOR 0,even   corresponds to    VSETCOLOR 0,0
SETCOLOR 0,odd    corresponds to    VSETCOLOR 0,&H777
```

Example:

In Low resolution:

```
FOR i%=0 to 15
  DEFFILL i%
  PBOX i%*20,0,319,199
NEXT i%
DO
  a%=INP(2)
  EXIT IF a%=27
  VSETCOLOR a% AND 15,RAND(-1)
LOOP
```

--> This program draws overlapping coloured boxes, the result being 16 vertical bars (the left hand one is the same colour as the border, i.e. colour 0). By ANDing the key code with 15, a colour register number can be chosen and this register is then set to a random value. The program is left by using the Escape key.

DEFMOUSE symbol
DEFMOUSE bitpattern$

symbol: **iexp**
bitpattern$: **sexp**

The first DEFMOUSE command variant determines the current mouse pointer, selected from the eight pre-defined types. The value of 'symbol' defines the type of pointer in the following way:

0 --> Arrow
1 --> Double clip
2 --> Bee
3 --> Pointing Hand
4 --> Open hand
5 --> Thin cross hairs
6 --> Thick cross hairs
7 --> Bordered cross hairs

The second variant of the DEFMOUSE command allows the user to define a mouse pointer. The action point, the mask colour, the pointer colour, the bit design pattern for the appearance of the mask and the appearance of the mouse pointer are specified by means of a character string. The action point, is that point of the mouse pointer whose coordinates are defined as the mouse position. If the mouse position is interrogated, it is the coordinates of the action point which are returned.

All these values must be entered as word size values and the command MKI$ can be used for this purpose. Thus, bitpattern$ is assembled as follows:

```
bitpattern$ =   MKI$(x-coordinate action point)
              + MKI$(y-coordinate action point)
              + MKI$(1)  ! normal, -1=XOR
              + MKI$(mask colour)
              + MKI$(pointer colour)
              + mask$        ! (bit pattern of the mask)
              + cursor$      ! (bit pattern of the cursor)
```

mask$ and cursor$ consist of 16 words each, each word being the bit pattern of a line.

Example:

```
DEFMOUSE 2
PAUSE 1
m$=MKI$(0)+MKI$(0)+MKI$(1)+MKI$(0)+MKI$(1)
FOR i%=1 TO 16
  m$=m$+MKI$(65535)
NEXT i%
FOR i%=1 TO 16
  m$=m$+MKI$(1)
NEXT i%
PBOX 200,150,400,250
DEFMOUSE m$
REPEAT
UNTIL MOUSEK
```

--> First, a bee appears as a mouse pointer. Then, after one second, the mouse pointer turns into a line. When the mouse is placed on the black background at the centre of the screen, the mouse changes to a rectangle, the mask having been defined as this rectangle. The final loop is then left by pressing a mouse button.

DEFMARK [colour] , [type] , [size]

colour,type,size: iexp

DEFMARK determines the colour, type and size of the marks, at the corner points of a polygon, which are displayed by the command POLYMARK. According to the actual screen resolution, values between 0 and 15 can be assigned to the numerical expression 'colour' (see the start of this chapter). The 'type' parameter gives the following corner marks:

1 --> Full-stop
2 --> Plus sign
3 --> Star
4 --> Rectangle
5 --> Cross
6 --> Diamond

Values larger than 6 result in the use of the star as the corner mark. Values for the 'size' of the shape are specified in pixels. Thus a value of 8 will produce a pointer that is 8 x 8 pixels in size. When only the second or third parameter of the command are required, one can omit the parameters in question and enter only the parameter-separating commas. Thus,

```
DEFMARK ,,4
```

means that the first two parameters keep their current value and the size of the marks is set to 4.

Example:

```
DIM x%(1),y%(1)
x%(0)=50
y%(0)=50
x%(1)=150
y%(1)=150
DEFMARK 1,4,2
POLYMARK 2,x%(),y%()
DEFMARK ,3,4
POLYMARK 2,x%(),y%() OFFSET 100,0
```

--> This draws two pairs of points with different corner marks.

DEFFILL [colour],[style],[pattern]
DEFFILL [colour],bitpattern$

colour,style,pattern : iexp
bitpattern$: sexp

This command determines the fill pattern for the commands PBOX, PCIRCLE, PELLIPSE, POLYFILL and FILL. It sets the colour, style and pattern of the filling and enables one to define one's own patterns. The parameter colour can be assigned values from 0 to 15, depending on the current screen resolution. (See the beginning of this chapter.) The following results are obtained from assignment of values to style:

0 --> empty
1 --> solid
2 --> dots
3 --> hatched
4 --> ATARI symbol (or user-defined)

Patterns can be selected from 24 dot-based patterns or 12 line-based patterns by means of the 'pattern' parameter. (See appendix: Fill Pattern Table). Parameters can be omitted from this definition, as long as the parameter-separating commas are included. Thus,

```
DEFFILL ,2,4
```

selects the fill-pattern 2,4 and leaves the fill colour as previously defined.

In the second variation of the command DEFFILL, using the 32 byte parameter 'bitpattern$', a 16 by 16 pixel pattern can be defined. This information must be presented in word format and can be assembled by means of the MKI$ command.

The medium resolution fill pattern is represented by two bit- planes which are combined to define the actual colours produced. The 16 words for the second bit-plane optionally follow the 16 words for the first.

The first bits from each of these bit-planes are combined, this two bit number (in the range 0 to 3) represents the colour of the pixel at the top left corner of the block. The second pair of bits represents the colour of the second pixel, to the right of the first, and so on.

For low resolution, four bit-planes are needed to represent a colour fill pattern (one plane could be used but this would give a single colour pattern), therefore the bit pattern must be 64 words (128 bytes) in length.

The first bit-plane represents the least significant bit of the colour code. If however the second of these planes is left off, this will result in a single colour fill pattern, the colour being that chosen in the parameter 'colour'.

Examples:

```
DEFFILL 1,2,4
PBOX 10,10,40,40
BOX 50,50,100,100
FILL 70,70
FOR i=1 TO 16
  f$=f$+MKI$(RAND(65535))
NEXT i
BOX 100,100,150,150
DEFFILL 1,f$
FILL 120,120
```

--> Draws two boxes filled with a standard fill pattern and a third filled with a random pattern.

```
DO
  FOR j%=0 TO 15
    f$= ""
    s%=BCHG(s%,j%)
    FOR i%=1 TO 16
      f$=f$+MKI$(s%)
    NEXT i%
    DEFFILL 1,f$
    PBOX 0,0,639,399
  NEXT j%
LOOP
```

--> Parallel vertical bars are displayed and increase in size until they fill the screen. This is done by defining and altering a fill pattern and displaying a rectangle filled with this pattern.

```
FOR i%=1 TO 64          ! 64 for low, 32 for med
'                       ! 16 for high resolution
  READ a%
  a$=a$+MKI$(a%)
NEXT i%
DEFFILL ,a$
PBOX 20,20,300,200
' First Bit-plane:
DATA -1,-1,-1,-1,-1,-1,-1,-1,0,0,0,0,0,0,0,0
' Second Bit-plane:
DATA -1,-1,-1,-1,0,0,0,0,-1,-1,-1,-1,0,0,0,0
' Third Bit-plane:
DATA -1,-1,0,0,-1,-1,0,0,-1,-1,0,0,-1,-1,0,0
' Fourth Bit-plane:
DATA -1,0,-1,0,-1,0,-1,0,-1,0,-1,0,-1,0,-1,0
```

--> This routine creates a fill pattern which, on a mono monitor, has two broad black and white strips. In medium resolution, four strips of half the width are produced in the four possible colours. In the case of the low resolution mode, 16 strips of one line height result, in all 16 colours. This pattern is then used to fill the box created by PBOX.

BOUNDARY n

n : iexp

The command BOUNDARY uses the Function vsf_perimeter to switch off (or on) the border normally drawn round a filled shape (with PBOX, PCIRCLE, etc). When n is not zero, a border is drawn around the fill area, when it is zero, no border is drawn.

Example:

```
DEFFILL 1,2,2
BOUNDARY 1     ! switch on border
PBOX 50,50,100,100
BOUNDARY 0     ! switch off border
PBOX 150,50,200,100
```

--> Draws two filled rectangles, one with and one without a border.

DEFLINE [style] , [thickness] , [begin_s, end_s]

style,thickness,begin_s,end_s : iexp

The command DEFLINE determines the appearance of lines drawn with the commands LINE, BOX, RBOX, CIRCLE, ELLIPSE and POLYLINE. The first parameter 'style' determines the line style, with a choice between pre-defined and user-defined styles. The following styles are available:

1 --> Solid line
2 --> Long-dashed line
3 --> Dotted line
4 --> Dot-dashed (Chain-linked) line
5 --> Dashed line
6 --> Dash dot dot..
7 --> User defined line

User-defined patterns can be created from a 16 bit value in which each bit corresponds to a set point in the monochrome mode.

The second parameter 'thickness' sets the width of the line in pixels and this parameter must have only odd values. If even the line thickness is rounded down to the next smallest odd number. The start and end symbols of a line are defined by means of begin_s and end_s. The available options are:

0 --> Square
1 --> Arrow
2 --> Round

Parameters can be omitted in the definition providing the parameter-separating commas are entered. Thus,

```
DEFLINE ,,1,1
```

specifies an arrow as a symbol for the line beginning and end, leaving style and width unchanged. (See table in the appendix.)

Example

```
FOR i=1 TO 6
  DEFLINE i
  LINE 50,i*50,200,i*50
NEXT i
DEFLINE 1,1,1,2
FOR i=2 TO 12 STEP 2
  DEFLINE ,i
  LINE 250,i*25,400,i*25
NEXT i
DEFLINE -&X101010101010101,1,0,0
LINE 500,10,500,390
VOID INP(2)
```

-> Lines in the six pre-defined styles are drawn followed by lines of different thickness. The line in the user-defined pattern is dotted. Finally, the program waits for a key to be pressed.

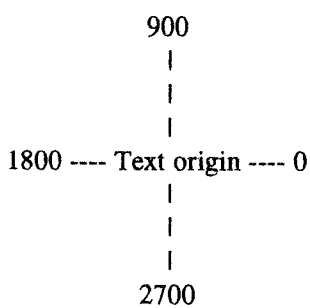
DEFTEXT [colour] ,[attr] ,[angle] ,[height] ,[fontnr]

colour,attr,angle,height,fontnr : iexp

This command determines the appearance of a character string that is to be displayed with TEXT. The parameter 'colour', depending on the current screen resolution, may contain a value between 0 and 15. The second parameter 'attr' sets the text attributes which can be created by combination of the values given below:

0 --> normal
1 --> bold
2 --> light
4 --> italic
8 --> underlined
16 --> outlined

The parameter 'angle' determines the direction of the text characters, the value being specified in 1/10 degree steps in a clockwise direction.



Note, however, that only the following values are permitted by GEM:

0 --> from left to right (as default)
900 --> from bottom to top
1800 --> upside down, from right to left
2700 --> from top to bottom

If a value other than one of the above is given, the nearest multiple of 900 is used.

The parameter 'height' specifies the text height of a capital letter in pixels and, with the normal character sets only the following character heights are actually readable:

4 --> Icon
6 --> Subscript
13 --> Normal character height
32 --> Expanded character height

Finally, the parameter fontnr specifies the number of the desired character set. This font must have previously been installed. (See also GDOS: VST_LOAD_FONT, VQT_NAME, ...)

Example:

```
FOR i|=0 TO 5
  DEFTEXT 1,2^i|,0,13
  TEXT 100,i|*16+100,"This is the text attribute"+STR$(i|)
NEXT i|
```

--> Displays the example text with different attributes.

GRAPHMODE n

n: iexp

The command GRAPHMODE determines the way in which graphics is outputted to the screen with relation to what is already there and is important when pictures are to be drawn on top of one another. Four possible modes can be represented by the numerical expression n, i.e.:

1 --> replace
2 --> transparent
3 --> xor
4 --> reverse transparency (inverted and transparent)

When n has the value 1, the new drawing is simply drawn over the existing screen contents obliterating it completely.

When n is equal to 2, the new drawing is ORed with the existing one, which means that the old picture can still be seen behind the new one.

With n equal to 3, the new drawing is XORed with the existing one. This means that, at each pixel where a graphic point was already present, that graphic dot is deleted and, for all other points, the image is drawn normally. The importance of this mode is that the process is reversible. By XORing the new drawing with itself the original screen is restored. Thus, by using this mode, animation is possible by repeated drawing and 'undrawing' of a figure as it is moved around the screen.

In the case where n = 4, the new drawing is inverted and then ORed with the existing one. In this way, a display similar to mode 2 is produced, (n=2), but the new picture is shown in reverse video.

Example:

```
FOR i%=1 TO 4
  GRAPHMODE i%
  DEFFILL 1,3,8
  PBOX 150*i%-100,10,150*i%,100
  DEFFILL 1,2,10
  PBOX 150*i%-140,50,150*i%-40,150
NEXT i%
```

--> Four filled rectangles are drawn, and a further four are drawn partly overlapping the previous ones. Each pair is combined using a different GRAPHMODE setting.

General Graphics Commands

First of all in this section, the use of the CLIP commands in graphic displays is considered.

The general graphic commands PLOT, LINE, BOX, RBOX, CIRCLE and ELLIPSE are considered next. These draw points, lines, rectangles, rectangles with rounded corners, circles and ellipses and, by means of PBOX, PRBOX, PCIRCLE and PELLIPSE, they can be filled with colours or patterns.

POLYLINE draws a polygonal shape, and corner points made up of symbols defined with DEFMARK can be added using POLYMARK. POLYFILL fills this Polygon with a defined pattern, in a defined colour. POINT returns the colour of a particular screen point. FILL fills a bordered area and TEXT makes it possible to display character strings in arbitrary places on the screen. CLS clears the entire screen. At the end of the section, the command, BITBLT is covered.

The Origin (location 0,0) with all these graphic commands is located in the top-left corner of the screen. The coordinates of graphic elements can lie outside the actual screen display area but only the visible parts of the graphics are represented.

CLIP x,y,w,h [OFFSET x0,y0]
CLIP x1,y1 TO x2,y2 [OFFSET x0,y0]
CLIP #n [OFFSET x0,y0]
CLIP OFFSET x,y
CLIP OFF

x,y,w,h,x0,y0,x1,y1,x2,y2,n : iexp

This group of commands provides the 'Clipping' function, i.e. the limiting of graphic displays within a specified rectangular screen area. CLIP defines a clipping rectangle for the VDI graphic commands. ACLIP does this for Line-A graphic routines. The screen area to be operated on, (clipping rectangle) can be defined by the coordinates of the diagonally-opposite corner points; as well as the top left co-ordinate and the width and height of the clipping rectangle.

The command CLIP x,y,w,h allows the input of the upper y-coordinate 'y', left x-coordinate 'x' as well as the width 'w' and height 'h' of the clipping rectangle.

The command CLIP x1,y1 TO x2,y2 offers a further option by accepting the coordinates of diagonally-opposite corner points (x1,y1) and (x2,y2).

The third variant makes it possible to define the limits of the window 'n'. The optional additional command CLIP OFFSET x0,y0 makes it possible to redefine the origin of the graphic display. In addition, the command CLIP OFFSET x0,y0 can also be used as a command in its own right, and in this case, serves the same purpose in setting the origin for the graphic displays at the point (x0,y0). The command 'CLIP OFF' switches off the clipping function.

The limiting of graphic displays by CLIPping does not apply to the commands GET, PUT and BITBLT, nor to the Line-A calls (where ACLIP should be used) or AES commands.

PLOT x, y
LINE x1,y1,x2,y2
DRAW [TO] [x,y]
DRAW [x1,y1] [TO x2,y2] [TO x3,y3] [TO...]

x,y,x1,y1,x2,y2 : iexp

PLOT draws a point with the coordinates x,y on the screen. LINE draws a line between the coordinate pairs of x1,y1 and x2,y2. The style and colour of this line can be defined by means of the commands DEFLINE and COLOR.

DRAW x,y corresponds to the command PLOT. By means of DRAW TO x,y, a line is drawn between the coordinates x,y and the last set point, regardless of whether this point was set by PLOT, LINE or DRAW.

A further variant of the command, DRAW x1,y1 TO x2,y2 corresponds to the LINE command and, additionally with this command, further coordinates can be specified allowing shapes such as polygons to be produced. This latter variant of the command makes it possible to create structures which are similar to the turtle-graphic commands of LOGO and the Hewlett-Packard standard plotter language, HPGL. In this way, it it is possible to simulate a plotter on the screen.

Examples:

```
x=50
y=50
colour=POINT(x,y)
PRINT colour
PLOT x,2*50
LINE 200,200,400,100
PRINT POINT(x,100)
```

--> The program examines the colour of the point 50,50 and prints this out on the screen. It then plots a second point, plots a line and reports on the colour of the second point plotted.

```
DO
  MOUSE mx,my,mk
  IF mk=1
    DRAW TO mx,my
  ENDIF
  EXIT IF mk=2
LOOP
```

--> When the left mouse key is pressed, a line is drawn between the last set point and the absolute coordinates mx, my. The loop is terminated by pressing the right mouse button.

DRAW expression
DRAW(i)
SETDRAW

i : **iexp**
expression : **a derivative of sexp and aexp, sexp must be first and the individual parts must be separated by a comma, semicolon or inverted comma.**

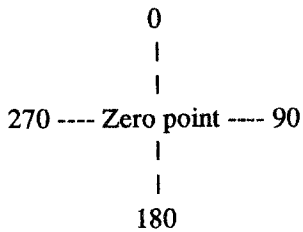
With DRAW an imaginary pen is moved over the screen and draws relative to the last point. The DRAW command's structure resembles the turtle graphic commands of the programming language Logo. The parameters of the DRAW command can contain a large number of individual commands, which are all passed to the command in the form of a string. Parts of the expression can be given in floating-point format, allowing for the use of variables. In this 'LOGO like' convention, an imaginary 'pen' is controlled by means of the graphic commands and its movement over the 'paper' creates the graphic image. The statement below is given as an example of how these commands may be used:

```
DRAW "FD 100 RT ",angle," PU BK";50
```

The available commands are:

FD	n	ForwarD	Moves the 'pen' n pixels 'forward'.
BK	n	BacKward	Moves the 'pen' n pixels 'backwards'.
SX	x	Scale x	Scales the 'pen movement' for FD and BK by the specified factor. The scale with SX and SY works only on the commands FD and BK. With SX0 or SY0 the scale is switched off. (This is quicker than scaling with the factor 1 (SX1,SY1).)
SY	y	Scale y	
LT	a	Left turn	Turns the pen to the left through the specified angle 'a', this being given in degrees.
RT	a	Right turn	Turns the pen as LT but to the right

TT a Turn To Turns the pen to the absolute angle 'a'. (See notation below:)



The data for the angle, 'a' is specified in degrees.

MA x,y Move Absolute Moves 'pen' to the absolute coordinates for x and y

DA x,y Draw Absolute Moves the 'pen' to the absolute coordinates for x and y and draws a line in the current colour from the last position to the point (x,y)

MR xr,yr Move Relative Moves the 'pen' position in the x and y directions relative to the last position.

DR xr,yr Draw relative Moves the 'pen' by the specified displacement relative to its last position and draws a line in the current colour from the last position to this point.

The command SETDRAW x,y,w is an abbreviation for the expression DRAW "MA",x,y,"TT",w.

CO c Colour The colour sets 'c' as 'character colour' (see parameter with COLOR command).

PU Pen UP Lifts the 'pen' up from the 'paper'.

PD Pen Down Lowers the 'pen' down onto the 'paper'.

Additionally the following pointer interrogation functions are available:

DRAW(0) returns x-position
DRAW(1) returns y-position
DRAW(2) returns angle in degrees
DRAW(3) returns the X-axis scale factor
DRAW(4) returns the Y-axis scale factor
DRAW(5) returns the pen flag (-1=PD, 0=PU)

All of these functions return floating point values.

Examples:

```
DRAW "ma 160,200 tt 0" !starts at 160,200 and angle 0
FOR i&=3 TO 10
  polygon(i&,90)   ! draws a Polygon with i& corners
NEXT i&
'
PROCEDURE polygon(n&,r&)    ! n&=number of corners
'                           ! r&=length of sides
LOCAL i&
  FOR i&=1 TO n&
    DRAW "fd",r&," rt ",360/n&
  NEXT i&
RETURN
```

--> Draws a set of polygons with an increasing number of sides.

```
FOR i=0 TO 359 STEP 8
  SETDRAW 320,200,i
  GRAPHMODE 3
  DRAW "fd 45 rt 90 fd 45 rt 90 fd 45 rt 90 fd 45"
  DRAW "bk 90 rt 90 bk 90 rt 90 bk 90 rt 90 bk 90"
  GRAPHMODE 1
  DRAW "fd 45 rt 90 fd 45 rt 90 fd 45 rt 90 fd 45"
  DRAW "bk 90 rt 90 bk 90 rt 90 bk 90 rt 90 bk 90"
NEXT i
```

--> Forms a shape from two small squares and two large ones, and then

rotates it through 360 degrees.

```
l%=48
' Square:
DRAW "ma 60,100 tt 45"
DRAW "fd",l%,"rt 90 fd",l%,"rt 90 fd",l%,"rt90 fd",
                                        l%,"rt90"
' Diamond, tall:
DRAW "mr100,0 tt45"
DRAW "sx0.5 sy0"
DRAW "fd",l%,"rt 90 fd",l%,"rt 90 fd",l%,"rt 90fd",
                                        l%,"rt90"
' Diamond, wide:
DRAW "mr 100,0 tt 45"
DRAW "sx0 sy0.5"
DRAW "fd",l%,"rt 90 fd",l%,"rt 90 fd",l%,"rt90 fd",
                                        l%,"rt90"
' Large diamond, tall
DRAW "mr 100,0 tt 45"
DRAW "sx 3 sy 2"
DRAW "fd",l%,"rt 90 fd",l%,"rt 90 fd",l%,"rt90 fd",
                                        l%,"rt90"
```

--> Draws a square at an angle, followed by three diamonds of various sizes. These diamonds are produced by changing the x and y scales and redrawing the original square.

```
SETDRAW 100,100,90
DRAW "PU FD 40 PD FD 40"
PRINT DRAW(0)              ! X-Co-ordinate
PRINT DRAW(5)              ! Pen flag
```

--> A horizontal line is drawn and the numbers 180 and -1 printed.

BOX x1, y1, x2, y2
PBOX x1, y1, x2, y2
RBOX x1, y1, x2, y2
PRBOX x1, y1, x2, y2

x1,y1,x2,y2 : iexp

BOX draws a rectangle on the screen. The co-ordinates of two opposite corners are specified by x1,y1 and x2,y2. Similarly, PBOX draws a filled rectangle, RBOX a rectangle with rounded off corners and PRBOX a filled rectangle with round corners.

Example:

```
BOX 20,20,120,120
RBOX 170,20,270,120
x=150
DEFFILL 1,2,4
PBOX 20,20 + x,120,120+x
PRBOX 170,20 + x,270,120+x
~INP(2)
```

--> An ordinary rectangle is drawn, followed by one that is filled and with rounded off corners. Finally both shapes are drawn again but with a different fill pattern. A key press is then waited for.

CIRCLE x, y, r [, w1, w2]
PCIRCLE x, y, r [, w1, w2]
ELLIPSE x, y, rx, ry [, w1, w2]
PELLIPSE x, y, rx, ry [, w1, w2]

x,y,r,w1,w2 : iexp

CIRCLE draws a circle with centre coordinates x,y and radius r. Additional starting and ending angles w1 and w2 can be specified to draw a circular arc. Similarly, PCIRCLE draws a filled circle or filled circle segment.

ELLIPSE draws an ellipse with the centre coordinates x,y, horizontal radius rx and vertical radius ry. Optional beginning and ending angles w1 and w2 can be specified to create an elliptical arc. Similarly, PELLIPSE draws a filled ellipse or a filled ellipse segment. The angles should be specified in 1/10 degree (0-3600) and are measured in an anticlockwise direction with zero pointing to the right.

Example:

```
CIRCLE 320,200,100
ELLIPSE 320,200,200,100,900,1800
PCIRCLE 320,200,100,1800,2700
PELLIPSE 320,200,200,100,2700,3600
```

--> A circle is drawn along with three circle or ellipse segments.

POLYLINE n,x(),y() [OFFSET x_off,y_off]
POLYMARK n,x(),y() [OFFSET x_off,y_off]
POLYFILL n,x(),y() [OFFSET x_off,y_off]

n,x_off, y_off : iexp
x(), y() : avar-array

POLYLINE draws a polygon with n corners. The x,y coordinates for the corner points are given in arrays x() and y(). The first corner points are in x(0) and y(0), the last in x(n-1) and y(n-1). The first and last corner points are connected automatically. An optional parameter, OFFSET, can be added to these coordinates, the magnitude of offset being given by x_off and y_off.

POLYFILL fills the polygon with the pattern and colour previously chosen by DEFFILL.

POLYMARK marks the corner points with the shape defined by DEFMARK.

Example:

```
DIM x%(3),y%(3)
FOR i%=0 TO 3
  READ x%(i%),y%(i%)
NEXT i%
DATA 120,120,170,170,70,170,120,120
POLYLINE 4,x%(),y%()
POLYFILL 3,x%(),y%() OFFSET -50,-50
DEFMARK , 4,10
POLYMARK 3,x%(),y%() OFFSET 40,-80
~INP(2)
```

--> Draws an outline triangle and a filled triangle, as well as rectangular corner marks of a further triangle.

POINT(x,y)

x,y : iexp

The colour of the point with coordinates x,y is determined using this function. Values between 0 and 15 are returned for a low resolution screen, between 0 and 3 for medium and 0 or 1 for high resolution.

Example:

```
a=POINT(100,100)
PLOT 100,100
PRINT a, POINT(100,100)
```

--> The computer reads the colour at 100,100, plots a point there and rereads its colour.

FILL x,y [,f]

x,y,f : iexp

This command fills any enclosed area. The filling procedure begins at the coordinates x,y. If the optional parameter 'f' is present then the filling procedure is limited only by points of the colour f and the edge of the screen. If f is not present, or with f=-1, any point with a colour other than the starting point x,y will be taken to be the edge of the area to be filled.

Examples:

```
LINE 0,180,639,180
FOR i=1 TO 19
  BOX i*20,100,i*20 +18,180
  TEXT i*20-4,195,i
  DEFFILL ,2,i
  FILL i*20+1,101
NEXT i
PAUSE 100
FILL 0,180
```

--> This draws a straight line, placing a row of filled boxes above it. After a seconds pause, the computer proceeds to fill in the straight line, and anything joining it, thus partly destroying the fill patterns.

```
LINE 0,280,639,280
FOR i=1 TO 19
  BOX i*20,200,i*20+20-i,280
  TEXT i*20-4,295,i
  DEFFILL ,2,i
  FILL i*20+1,201,1
NEXT i
```

--> The same result as above occurs, except that the resulting fill patterns will be different, due to the extra parameter in the fill command causing a slightly different fill mechanism to be used.

CLS [#n]

n : iexp

Deletes the screen by the output of an ESC-E-CR. Can also be sent to files.

Example:

```
PBOX 100,100,500,200
REPEAT
UNTIL MOUSEK
CLS
```

--> Fills the screen partly with a rectangle and deletes it after a mouse button is pressed.

TEXT x,y [,l],expression

x,y,l : iexp
expression : sexp or aexp

Displays the text in 'expression' starting at the point with the graphic coordinates x,y. This point refers to the bottom left corner of the first character of the expression. The parameter l sets the length of the text in pixels. With l positive, the spacing between characters will be adjusted to achieve this length, whereas with l negative, the length is created by altering the size of the spaces between words. When l is zero, the unchanged text is displayed.

Using DEFTEXT, various attributes of the text can be altered. DEFTEXT however, only works with the TEXT command and with the PRINT command when used inside a window.

Example:

```
s$= "this is an example"
FOR i=0 TO 23
  DEFTEXT 1,i,0,6
  TEXT 50,i*16+16,s$
NEXT i
DEFTEXT 1,0,0,13
TEXT 350,50,350-50
TEXT 350,100,s$
TEXT 350,150,250,s$
TEXT 350,200,-250,s$
~INP(2)
```

--> Writes text in various forms on the screen and then waits for a key to be pressed.

SPRITE bit_pattern$ [,x,y]

bit_pattern$: **svar**
x,y : **aexp**

The SPRITE command enables a 16x16 pixel block to be moved around the screen. The appropriate bit information for the pattern and its mask is put into the string 'bit_pattern$'. All values must be given in word size. For this purpose one can use the command MKI$, so bitpattern$ is formed as follows:

```
bitpattern$ =   MKI$(x-coordinate of action point)
              + MKI$(y-coordinate of action point)
              + MKI$(0)    ! for normal or MKI$(-1) for XOR
              + MKI$(mask colour)    ! mostly 0
              + MKI$(Sprite colour)    ! mostly 1
              + sprite$
```

Contained in sprite$ is the bit information for the sprite shape and its mask, which must be specified, unlike DEFMOUSE, not successively but alternately.

Example:

```
gfa$=MKI$(1)+MKI$(1)+MKI$(0)
gfa$=gfa$+MKI$(0)+MKI$(1)
FOR i%=1 TO 16
  READ pattern%, mask%
  gfa$=gfa$+MKI$(mask%)+MKI$(pattern%)
NEXT i%
DATA 0,0,0,32256,15360,16896,8192,24064
DATA 8192,24560,11744,21008,9472,23280,9472,23295
DATA 15838,16929,274,32493,274,749,286,737
DATA 18,1005,18,45,18,45,0,63
REPEAT
  ADD mx%,(MOUSEX-mx%)/50
  ADD my%,(MOUSEY-my%)/50
  SPRITE gfa$,mx%,my%
UNTIL MOUSEK=2
```

--> A Sprite moves over the screen, following the movement of the mouse pointer.

Grabbing Sections of Screen

SGET screen$
SPUT screen$

screen$: svar

SGET copies the entire screen (32000 bytes) into a string. Similarly, SPUT copies a 32000 byte long string into the screen memory, thus displaying it.

```
PCIRCLE 100,100,50
SGET b$
~INP(2)
CLS
~INP(2)
SPUT b$
```

--> Draws a filled circle on the screen. After a key is pressed it disappears, after a further key depression it reappears.

GET x1,y1,x2,y2,section$
PUT x1,y1,section$ [,mode]

x1,y1,x2,y2,mode : **iexp**
section$: **svar**

GET puts a section of screen into a string variable (x1,y1 and x2,y2 are coordinates of diagonally opposite corners). Similarly PUT places a screen section (read in with GET) onto the screen at coordinates x1,y1. Using 'mode' it is possible to control the way string is placed on the screen in relation to the existing screen contents. In following table the relationship between the new picture and the existing one are shown for each value of mode. The term 's' represents a pixel from the new picture (the source picture), and 'd' the corresponding pixel from the existing screen (the destination).

Mode	Placing rule	Effect
0	0	All points are cleared.
1	s AND d	Only the points which are set in both screens remain set.
2	s AND (NOT d)	Sets only the points which are set in the source and clear in the destination.
3	s	The new source screen is simply transferred (GRAPHMODE 1-Default).
4	(NOT s) AND d	Only the points which are clear in the source and set in the destination are set.
5	d	The screen remains unchanged.
6	s XOR d	Only those points are set in one but not both remain set (GRAPHMODE 3).
7	s OR d	All points are set in which either or both the source and destination are set (GRAPHMODE 2).
8	NOT (s OR d)	All points which are clear in both screens become set.
9	NOT (s XOR d)	All points where both source and destination are set, or both are clear, are set.

10	NOT d	The destination screen is inverted.
11	s OR (NOT d)	A point is set if either the source is set, or the destination is clear, or both.
12	NOT s	The source screen is inverted before the placing.
13	(NOT s) OR d	GRAPHMODE 4
14	NOT (s AND d)	All points which were not set in both screens become set.
15	1	All points are set.

The important modes are:

3 Replace (default)
4 XOR
7 Transparent
13 Inverse transparent

In addition, if the current fill pattern is a user-defined one and bit 4 of 'mode' is set, then the result of the above calculations will be ANDed with the user-defined pattern.

VSYNC

This command is used for the synchronisation of the screen display. When this command is issued, the computer pauses until the vertical scanning line (raster scan) reaches the top of the screen. This results in much less screen flicker. VSYNC can be used, for example, in the animation of screen sections using GET and PUT.

Example:

```
t%=TIMER
FOR i%=1 TO 100
  VSYNC
NEXT i%
PRINT SUB(TIMER,t%)/200
```

--> This displays the time for 100 scans of the screen to occur.

BITBLT s_mfdb%(),d_mfdb%(),par%()

s_mfdb%(),d_mfdb%(),par%() : integer-array

The command BITBLT allows the copying of rectangular screen sections. It is similar to the commands GET and PUT but it is quicker and more flexible. However, it is also more complicated to use.

The parameters of the command are stored in three arrays. The first one, s_mfdb%(), contains the structure of the source screen - the one to be copied. In the same way, d_mfdb%() contains the structure of the destination, i.e. the place where the picture is to be copied to. The third array contains the coordinates of the source and target areas and also the copy mode.

This command has a VDI-routine as its basis. During BITBLT adr% and BITBLT x%() a LINE-A routine is called (see section on LINE-A calls).

The structure of the source (s_mfdb%) is same as that of the destination screen (d_mfdb%). The abbreviations mean:

s_mfdb%() SOURCE MEMORY form description block
d_mfdb%() DESTINATION MEMORY form description block

The Array Elements are:

_mfdb%(0)	Contains the source/destination address. This address must be an even number. Usually either s_mfdb%(0) or d_mfdb%(0) equals the screen address (XBIOS(2)).
_mfdb%(1)	Width of the screen in pixels. This value must be divisible by 16.
_mfdb%(2)	Height of the screen in pixels.
_mfdb%(3)	Screen width in words (= pixel count/16).
_mfdb%(4)	Reserved, always 0.
_mfdb%(5)	Number of bit planes: High resolution = 1 Medium resolution = 2 Low resolution = 4

_mfdb%(6) to _mfdb(6) are reserved for future extensions.

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

The array par%() has the following structure:

par%(0)	Left x-coordinate of the source block
par%(1)	Upper y-coordinate of the source block
par%(2)	Right x-coordinate of the source block
par%(3)	Lower y-coordinate of the source block
par%(4)	Left x-coordinate of the destination block
par%(5)	Upper y-coordinate of the destination block
par%(6)	Right x-coordinate of the destination block
par%(7)	Lower y-coordinate of the destination block
par%(8)	Copy mode

The values for 'copy mode' correspond to those with GET/PUT. The important ones are:

3 =	Replace	(GRAPHMODE 1)
6 =	XOR	(GRAPHMODE 2)
7 =	Transparent	(GRAPHMODE 3)
13 =	Inverse transparent	(GRAPHMODE 4)

Example:

```
DIM smfdb%(8),dmfdb%(8),p%(8)
'
FOR i%=0 TO 639 STEP 8
  LINE i%,0,639,399
NEXT i%
'
GET 0,0,639,399,a$
mirrorput(0,0,a$)
'
PROCEDURE mirrorput(x%,y%,VAR x$)
  IF LEN(x$)>6          ! Only if something there
    a%=V:x$
    b%=INT{a%}
    h%=INT{a%+2}
    '
    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)
    dmfdb%(1)=640
    dmfdb%(2)=400
    dmfdb%(3)=40
    dmfdb%(5)=1
    '
    p%(1)=0
    p%(3)=h%
    p%(4)=x%+b%
    p%(5)=y%
    p%(6)=x%+b%
    p%(7)=y%+h%
    p%(8)=3
```

```
      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
```

--> Draws a set of lines, forming a triangle. The entire screen is read into a string and then each pixel-wide column is placed back on the screen on the other side. This has the effect of reflecting the screen in an axis down the centre of the screen (cf. BITBLT in the section on LINE-A calls).

Chapter 9
Event, Menu and Window Management

Event Management

There are commands in GFA BASIC which allow the monitoring of GEM Events in a straightforward way. These events are the depression of a key or a mouse button, the arrival of the mouse pointer inside or outside one of two specified rectangular screen areas, and the arrival of a 'GEM message', in which information about window management is passed.

The monitoring of these events is set up by ON MENU xxx GOSUB, where xxx is the event to be reacted to, and is actually invoked within a program by the ON MENU command. Each time this command is encountered, a check is made to see if an Event has occurred. If so, and if there was a previous ON MENU xxx GOSUB to define the reaction to that event, then the program branches to the appropriate procedure.

ON MENU [t]

t : iexp

The command ON MENU supervises EVENT handling. Before using it, the required reaction should have been specified with an ON MENU xxx GOSUB command, the variants of which are explained in the remainder of this chapter. For constant supervision of Events it is necessary to use this command repeatedly. For this reason, the ON MENU command is normally found in a loop.

The parameter t contains the time (in thousandths of a second) to elapse before the ON MENU command is terminated. The reason for this delay is that occasionally GEM does not notice the releasing of a mouse button (typical effect: When the 'Close' box of a window is clicked, the window sometimes remains open until the button is energetically clicked a few times). By giving a suitable value to 't', this should be prevented.

It is a good idea to use the parameter 't' even if a program is not specifically concerned with the mouse buttons, as other programs, or GEM routines called from within it, may do so.

Example:

```
ON MENU BUTTON 1,1,1 GOSUB test
t%=TIMER
REPEAT
  PRINT (TIMER-t%)/200
  ON MENU 2000
UNTIL MOUSEK=2
PROCEDURE test
RETURN
```

--> The time since the program started is displayed every two seconds. If the left mouse button is pressed, then an Event occurs, and ON MENU is terminated before the expiry of the two second period. A press of the right mouse button ends the program. If one changes the first line to 'ON MENU BUTTON 0,0,0 GOSUB test', then the mouse monitoring is switched off, and the time parameter behind ON MENU will have no effect.

MENU(x)

x : aexp (from -2 to 15 inclusive)

The variables MENU(-2) to MENU(15) contain all the relevant information from an Event. In the case when an item in a menu is selected, the index of the selected item in the item list will be found in MENU(0). (See next section on Pull-down Menus).

MENU(-2) and MENU(-1) contain the address of the Message buffer and the address of the menu Object tree respectively. The Message Buffer lies in the variables MENU(1) to MENU(8) and the AES-Integer Output Block in MENU(9) to MENU(15). The use of these variables to determine specific information is only briefly discussed here, starting with MENU(1) and the Message Buffer.

The Identification number of an Event when it occurs can be found in MENU(1). The other elements of the message buffer contain various values, the interpretation of which depends on the value in MENU(1), as shown in the following table, where different possible values of MENU(1) are listed together with the meanings of other relevant MENU(x) variables in each case. The values relating to window management will tend to be the most extensively used.

MENU(1) = 10	A Menu item was chosen:
MENU(0)	Menu item index in the item list
MENU(4)	Object number of the menu title
MENU(5)	Object number of the chosen menu item
MENU(1) = 20	A redraw of a rectangular window area is required:
MENU(4)	Identification number (handle) of the window
MENU(5),(6)	Coordinates of the top left corner of the area
MENU(7),(8)	Width and height of the area (See ON MENU MESSAGE GOSUB for an example)

MENU(1) = 21 — A window was clicked (this normally means that the user wishes to activate this window):

MENU(4) — ID (handle) of the clicked window

MENU(1) = 22 — The 'Close' box of a window was clicked:

MENU(4) — ID of the window

MENU(1) = 23 — The 'Full' box (top right) of a window was clicked (this normally means that the user wants to bring that window to maximum size):

MENU(4) — ID of the window

MENU(1) = 24 — One of the four arrow boxes or a slider bar area of the window border was clicked. The movement of the slider bar is reported by MENU(1)=25 or 26. MENU(1)=24 only shows that one of the gray areas was clicked:

MENU(4) — ID of the window

MENU(5) — The area that was clicked:

- 0: Above the vertical slider
- 1: Below the vertical slider
- 2: Up arrow
- 3: Down arrow
- 4: To the left of the horizontal slider
- 5: To the right of the horizontal slider
- 6: Left Arrow
- 7: Right Arrow

MENU(1) = 25 — The horizontal slider was moved:

MENU(4) — ID of the window

MENU(5) — Position of the moved slider (a number between 1 and 1000)

MENU(1) = 26 — The vertical slider was moved:

MENU(4) — ID of the window

MENU(5) — Position of the moved slider (a number between 1 and 1000)

MENU(1) = 27 The size of the window was changed by means of the 'sizing' box (lower right):

MENU(4)	ID of the window
MENU(5),(6)	New x and y coordinates of top left corner
MENU(7),(8)	New width and height of the window

MENU(1) = 28 The position of a window was changed:

MENU(4)	ID of the window
MENU(5),(6)	New x and y coordinates of top left corner
MENU(7),(8)	New width and height of the window

MENU(1) = 29 A new GEM window was activated. This can happen with the closing of another active window, for example one which was used by a Desk Accessory:

MENU(4)	ID of the window

MENU(1) = 40 An accessory was selected. This value can only be received by an accessory, which should check the value in MENU(5) to see if that accessory is the one referred to, or if another has been started:

MENU(5)	Menu identification number of the accessory

MENU(1) = 41 An accessory was closed. This value can only be received from an accessory, which should do a check as for MENU(1)=40:

MENU(5)	Menu identification number of the accessory

The variable MENU(9) contains bit information on which kind of event has occurred. If the bit for the appropriate event is set, the variables MENU(9) to MENU(15) and GINTOUT(0) to GINTOUT(7) will contain information as follows:

Bit 0 --> Keyboard
Bit 1 --> Mouse button
Bit 2 --> Mouse has entered/left rectangle 1
Bit 3 --> Mouse has entered/left rectangle 2
Bit 4 --> A message arrived in the message buffer
Bit 5 --> Timer event

MENU(10) x-position of the mouse when event terminated
MENU(11) y-position of the mouse when event terminated
MENU(12) Mouse buttons pressed:
0 --> None
1 --> Left
2 --> Right
3 --> Both buttons

(See ON MENU BUTTON x,y,z GOSUB for an example)

MENU(13) supplies the status of the keyboard 'shift' keys; for each pressed key a bit is set as follows:
Bit 0 --> Right shift
Bit 1 --> Left shift
Bit 2 --> Control
Bit 3 --> Alternate

(See ON MENU KEY GOSUB for an example)

MENU(14) gives information about a pressed key. The low-order byte contains the ASCII code of the character, and the high-order byte the keyboard Scan code.

(See ON MENU KEY GOSUB for an example.)

MENU(15) returns the number of mouse clicks (single click, double click, etc) that caused the event.

ON MENU BUTTON clicks,but,state GOSUB proc

clicks,but,state : iexp
proc : procedure name

This sets up the action to be taken when one or more clicks of the mouse are received. With a subsequent ON MENU command, the named procedure will be branched to if the conditions imposed by the parameters are met:

clicks > Sets the maximum number of clicks which generate a response.

button > The expected button combination as follows:

0 --> Any?
1 --> Left
2 --> Right
3 --> Both

state > Specifies which button state (up or down) will cause the Event. With state=0, the Event will be prompted by the button(s) being up, and with state=1, the button(s) being down will cause the Event.

proc > The name of the procedure to which the program will branch if the above conditions are met.

Example:

```
ON MENU BUTTON 1,1,0 GOSUB box
GRAPHMODE 3
REPEAT
  ON MENU
UNTIL MOUSEK=2
'
PROCEDURE box
  ADD i%, 7
  IF i%>200
    i%=3
  ENDIF
  BOX 320-i%,200-i%,320+i%,200+i%
RETURN
```

--> Boxes increasing in size are drawn on the screen so long as the left mouse button is not pressed. The program will terminate if the right mouse button is pressed.

ON MENU KEY GOSUB proc

proc : procedure name

This command enables the monitoring of the keyboard. The parameter proc is the name of a procedure to which the program branches, if a key was pressed during an ON MENU command.

Example:

```
ON MENU KEY GOSUB key_output
REPEAT
  ON MENU
UNTIL MOUSEK=2
'
PROCEDURE key_output
  PRINT "keyboard shift keys: "; MENU(13)
  PRINT "ASCII-code: "; BYTE(MENU(14))
  PRINT "Scan-code: "; SHR(MENU(14),8)
  PRINT
RETURN
```

--> With press of a key the current condition of the keyboard shift keys (shift, control, Alternate) is announced as well as the ASCII-code and Scan-code of the pressed key. A press of the right mouse button terminates the program. For the meaning of MENU(13) and MENU(14) see MENU(x) .

ON MENU IBOX n, x, y, b, h GOSUB proc
ON MENU OBOX n, x, y, b, h GOSUB proc

n, x, y, b, h : iexp
proc : procedure name

These two commands monitor the mouse coordinates. If the mouse enters (IBOX) or leaves (OBOX) a rectangular display area, the procedure proc is branched to.

It is possible to define two rectangular display areas which are supervised separately. 'n' is the number (1 or 2) of the appropriate rectangle, x its left x-coordinate, y the upper y-coordinate, b the width and h the height of the rectangle. The monitoring takes place during the execution of an ON MENU command.

Example:

```
ON MENU IBOX 1,250,130,140,140 GOSUB enter_box
ON MENU OBOX 2,50,50,540,300 GOSUB exit_box
'
GRAPHMODE 3
BOX 250,130,390,270
BOX 50,50,590,350
REPEAT
  ON MENU
UNTIL MOUSEK=2
'
PROCEDURE enter_box
  BOX 250+i%,130+i%,390-i%,270-i%
  IF i%=70
    i%=0
  ENDIF
  ADD i%,2
RETURN
'
PROCEDURE exit_box
  BOX 0+j%,0+j%,639-j%,399-j%
  IF j%=0
    j%=50
  ENDIF
  SUB j%,2
RETURN
```

--> When the mouse enters the inner rectangle, shrinking boxes appear inside it. when it leaves the outside rectangle, growing boxes appear there. Pressing the right mouse button terminates the program.

ON MENU MESSAGE GOSUB proc

proc : procedure name

If a message arrives in the message buffer, then the program branches to the procedure with the name proc. The monitoring of the message buffer takes place with each ON MENU command. The structure of the message buffers is discussed in the section concerning MENU(x) .

Example:

```
DIM m$(10)
FOR i%=0 TO 10
  READ m$(i%)
NEXT i%
DATA Desk, Redraw, ------------------------
DATA 1,2,3,4,5,6,"",""
OPENW 4,0,0
MENU m$()
ON MENU MESSAGE GOSUB read_message
PRINT AT(1,1);
REPEAT
  ON MENU
UNTIL MOUSEK=2
'
PROCEDURE read_message
  IF MENU(1)=20
    PRINT CHR$(7);
    PRINT "A screen section"
    PRINT "must be redrawn"
  ELSE
    PRINT CHR$(7);
    PRINT "Something has happened !!!"
  ENDIF
RETURN
```

--> To test this program, an accessory must be loaded. If selected, then the program announces that a screen section was covered and has to be redrawn (see also MENU(x)). At the beginning this message appears once. The program can be terminated through a press of the right mouse button.

Pull-down Menus

In this section the commands specific to GFA BASIC 3 for control of pull-down-menus are given. Unfortunately a certain confusion prevails in the literature on this topic as to the meaning of some of the terms used. They shall therefore be stated here as they are used in this manual. We use the term 'pull-down-menu' as a general term for menu titles or headings. In the top screen line the constantly visible part of the menu, is where the menu list is located. This contains the individual headings. If the mouse arrow arrives at one of these headings, then under it a so-called menu unfolds. Each part of this menu can be selected individually as a menu entry. This choice of terminology is not generally obligatory but is used throughout this manual, elsewhere slightly different definitions are used.

During the creation of a menu its entries are set in an array m$(). With the command MENU m$() this pull-down-menu is displayed on the screen. The command ON MENU GOSUB determines which procedure the program branches to on the selection of a menu entry. When the program is running, a check is made to see whether an entry was selected on each occurrence of an ON MENU-command.

The command MENU OFF returns reverse video entries in the menu list to normal display. With MENU KILL the menu is switched off. The command MENU x,y enables menu entries to be provided with ticks or to be displayed in light text, making this menu entry non-selectable.

ON MENU GOSUB proc
MENU m$()

proc : procedure name
m$() : string-array

These two commands are responsible for generating and managing a menu, and are supported by the commands and variables in the previous section (ON MENU, MENU()).

With ON MENU GOSUB proc, the procedure to which control will be passed on selection of a menu entry is determined. If an accessory is currently open, the procedure will not be called. Within the procedure, which menu entry was selected can be found by using the variable MENU(0). MENU(0) is the index of the selected entry in the array of the entries m$(), and m$(MENU(0)) indicates the text clicked on the menu if OPTION BASE 0 is currently selected. If OPTION BASE 1 has been chosen, then the text of the selected item is in m$(MENU(0)+1).

The command MENU m$() puts the menu onto the screen. The string array m$() contains the headings, entries and reserved space for the accessories. The following format must be used during the arrangement of the entries in the array m$():

m$(0)	Heading of the first menu in which accessories can exist
m$(1)	Name of the first entry in the first menu
m$(2)	A line of of minus signs
m$(3) - m$(8)	Reserved space for accessories These elements need only be a single character long, as their contents are ignored. If an accessory was loaded when the computer was switched on, this will take up one of these strings. Otherwise they will not be printed.
m$(9)	an empty string, which marks the end of the first menu

All further menus have the following format:

1. Heading of the menu.
2. List of the menu entries.
3. An empty string which marks the end of the menu.

After the last menu another further empty-string marks the end of the entire Pull-down-menu. A menu entry which begins with a minus sign is represented but is not selectable and is shown in light text.

Example: See end of this chapter.

MENU OFF
MENU KILL

MENU OFF returns a menu title to 'normal video' display mode. (After an item is chosen from a menu, the menu title is displayed in inverse video).

MENU KILL deactivates a menu but does not, however, remove the menu title list from the screen. In addition MENU KILL turns off the ON MENU GOSUB options.

MENU x,y

x,y : aexp

With this instruction the x-th entry in a menu can be given certain attributes. The numbering of the entries corresponds to the indexing of the array of menu entries, counting from zero and including titles, entries for Accessories and null strings ("").

The second parameter y specifies the attribute to be given, or removed from, the x-th menu entry as follows:

y		Effect
0	-->	Tick to be removed, if present, from in front of a menu item
1	-->	Tick to be installed in front of a menu item
2	-->	Menu item to be made non-selectable, and printed light
3	-->	Menu item to be made selectable, and written in normal characters

Example: See this end of the section.

Sample Pull-down menu program:

```
DIM entry$(20)
DATA "  Desk  "," Test  "
DATA ----------------,1,2,3,4,5,6,""
DATA "  File  "," Load  "," Save  "
DATA ---------," Quit  ",""
DATA "  Titles  "," Entry 1  "," Entry 2  ","",""
DATA End
'
i%=-1
REPEAT
  INC i%
  READ entry$(i%)
UNTIL entry$(i%)="End"
entry$(i%)=""
'
MENU entry$()
ON MENU GOSUB evaluate
OPENW 0
'
REPEAT
  ON MENU
UNTIL MOUSEK=2
'
PROCEDURE evaluate
  MENU OFF
  ' MENU(0) contains array index of selected item
  m%=MENU(0)
  PRINT entry$(m%)
  '
  ALERT 0,"Tick before item?",0,"YES|NO",a%
  IF a%=1
    MENU m%,1
  ELSE
    MENU m%,0
  ENDIF
  '
```

```
  ALERT 0,"Lightened characters | (Not selectable)
",0," Yes| no ",a%
  IF a%=1
    MENU m%,2
  ELSE
    MENU m%,3
  ENDIF
RETURN
```

--> A menu is created and monitored. When a menu item is selected, its text is printed and the user is asked first if it is to be ticked, and then whether it should be 'non-selectable' or not.

Window Commands

GFA-BASIC offers a number of commands for simple window management: (OPENW, CLOSEW, CLEARW, TITLEW, INFOW), but if one wants to program windows more efficiently, taking full advantage of the GEM facilities, then the appropriate AES-routines will need to be used (see Chapter 11 - AES Libraries). Additionally the functions W_HAND and W_INDEX are available as a link between the simpler window management commands and the more extensive AES functions in the Window library.

With the simplest of the instructions, OPENW, all four windows share a common corner, their other corners all lying on the screen edges. When this method is used, only one pair of coordinates is needed to define the size and position of all four windows, these being the coordinates of the common corner.

After a window has been opened, most commands (PRINT, PRINT AT, TEXT etc.) will take as their origin, the top left corner of the work-area of the window but, commands such as GET, PUT and BITBLT, which access the screen memory directly, will still take the top left corner of the screen for their origin. Graphics and text which leave the current window area are automatically clipped.

OPENW nr [,x_pos,y_pos]
OPENW #n,x,y,w,h,attr

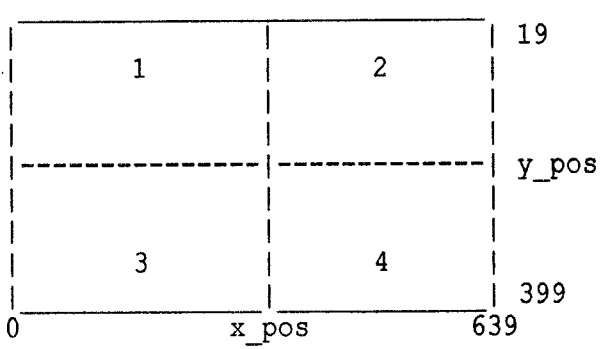
CLOSEW nr
CLOSEW #n

nr,x_pos,y_pos : aexp
n,x,y,w,h,attr : iexp

With 'OPENW nr' window number nr is opened. The parameters x_pos and y_pos determine the position of the 'free' window corner, i.e. the one which is not on an edge of the screen. The AES routines, 'OPENW #n' or WINDTAB are required for more flexible window management. The coordinates of the possible windows set up with this instruction (assuming high-resolution mode) will be:

No.	Top left corner	Lower right corner
1	(0,19)	(x_pos,y_pos)
2	(x_pos,19)	(639, y_pos)
3	(0, y_pos)	(x_pos,399)
4	(x_pos,y_pos)	(639,399)

i.e.:



The point (x_pos,y_pos) is thus the point of contact of the four possible windows.

With the instruction OPENW 0 no genuine window is opened, but the coordinate origin is moved to (0,19). Thus the upper 19 lines of the screen are protected from graphical or text output. This usefully protects a menu bar against accidental overwriting.

The instruction 'CLOSEW nr' closes the window with number nr, 'CLOSEW #n' the window with the arbitrarily assigned number n.

Example: (See also sample program at the end of the section.)

```
REPEAT
  IF MOUSEK=1
    CLOSEW 1
    OPENW 4,320,200
  ENDIF
  IF MOUSEK=2
    CLOSEW 4
    OPENW 1,100,100
  ENDIF
UNTIL MOUSEK=3
CLOSEW #1
CLOSEW #4
```

--> Pressing the left mouse button opens window 4 and closes window 1, or pressing the right button opens window 1 and closes window 4. Simultaneously pressing both buttons terminates the program.

The second variant, OPENW #n, opens a window with the arbitrarily assigned number 'n', with the position, size and attributes specified in x, y, w, h and attr. The expression attr determines which components (title bar, sliders etc.) the window is to have (see WINDTAB below or WIND_CREATE in the AES section). 'n' is then the number to be used with TITLEW, INFOW, etc. - it is NOT the GEM 'handle' of the window.

CLOSEW #n closes such a window.

Example:

```
TITLEW #1," Title 1 "     ! Gives title to window #1
INFOW #1,STRING$(15,"...| ") ! Allocates Info. line
OPENW #1,16,32,600,300,&X111111111111 ! Sets coords
'                  ! + attributes and opens a window
~INP(2)
CLOSEW #1                  ! Important! Closes window
```

--> Opens a window with a title and an info line. Pressing a key terminates the program.

W_HAND(#n)
W_INDEX(#hd)

n,hd : aexp

W_HAND returns the GEM 'handle' (Identification Number) of the window whose 'channel number' was specified in n. W_INDEX performs the inverse function and returns the window number for the specified GEM handle.

Note: The window number is the value used in the simple window control commands. The GEM handle is a different value which refers to the same window. The GEM handle should be used to specify a particular window when using the AES routines.

Example:

```
OPENW 2
PRINT W_HAND(#2)
~INP(2)
CLOSEW #2
```

--> Prints the 'handle' of the window numbered 2 on the screen. Pressing a key terminates the program.

CLEARW [#] n
TITLEW [#] n,title$
INFOW [#] n,info$
TOPW #nr
FULLW [#] n

The instruction CLEARW deletes the contents of window number 'n'. TITLEW writes the text in 'title$' in the top line of the window. INFOW writes the text in 'info$' on the second (information) line of the window, and TOPW activates the window number n. FULLW brings window n to full screen size. CLEARW[#]n clears every visible part of a window, without activating it. Internally it is done by WIND_UPDATE and WIND_GET.

Example: (See also sample program at the end of the section.):

```
DEFFILL 1,2,4
PBOX 0,0,639,399
OPENW 1
PAUSE 50
FULLW #1
PRINT " Window 1"
OPENW 4,100,100
PAUSE 50
CLEARW 1
OPENW 3
PAUSE 50
TOPW #1
PAUSE 50
CLOSEW #1
TITLEW 4," Window 4 "
INFOW 3," Window 3 "
PAUSE 100
CLOSEW #3
CLOSEW 4
```

--> Some windows are opened, altered and then closed again.

WINDTAB
WINDTAB(i,j)

i,j : iexp

The value of WINDTAB gives the address of the Window Parameter table, where the information which determines the appearance of a window is stored. (The next piece of information following the table is the coordinates of the graphics origin.)

The table consists of 68 bytes and is constructed in word (2-byte) format. The use of the table is shown at the end of the section in a sample program, where the parameters of the window to be created are placed directly into the window table and the window then opened with a simple OPENW instruction.

WINDTAB, in a similar way to INTIN(), etc., can be used as a two-dimensional array, WINDTAB(). The first index refers to the number of the window (1 to 4, or 0). The second index is:

0 Handle
1 Attributes
2 x-coordinate
3 y-coordinate
4 Width (external)
5 Height (external)

The Window Parameter table can also be modified by using 'DPOKE WINDTAB+offset', the offsets having the following meanings:

Offset	Description
0	Handle of window 1
2	Attributes for window 1 (see structure below)
4	X-coordinate for window 1
6	Y-coordinate for window 1
8	Width of window 1
10	Height of window 1
12 to 22	Corresponding values for window 2
24 to 34	Corresponding values for window 3
36 to 46	Corresponding values for window 4
48	-1
50	0
52 to 58	Coordinates and size for the Desktop window (0)
60 and 62	Coordinates of the 'join point' of the four windows
64 and 65	Origin for graphic instructions (CLIP OFFSET)

This graphic origin is applicable to AES, Line A and direct VDI calls but PUT, GET and BITBLT always take the top left corner of the screen for their origin.

The window attribute word is constructed bit by bit, with each set bit denoting the presence of a particular window component.

Bit	Associated window element
0	Window title
1	'Close' box (top left)
2	'Full' box (top right)
3	'Move' line, with which the window can be shifted
4	Information line
5	'Sizing' box (bottom right)
6	Up arrow
7	Down arrow
8	Vertical slider bar (right)
9	Left arrow
10	Right arrow
11	Horizontal slider bar (bottom)

Example:

```
' It is also possible to imply WINDTAB manipulation
' using the OPENW #n,x,y,w,h,attr instruction,
' where with Version 2 only WINDTAB was available.
'
OPENW #1,100,120,200,70,&HFFF
'
' corresponds to
'
DPOKE WINDTAB+2,&HFFF
DPOKE WINDTAB+4,100
DPOKE WINDTAB+6,120
DPOKE WINDTAB+8,200
DPOKE WINDTAB+10,70
OPENW 1
'
' or
'
WINDTAB(1,1)=&HFFF
WINDTAB(1,2)=100
WINDTAB(1,3)=120
WINDTAB(1,4)=200
WINDTAB(1,5)=70
OPENW 1
```

Other Window-related Commands

RC_INTERSECT(x1,y1,w1,h1,x2,y2,w2,h2)

x1,y1,w1,h1 : iexp
x2,y2,w2,h2 : ivar

The function RC_INTERSECT (rectangle intersection) can be used to find whether two rectangles overlap. The two rectangles are specified by the coordinates of the of the top left corner (x,y), the width w and height h.

If the rectangles overlap, then the logical value TRUE (-1) is returned and, after the function call, x2,y2,w2,h2 will contain the coordinates and size of the rectangular area which is common to both rectangles.

If they do not overlap, FALSE (0) is returned and x2,y2,w2,h2 will contain the coordinates and size of a rectangle which lies between the two specified rectangles. In this case, either the width w2 or the height h2, or both, will be negative or zero.

This function is normally used for the control of 'Redraws' with GEM windows.

Example:

```
BOX 100,100,400,300
x=200
y=200
w=300
h=150
BOX x,y,x+w,y+h
'
IF RC_INTERSECT(100,100,300,200,x,y,w,h)
  PBOX x,y,x+w,y+h
ENDIF
```

--> Two rectangles are drawn and the common area is represented in black.

```
oldmx=0
oldmy=0
DO
  MOUSE mx,my,mk
  IF mx<>oldmx OR my<>oldmy
    CLS
    oldmx=mx
    oldmy=my
    x=120
    y=100
    w=75
    h=75
    BOX x,y,x+w,y+h
    BOX mx,my,mx+50,my+50
    PRINT RC_INTERSECT(mx,my,50,50,x,y,w,h)
    PBOX x,y,x+w,y+h
  ENDIF
LOOP
```

--> Two boxes are drawn on the screen, one of which can be moved about with the mouse. The rectangle supplied by the function RC_INTERSECT is shown in black. The value -1 (TRUE), or 0 (FALSE) is shown in the top left corner of the screen, depending on whether or not the moving rectangle overlaps the fixed one.

RC_COPY s_adr,sx,sy,w,h TO d_adr,dx,dy [,m]

s_adr,d_adr,sx,sy,w,h,dx,dy,m : iexp

The instruction RC_COPY makes possible the copying of rectangular 'screen' sections between areas of memory, each of which represents a screen display which may be sent to the monitor by specifying the screen address appropriately (see XBIOS(5)). The parameters s_adr and d_adr contain the starting addresses of the source and destination screens. The coordinates of the top left corner and the width and height of the rectangle to be copied should be specified in sx, sy, w and h. The coordinates of the top left corner of the destination rectangle are dx and dy. An optional logical operation may be performed between the source and destination rectangles, given by 'm' in the range 0 to 15 (see PUT). The default value for m is 3 (replacement mode).

Example:

```
FOR r=1 TO 400
  CIRCLE 320,200,r
NEXT r
SGET pic$
s_adr%=V:pic$
d_adr%=XBIOS(2)
'
FOR i%=1 TO 1000
   RC_COPY  s_adr%,RAND(10)*64,RAND(10)*40,64,40  TO
d_adr%,RAND(10)*64,RAND(10)*40
NEXT i%
```

-> A simple picture is drawn and stored in pic$. RC_COPY then copies random 'screen' sections from pic$ in memory to the current screen memory (given by XBIOS(2)).

ALERT sym,text$,default,button$,choice

sym,default : iexp
text$,button$: sexp
choice : avar

The command ALERT creates an Alert box. The expression sym determines which symbol is to appear in the box. The following are valid:

0 --> No symbol
1 --> Exclamation mark
2 --> Question mark
3 --> Stop sign

The text to appear in the box is given by the expression text$. A maximum of 4 lines is permitted, with a maximum of 30 characters in each line, separated by 'rule' characters (|). Lines which are longer than 30 characters are truncated.

The expression 'default' specifies which of the box's buttons is to be the one with a bold border, selectable by pressing Return or Enter. As no more than three buttons are allowed, 'default' can take a value between 0 and 3, 0 indicating that there will be no default button and selection can only be made by clicking with the mouse.

The string expression button$ contains the text for the buttons, with a maximum of 8 characters per button. The individual button legends are separated by 'rule' characters (|).

On exiting from the Alert box, the variable 'choice' contains the number (1 to 3) of the selected button. (See: FORM_ALERT).

Example:

```
ALERT 1,"Pick albutton",1,"Left|Right",a%
ALERT 0, "You pressed| Button "+STR$(a%),0,"Ok",a%
```

--> A box with two buttons appears. After the selection has been made,a second box materialises stating which button in the first box was chosen. This second Alert box has no symbol and no Default button.

FILESELECT #title$, path$, default$, name$

title$,path$,default$: sexp
name$: svar

This instruction causes a File-Select box to be created on the screen, enabling the user-friendly selection of a filename.

The expression title$ can be a maximum of 30 characters and allows a header to be placed in the File-Select box - available from TOS 1.4 onwards. In the expression path$ the initial drive- and path-name should be specified. If no drive is specified, then the current drive is assumed. default$ contains the name of the file which will appear as the current choice. This can be selected by pressing Return, edited or deleted (by pressing ESCape). After exiting from the Box, the name of the selected file will be found in the string name$. If the Cancel button was selected, then name$ will contain a null string ("").

The formats of path$, default$ and name$ conform to the conventions of the Hierarchical Filing system, described in the chapter 'General Input and Output', section 'File Management'.

(See also FSEL_INPUT)

Example:

```
DO
  FILESELECT "a:\*.PRG","GFABASIC.PRG",name$
  IF name$=""
    PRINT "You clicked the Cancel button"
  ELSE IF RIGHT$(name$)="\"
    PRINT "You clicked the OK button without naming
                                                a file"
  ELSE
    PRINT "You have selected the file: ";name$
  ENDIF
LOOP
```

--> A Fileselect Box appears and responds to the choice made by the user.

CHAPTER 10
SYSTEM ROUTINES

GEMDOS, BIOS and XBIOS

GEMDOS(n [,x,y...])
BIOS(n [,x,y...])
XBIOS(n [,x,y...])

n,x,y : iexp

These three functions are used to call GEMDOS, BIOS and XBIOS routines, in each case the optional parameter list is passed to function number n. In order to be able to pass the parameters in the correct variable size, they can be prefixed with W: or L:, to denote word (16-bit) and longword (32-bit) sized values respectively.

The number and meaning of the parameters depends, as does the returned value, on the system routine being called. (See also Appendix.)

Examples:

```
IF GEMDOS(17)
  PRINT "Printer ready"
ELSE
  PRINT "Printer not ready"
ENDIF
```

Checks if a printer is ready to receive data through the parallel interface and reports the status as a logical (TRUE or FALSE) value.

```
REPEAT
UNTIL BIOS(11,-1) AND 4
```

Waits until the Control key is pressed. (BIOS(11,-1) reports on the current status of the keyboard 'shift' keys):

Bit	Key
0	Right shift
1	Left shift
2	Control
3	Alternate
4	Caps Lock

```
CIRCLE 320,200,180
PAUSE 30
BMOVE XBIOS(2),XBIOS(2)+16000,16000
```

--> Draws a circle and copies its upper half to the lower part of the screen. (XBIOS(2) returns the address at which the physical screen memory begins).

L:x
W:x

x : iexp

These two functions enable the user to pass numerical expressions to Operating System functions and C-routines as either a word (2-byte, W:) or longword (4-byte, L:). By default the word format is used.

Example:

```
DIM screen_2|(32255)
phys_base%=XBIOS(2)
old_screen%=phys_base%
log_base%=V:screen_2|(0)+255 AND &HFFFFFF00
~XBIOS(5,L:log_base%,L:phys_base%,-1)
SWAP log_base%,phys_base%
REPEAT
  IF MOUSEK=1
    ~XBIOS(5,L:log_base%,L:phys_base%,-1)
    SWAP log_base%,phys_base%
    REPEAT
    UNTIL MOUSEK=0
  ENDIF
  PLOT MOUSEX,MOUSEY
UNTIL MOUSEK=2
~XBIOS(5,L:old_screen%,L:old_screen%,-1)
```

--> XBIOS(5) is used to switch between two display screens each time the left mouse button is pressed. Moving the mouse causes points to be plotted on the screen which is NOT currently visible. Pressing the right mouse button terminates the program.

LINE-A Calls

ACLIP,PSET,PTST,ALINE,HLINE ARECT,APOLY,ACHAR,ATEXT

In the following section a group of instructions are discussed which correspond in principle to instructions that have already been presented in the graphics chapter. The output is substantially quicker, however, and a slightly different syntax is used. For Line-A graphics, clipping should always be switched on (with ACLIP), since areas of memory can be overwritten by a line leaving the screen area, for example. Otherwise the clipping area is that set by the last used graphic command (AES or VDI), for example OPENW, FILESELECT, ALERT, etc.

A VDI call changes the ACLIP setting previously set with ACLIP. Line-A calls are independent of the VDI DEFxxx commands.

NOTE!!!: The colours specified for the LINE A routines correspond to the hardware colour register numbers (as used by SETCOLOR) and NOT to the numbers used by the VDI (COLOR). They convert as follows:

COLOR No:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LINE-A colour No.	0	15	1	2	4	6	3	5	7	8	9	10	12	14	11	13

LINE A etc.

colour No:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
COLOR No:	0	2	3	6	4	7	5	8	9	10	11	14	12	15	13	1

ACLIP flag,xmin,ymin,xmax,ymax

flag,xmin,ymin,xmax,ymax : iexp

This instruction makes it possible to define a 'clipping' rectangle, to which LINE-A screen output will be limited. The coordinates of the top left and bottom right corners of the Clipping rectangle are given by xmin, ymin, xmax and ymax. If 'flag' is given a non-zero value the clipping is active, otherwise (if flag=0) it is switched off. ACLIP is not valid (unfortunately) for PSET, PTST, ALINE, HLINE and BITBLT.

PSET x,y,f

x,y,f : iexp

PSET corresponds to the PLOT command, and will set the point x,y to colour f, which can take values from 0 to 15, depending on the current screen resolution.

Example:

```
FOR x%=0 to 15
  FOR y%=0 TO 100 STEP 2
    PSET x%,y%,x%
  NEXT y%
NEXT x%
```

--> Draws vertical dotted lines in the 15 colours.

PTST(x,y)

x,y : iexp

The function PTST corresponds to the function POINT(). It returns the colour of the pixel at screen position x,y.

Example:

```
PSET 100,100,6
x=PTST(200,100)
PRINT x,PTST(100,100)
```

--> Prints the colours of the pixels at screen positions (200,100) and (100,100).

ALINE x1,y1,x2,y2,f,lm,m

x1,y1,x2,y2,f,ls,m : iexp

ALINE corresponds to the command LINE, where (x1,y1) and (x2,y2) are the coordinates of the end points of the line. The expression f contains the colour, which, depending on the current screen resolution, takes values from 0 to 15. 'ls' contains 16 bits of information for the desired style of line (solid, dashed, dotted, etc.). Each set bit corresponds to one point to be plotted.

The parameter m determines the graphic mode and can take values from 0 to 3:

m	mode
0	Replace
1	Transparent
2	Inverting
3	Inverted transparent

Example:

```
ym%=INT{L~A-4}-1
FOR i%=0 TO 255
style%=256*i%+i%
  ALINE i%,0,i%,ym%,1,style%,0
NEXT i%
```

--> Draws vertical lines of varying dottedness extending from the top to the bottom of the screen. The term INT{L~A-4} returns the maximum y coordinate of the screen, this being stored four bytes before the beginning of the Line-A Parameter Table (which starts at the address L~A).

HLINE x1,y,x2,f,m,addr,num_pattern

x1,x2,y,f,m,addr,num_pattern : iexp

HLINE is similar to the ALINE command, but only horizontal lines can be drawn. x1 and x2 contain the x-coordinates of the line end points, and y the common y-coordinate. The expression f contains the colour which, depending on the current screen resolution, take values from 0 to 15. The parameter m determines the graphic mode as with ALINE.

Addr is the address of a block of memory which contains bit information for several line styles each of 16 bits. Which style is used for a given line depends on both the y-coordinate and the parameter num_pattern. They are ANDed together and the resulting number used as an index to the style table. Thus num_pattern should generally be one smaller than a power of two (0,1,3,7,15 etc.), giving the effect that, with num_pattern=7, the first eight styles will be used sequentially as the y-coordinate moves down the screen. If num_pattern=3, one of the first four styles will be used, depending on the y-coordinate.

Example:

```
ACLIP 1,0,0,639,399
'
pattern%=&X11111111111111111010101010101010
z%=V:pattern%
'
FOR i%=0 TO 199
  HLINE 0,i%,639,1,0,z%,1
NEXT i%
```

--> Two 16-bit line patterns are put into the variable pattern%. The last parameter is 1 (implying 2 possible styles). The lines drawn then use the two 16 bit line patterns in pattern% alternately.

ARECT x1,y1,x2,y2,f,m,addr,num_pattern

x1,y1,x2,y2,f,m,addr,num_pattern : iexp

ARECT corresponds to PBOX. (x1,y1) and (x2,y2) are the coordinates of two opposite corners of the rectangle. The parameters f,m,addr and num_pattern have the same meaning as for HLINE.

Example:

```
ACLIP 1,0,0,639,399
DIM pattern&(1)
'
pattern&(0)=-21846
pattern&(1)=21845
'
pattern_adr%=V: pattern&(0)
ARECT 100,100,200,200,1,0,pattern_adr%,1
```

--> A rectangle filled with a chequer-board pattern (equivalent to using PBOX after DEFFILL 2,2,4 except that no border is drawn).

APOLY adr_pnt,num_pnt,y0 TO y1,f,m,addr,num_pattern

adr_pnt,num_pnt,y0,y1,f,m,addr,num_pattern : iexp

APOLY is similar to the POLYFILL command. It draws an (invisible) sequence of jointed lines, with 'num_pnt' corners and fills the resulting area with a user defined pattern. 'adr_pnt' is the address of the array which contains alternating x- and y-coordinates of the corner points. The parameter 'num_pnt' contains the number of points. y0 and y1 specify the highest and lowest parts of the screen where filling can take place - it is like a vertical-only clipping facility. The parameters f, m,addr,num_pattern correspond to those at HLINE.

Example:

```
DIM x&(9),pattern&(1)
FOR i%=0 TO 7
  x&(i%)=RAND(100)
NEXT i%
x&(8)=x&(0)
x&(9)=x&(1)
'
adr_corners%=V:x&(0)
pattern&(0)=-1
adr_pattern%=V:pattern&(0)
'
ACLIP 1,0,0,200,200
APOLY adr_corners%,4,0 TO 100,1,1,adr_pattern%,0
```

--> Draws a random filled quadrilateral.

BITBLT adr%
BITBLT x%()

adr%: iexp
x%(): Four byte Integer array

The command BITBLT calls the LINE A-routine of the same name, whereas the command with three parameter arrays calls a VDI-routine. In the command variation for which an address is given as a parameter, a 76 byte long table must start at this address. The meaning of the values in this storage area can be taken from the table that follows. The offset distances of the elements from the start of the table are shown in the column 'offset'.

In the command variation with the one four byte-Integer array, this array must have at least 23 elements, a specified parameter standing in each. Which parameter must stand in which array element can be seen in the column 'index' in the following table.

The parameters marked with a star are affected by the routine, therefore one should normally work with the command variant which uses an array parameter, since as a copy of the array is used by the BITBLT routine, the array itself will not be altered. In contrast, the table elements which were given by the passing of an address will be changed.

Name		Index	Offset	Meaning
B_WD		00	00	Screen width in pixels
B_HT		01	02	Screen height in pixels
PLANE_CT	*	02	04	Number of bit-planes
FG_COL	*	03	06	Foreground colour
BG_COL	*	04	08	Background colour
OP_TAB		05	10	Logical 'put' mode 0 to 15 (see PUT)
S_XMIN		06	14	x offset for source
S_YMIN		07	16	y offset for source
S_FOR		08	18	Address of the source screen
S_NXWD		09	22	Offset to the next word of the same bit-plane
S_NXLN		10	24	Offset to the next line of the source screen
S_NXPL		11	26	Offset to the next bit-plane (always 2)
D_XMIN		12	28	x offset for destination
D_YMIN		13	30	y offset for destination
D_FORM		14	32	Address of the destination screen
D_XNWD		15	36	Offset to the next word of the same bit-plane
D_NXLN		16	38	Offset to the next line of the destination screen
D_NXPL		17	40	Offset to the next bit-plane (always 2)
P_ADDR		18	42	Pointer to fill pattern table (0=no pattern) (Source ANDed with pattern before moving)
P_NXLN		19	46	Offset to the next line of the mask
P_NXPL		20	48	Offset to the next plane of the pattern
P_MASK		21	50	Mask as for HLINE
SPACE	*	22	52	Next 24 bytes are workspace for the Blitter

Examples:

```
DIM x%(1000)
'
FOR i%=0 TO 639 STEP 8
  LINE i%,0,639,399
NEXT i%
'
GET 0,0,639,399,a$
mirrorput(0,0,a$)
'
PROCEDURE mirrorput(x%,y%,VAR x$)
  IF LEN(x$)>6  ! Only do it if something present
    xx%=V:x%(0)
    a%=V:x$
    b%=INT{a%}
    h%=INT{a%+2}
    '
    INT{xx%}=1
    INT{xx%+2}=h%
    INT{xx%+4}=1
    INT{xx%+6}=1
    INT{xx%+8}=0
    {xx%+10}=&H3030303
    INT{xx%+14}=9999
    INT{xx%+16}=0
    {xx%+18}=a%+6
    INT{xx%+22}=2
    INT{xx%+24}=SHR(b%+16,4)*2
    INT{xx%+26}=2
    INT{xx%+28}=9999
    INT{xx%+30}=0
    {xx%+32}=XBIOS(3)
    INT{xx%+36}=2
    INT{xx%+38}=80
    INT{xx%+40}=2
    {xx%+42}=0    ! pattadr
    INT{xx%+46}=0 ! p_nxtln
```

```
    INT{xx%+48}=0 ! p_nxpl
    INT{xx%+50}=0 ! p_mask
    '
    ABSOLUTE i&,xx%+14
    ABSOLUTE di&,xx%+28
    '
    FOR i&=0 TO b%
      INT{xx%+4}=1
      di&=SUB(639,i&)
      BITBLT xx%
    NEXT i&
    '
  ENDIF
RETURN
```

--> An example of BITBLT adr%.

```
DIM x%(1000)
'
FOR i%=0 TO 639 STEPS 8
  LINE i%,0,639,399
NEXT i%
'
GET 0,0,639,399,a$
mirrorput(0,0,a$)
'
PROCEDURE mirrorput(x%,y%,VAR x$)
  IF LEN(x$)>6      ! Only do if something present
    a%=V:x$
    b%=INT{a%}
    h%=INT{a%+2}
    '
    x%(0)=1
    x%(1)=h%
    x%(2)=1
    x%(3)=1
    x%(4)=0
    x%(5)=&H3030303
```

```
    x%(6)=9999
    x%(7)=0
    x%(8)=V:x$+6
    x%(9)=2
    x%(10)=SHR(b%+16,4)*2
    x%(11)=2
    x%(12)=9999
    x%(13)=0
    x%(14)=XBIOS(3)
    x%(15)=2
    x%(16)=80
    x%(17)=2
    x%(18)=0 ! pattadr
    x%(19)=0 ! p_nxtln
    x%(20)=0 ! p_nxpl
    x%(21)=0 ! p_mask
    '
    FOR i%=0 TO b%
      x%(6)=i%
      x%(12)=639-i%
      BITBLT x%()
    NEXT i%
    '
  ENDIF
RETURN
```

--> An example of BITBLT x%().

Both routines reflect a picture about a vertical line down the centre of the screen. In the routine which uses BITBLT adr%, the number of bit-planes must be re-specified each time the instruction is used. Which routine to use is more or less a matter of taste, as there is very little difference in speed between the two. A VDI version of the program producing the same effect is listed under BITBLT in the graphics chapter.

ACHAR code,x,y,font,style,angle

code,x,y,font,style,angle : iexp

By means of ACHAR an individual character with the ASCII value 'code' can be displayed at the point with graphics coordinates (x,y). The numerical expression 'font' can accept values between 0 and 2 as follows:

0 = 6x6 ('Icon' font)
1 = 8x8 (Normal font for colour)
2 = 8x16 (Normal monochrome font)

Larger values for 'font' are taken to be Font-header addresses. The font must be present in the format into which it is converted by GDOS on loading, that is in the Motorola format with High-byte before Low-byte.

Text type (bold, faint etc: 0 to 31) and output angle (0,900,1800,2700) can be specified as for TEXT with DEFTEXT. In contrast to these TEXT commands, the x and y coordinates refer to the top left corner of the character, not the bottom left corner.

ATEXT x,y,font,s$

x,y,font : iexp
s$: sexp

The command ATEXT outputs character strings at arbitrary screen positions but, unlike with ACHAR, no character style or output angle can be specified. The parameters x,y and font are the same as for ACHAR.

Example:

```
OPENW 0                ! Protect top line from overwriting
EVERY 400 GOSUB a_clock  ! Every 2 seconds call a_clock
FOR i%=1 TO 100000
  PRINT USING "######",i%; ! Display numbers
NEXT i%
'
PROCEDURE a_clock
  ACLIP 1,20,0,120,15       ! Switch clipping on,
  '      otherwise it will remain as set by OPENW 0,
  '                        i.e. nothing will be seen
  ATEXT 20,0,2,TIME$        ! Display the time
RETURN
```

--> Numbers are printed continuously, but every 2 seconds a time display is updated on the top line.

L~A

L~A

The function L~A returns the base address of the LINE-A variables (see appendix).

Example:

```
PRINT INT{L~A}
```

--> Gives the number of bit-planes in the current resolution. See also the examples under BITBLT.

VDI Routines

The VDI functions are divided into seven categories:

Control functions
Output functions
Attribute functions
Raster functions
Input functions
Inquire functions
Escape functions

Three kinds of parameters are distinguished: Input, Output and those used for both input and output. These parameters belong in five arrays:

CONTRL	Control
INTIN	Integer input
PTSIN	Point coordinate input
INTOUT	Integer output
PTSOUT	Point coordinate output

The input parameters are stored in:

CONTRL(0)	'Open workstation' opcode
CONTRL(1)	Number of points in PTSIN array
CONTRL(3)	Length of the INTIN array
CONTRL(5)	Device handle
INTIN()	Integer value input array
PTSIN()	Point coordinates input array

The output parameters are:

CONTRL(2) Number of the points in the array
CONTRL(4) Length of the INTOUTarrays
INTOUT() Output array of the integer values
PTSOUT() Output array of the point coordinates

Input and output parameters are:

CONTRL(6) Device identification
CONTRL(7-n) Opcode dependent information

The array elements meaning depends on which VDI function is being called.

CONTRL
INTIN
PTSIN
INTOUT
PTSOUT

These functions give the addresses in the VDI Parameter Block, which are the addresses of the first bytes of the CONTRL, INTIN etc. arrays. The arrays can be accessed by placing an index value in brackets after the array name, for example CONTRL(2)=x& corresponds to DPOKE CONTRL+4,x& and x&=CONTRL(4) to x&=DPEEK(CONTRL+4). The other arrays are also organised in word format.

The meanings are:

CONTRL	-->	address of the VDI control table
INTIN	-->	address of the VDI integer input table
PTSIN	-->	address of the VDI point coordinate input table
INTOUT	-->	address of the VDI integer output table
PTSOUT	-->	address of the VDI point coordinate output table

These tables (arrays) contain the parameters for VDI-calls.

VDISYS [opcode [,c_int,c_pts [,subopc]]]

opcode,c_int,c_pts,subopc : iexp

With the instruction VDISYS the VDI function with the function code 'opcode' is called. If opcode is not specified, then the function code must, like the other parameters, be placed in the control block with 'DPOKE CONTRL,opcode' or 'CONTRL(0)=opcode'.

The number of values in the integer and point-coordinate input arrays can be specified in the parameters c_int and c_pts. These values need not then be placed into the Control Block. The optional parameter 'subopc' contains the sub-opcode of the routine to be called. This must only be specified for some VDI routines, for example Escape functions routines. The parameters c_int, c_pts and subopc will be inserted into the CONTRL block like so:

opc	-->	CONTRL(0)
c_int	-->	CONTRL(3)
c_pts	-->	CONTRL(1)
subopc	-->	CONTRL(5)

Examples:

```
CONTRL(1)=3
CONTRL(5)=4
PTSIN(0)=320
PTSIN(1)=200
PTSIN(4)=190
VDISYS 11
PAUSE 25
CLS
'
PTSIN(0)=320
PTSIN(1)=200
PTSIN(4)=190
VDISYS 11,0,3,4
PAUSE 25
CLS
'
PCIRCLE 320,200,190
```

--> Three different ways of drawing the same filled circle.

```
VDISYS 5,0,0,13
PRINT "Inverted"
VDISYS 5,0,0,14
PRINT "Normal"
```

--> Prints 'Inverted' in inverse video and 'Normal' normally.

VDIBASE

The system variable VDIBASE contains the address starting from which the current GEM version puts parameters for use in the VDI routines (text style, clipping etc). The structure of this area could well be different in future versions of GEM. VDIBASE is contained as a keyword in GFA BASIC in order to give programmers the ability to access all VDI parameters.

WORK_OUT(x)

x : iexp

This function determines the values found in INTOUT(0) to INTOUT(44), PTSOUT(0) and PTSOUT(1) after returning from the function OPEN_WORKSTATION. The index 'x' can take values from 0 to 56 (see Appendix).

Example:

```
screen_width&=WORK_OUT(0)
screen_height&=WORK_OUT(1)
PRINT screen_width&,screen_height&,WORK_OUT(10)
```

--> The numbers 639 and 399 (in monochrome mode) for the screen width and height are printed, followed by a 1 for the number of the available character sets (WORK_OUT(10)).

Special VDI Routines and GDOS

The following VDI Workstation and query functions are available only if GDOS (releases 1.0 and after) has been booted and a valid ASSIGN.SYS file is available.

The machine independent GDOS (Graphics DEVICE Operating System) contains graphic functions and works together with device dependent drivers for different output devices (screen, printer, plotter, Metafiles, etc.).

The ASCII file ASSIGN.SYS contains all necessary data concerning the current device configuration. In this file all current Operating System and character sets must be registered, and if necessary the access path to the Operating System must be specified, if it is not on drive A:. The following syntax should be noted:

```
id      DEVICE.SYS;
;       Remarks
        part1.FNT
        part2.FNT
        ...
        partn.FNT
```

id contains a number between 1 and 32767. With it the type of device is determined as follows:

01 ... Screen
11 ... Plotter
21 ... Printer
31 ... Metafile
41 ... Camera
51 ... Graphic-Tablet

If the Operating System is located on drive C: in the file GEMSYS, then the associated ASSIGN.SYS file looks, for example, like this:

```
path = c:\gemsys\
01p screen.sys;  Screen driver in ROM, therefore
                              a 'p' after the ID
ATSS10.FNT
ATSS12.FNT
02p screen.sys;  Driver for low resolution
ATSS10.FNT
ATSS12.FNT
03p screen.sys;  Driver for medium resolution
ATSS10CG.FNT
ATSS12CG.FNT
04p screen.sys;  Driver for high resolution
ATSS10.FNT
ATSS12.FNT
21 FX80.sys;     Printer driver for the FX-80
                                  and compatible
ATSS10EP.FNT
ATSS12EP.FNT
31 META.SYS;     Metafile driver
ATSS10MF.FNT
ATSS12MF.FNT
```

The screen-drivers SCREEN.SYS are only dummy entries, required by GDOS' syntax checker. The ID's (2,3 and 4) have only to assign, for each resolution, the order of the given fonts. Therefore the call to open a virtual screen memory workstation is V_OPNVWK(XBIOS(4)+2). GFA BASIC V3 makes this call internally.

GDOS?

GDOS? supplies TRUE if GDOS (release 1.0 or after) is resident and FALSE otherwise.

Example:

```
IF NOT GDOS?
  ALERT 1, "GDOS not found.",1," OK ",r%
  END
ENDIF
```

V~H

With the following VDI functions, 'hd' stands for the VDI handle and 'id' for the identification of an output device. With this in mind, consider the following functions:

V~H Returns the internal VDI handle of GFA BASIC. (e.g. PRINT V~H)

V~H=x Sets the internal VDI handle (accessible through CONTRL(6)) to the value x.

V~H=-1 Sets the VDI handle (CONTRL(6)) to the value resulting from the internal V_OPNVWK.

V_OPNWK(id)
V_OPNWK(id,1,1,1,1,1,1,1,1,1,2)
V_CLSWK()

ID : iexp

The numbers 1 and 2 are default values for the settings of the VDI parameters. They can be changed (cf. WORK_OUT).

The function V_OPNWK (Open Workstation) supplies the handle hd for the specified device 'id'. In addition, further information about the device may be requested by means of INTOUT() and PTSOUT() (c.f. VDISYS in the section 'System Routines').

The function V_CLSWK (Close Workstation) closes the current workstation opened with V_OPNWK and flushes its buffers. In addition a V~H=-1 is carried out.

Example: See V_CLRWK.

V_OPNVWK(id)
V_OPNVWK(id,1,1,1,1,1,1,1,1,1,2)
V_CLSVWK(id)

id : iexp

The function V_OPNVWK (Open Virtual Workstation) opens a virtual screen driver and supplies the handle for the specified device id (cf. V_OPNWK).

The function V_CLSVWK (Close Virtual Workstation) closes a virtual Workstation opened with V_OPNVWK. In addition V~H=-1 is issued.

V_CLRWK()
V_UPDWK()

The function V_CLRWK (Clear workstation) clears the output buffer. For example, the screen or the printer buffer is cleared.

For output of graphics to a printer, all commands are collected in a buffer. The function V_UPDWK (update workstation) sends these buffered graphic instructions to the attached device. Unlike printer graphics, for example, all graphic instructions sent to the screen are implemented immediately.

Example (if GDOS is resident):

```
RESERVE 25600    ! Reserve sufficient storage memory
handle%=V_OPNWK(21)   ! Determine identification
'                     ! for output device
'
IF handle%=0
  ALERT 3,"Installation error!",1,"Cancel",r%
  RESERVE
  END
ENDIF
'
x_res%=INTOUT(0)      ! Determine x and y-resolution
y_res%=INTOUT(1)      ! of the attached device
'
V~H=handle%           ! Sets the internal VDI-handle
'                     ! to printer identification
~V_CLRWK()            ! Clear buffer
'
CLIP 0,0,x_res%,y_res%
BOX 0,0,x_res%,y_res%
LINE 0,0,x_res%,y_res%
LINE 0,y_res%,x_res%,0
'
~V_UPDWK()            ! Carry out Graphic commands
~V_CLSWK()
'
RESERVE               ! Return memory to GFA BASIC
```

--> Sends a rectangle with a diagonal line through it to an attached printer, if GDOS is resident.

VST_LOAD_FONTS(x)
VST_UNLOAD_FONTS(x)

x : iexp

The function VST_LOAD_FONTS loads the additional character sets specified in ASSIGN.SYS, if sufficient memory is available, and the number of loaded fonts is returned. If no further fonts are available, zero is returned.

The parameter x is should be zero with present versions of GEM, though this may change in the future. It is important to reserve sufficient memory for the additional character sets, using RESERVE.

The function VST_UNLOAD_FONTS removes the character sets loaded previously with VST_LOAD_FONTS from memory. The parameter x is currently zero as for VST_LOAD_FONTS.

Example:

```
RESERVE 25600
'
num_fonts%=VST_LOAD_FONTS(0)    ! How many additional
'                               ! character sets?
face%=VQT_NAME(num_fonts%,font$) ! Index and name
'                                ! of loaded Fonts
'
FOR i%=1 TO num_fonts%
  DEFTEXT,,,,face%
  TEXT 80,80,"This is the "+font$+" font."
  ~INP(2)
NEXT i%
'
~VST_UNLOAD_FONTS(0)             ! Remove fonts
'
RESERVE
```

--> Displays the names of the loaded character sets in their own fonts.

VQT_EXTENT(text$ [,x1,y1,x2,y2,x3,y3,x4,y4])

text$: **sexp**
x1,y1,x2,y2,x3,y3,x4,y4 : **ivar**

The function VQT_EXTENT returns the corner coordinates of a rectangle which will surround the text in text$. The coordinates can either be found in the variables x1,y1 to x4,y4, or in PTSOUT(0) to PTSOUT(7). The corner points are numbered in the clockwise direction:

Corner point	Position
x1,y1	Top left
x2,y2	Top right
x3,y3	Bottom right
x4,y4	Bottom left

Example:

```
INPUT text$
CLS
ATEXT 100,25,2,text$
~VQT_EXTENT(text$,x1,y1,x2,y2,x3,y3,x4,y4)
BOX x4+100,y4+25,x2+100,y2+25
```

--> The input text is put at the arbitrary screen position (100,25), and a box is neatly put round it.

VQT_NAME(i,font_name$)

i : **ivar**
font_name$: **svar**

The function VQT_NAME supplies the handle of the font with the identification number 'i' and places the name of the loaded character set into the string variable font_name$.

Example:

```
RESERVE 25600
'
num_fonts%=VST_LOAD_FONTS(0)
face%=VQT_NAME(num_fonts%, font$) ! Index and name
'                                 ! of loaded Fonts
h%=12                             ! Text height
s$="example text"
x0%=80                          ! Output coordinates
y0%=80                            !  for s$
DEFTEXT 1,0,0,h%,face%
'
~VQT_EXTENT(s$,x1%,y1%,x2%,y2%,x3%,y3%,x4%,y4%)
'
GRAPHMODE 4
TEXT x0%,y0%,s$
PBOX x0%+x1%,y0%+y1%-h%-1,x0%+x3%,y0%+y3%-h%
'
~VST_UNLOAD_FONTS(0)              ! Remove fonts
'
RESERVE
```

--> The string s$ is displayed inverted on the screen at (x0%,y0%).

Non-BASIC Routine Calls

The commands explained in this section enable subroutines which are written in C or assembler to be called.

C: addr([x,y,...])

addr : avar (at least 32 bit, ideally integer-type: adr%)
x,y : iexp

The function C: calls a C or assembler subroutine, located at the address addr. The parameters in brackets (x, y...) will be passed to the routine. The parameter passing is via the stack, as in C. The parameters can be sent as 32-bit longwords with the prefix 'L:',or 16-bit words with the prefix 'W:'. If there is no prefix, a word value will be sent. When this function is called, first the return address and then the parameters will be found on the stack.

So, for example:

VOID C:adr%(L:x,W:y,z) leads to the following situation on the stack:

(sp)	-->	Return address (4 bytes)
4(sp)	-->	x (4 bytes)
8(sp)	-->	y (2 bytes)
10(sp)	-->	z (2 bytes)

The value returned by the function is the contents of register d0 on return from the subroutine (for which RTS must be used).

Example:

The assembler program used here fills an area of memory (e.g. an array) starting from a certain address with the numbers (longwords) from 0 to n.

```
206F0004      move.l  4(sp),a0  ; Start address
202F0008      move.l  8(sp),d0  ; Number of values
7200          moveq.l #0,d1     ; Counter
6004          bra.s   ct_2      ; Loop re-entry point
20C1    ct_1:move.l   d1,(a0)+  ; Loop, value write
5281          addq.l  #1,d1     ; Increment counter
B081    ct_2:cmp.l    d1,d0      ; finished?
64F8          bcc.s   ct_1      ; no, around again ->
4E75              rts           ; Return to GFA BASIC
```

The GFA BASIC program is:

```
FOR i%=1 TO 11
  READ a%
  asm$=asm$+MKI$(a%)
NEXT i%
DATA $206F,$0004,$202F,$0008,$7200,$6004
DATA $20C1,$5281,$B081,$64F8,$4E75
'
DIM x%(10000)
asm%=V:asm$
~C:asm%(L:V:x%(0),L:10000)
PRINT "E.g. x%(12) = ";x%(12)
```

--> The array x%() is filled with the numbers from 0 to 10000, corresponding to:

```
FOR i%=1 TO n%
  x%(i%)=i%
NEXT i%
```

To insert an assembler program in a GFA BASIC V3 program, the INLINE command can also be used. First a file is created containing the assembler code above by using (after the above read-data loop):

```
BSAVE "COUNT.INL",V:asm$,22
```

Then type in the following program:

```
INLINE asm%,22
DIM x%(10000)
~C:asm%(L:V:x%(0),L:10000)
```

Then, while still in the Editor with the cursor on the line containing the INLINE instruction, press the Help key. From the resulting menu choose 'Load', and from the Fileselect box select the file COUNT.INL. It will be loaded into an area specially allocated within the program, and will be saved with the program when it is Saved in the normal way. See INLINE, Chapter 2, section: Memory Management.

MONITOR [x]

x : iexp

This instruction enables one to call assembler subroutines or debugging programs or other utilities. For this purpose the 'illegal instruction vector' (address 16) must be set to the address of the subroutine. The MONITOR instruction then produces an illegal Instruction exception, which causes the computer to branch to the subroutine. This subroutine must end with RTE (Return from Exception). The parameter x is passed to the subroutine in register d0.

An example of its use is with tracing a C program, using a debugger. GFA BASIC should be loaded and started from the debugger. Then, after inputting the C program, insert the line 'MONITOR asm%' directly before the INLINE statement. After the start of the C program the debugger will respond because of the 'Illegal Instruction exception' generated by the MONITOR command. Now one can disassemble, single-step or edit the C program at the address now in register d0. When the C program has been edited/examined satisfactorily, one can continue the execution of the program with the 'GO' instruction of the debugger. If the 'Break' keys Shift-Control-Alternate are pressed whilst in the debugger, a flag will be set which will cause the BASIC program to halt as soon as control is returned to it.

CALL addr([x,y,...])

addr: avar (at least 32 bit, ideally integer-type: adr%)
x,y : iexp, sexp

Assembler or C subroutines can also be called with the instruction CALL, where 'addr' is the address of the assembler program, which must end with an RTS instruction. It is possible to pass a parameter list to the routine. When the CALL is made, the return address is found at the top of the stack followed by the number of parameters given as a 16-bit word, and finally the address, as a 32-bit word, of the parameter list. All parameters will be put into memory as long (32-bit) words.

Strings may also be passed to the subroutine, in which case it will be the address of the string which is found in the parameter list.

Stack structure

(sp)	-->	Return address
4(sp)	-->	Number of parameters (16 bit)
6(sp)	-->	Address of the parameters (32 bit)

RCALL addr,reg%()

reg%() : Name of Integer (4-byte) array
addr : iexp

The instruction RCALL enables the assembler routine to start with pre-allocated values in the registers, and the BASIC program to query the register contents when the routine returns. The integer-sized array reg%(), which must have at least 16 elements, serves this purpose. Before the assembler routine is started, the entries in this array are automatically copied into the appropriate registers. At the end of the routine the contents of the registers are written back into the appropriate array elements. The registers and array elements are related as follows (assuming OPTION BASE 0):

Data registers	d0 to d7	<-->	reg%(0)	to reg%(7)
Address registers	a0 to a6	<-->	reg%(8)	to reg%(14)
User-stack-Pointer	(a7)	-->	reg%(15)	(return only)

Example:

The assembler listing below expects the address (logical or physical) of the screen memory in a0. It then inverts the screen display between given y coordinates, which are passed in d0 and d1. With a colour display, the y coordinates will have to be adjusted accordingly.

```
sub   d0,d1    ; Number of lines to invert
mulu  #20,d1   ; Number of bytes to invert
subq  #1,d1    ; Number of bytes to invert
mulu  #80,d0   ; Number of bytes not to be inverted
add.l d0,a0    ; Address of first byte to be inverted
loop:          ; Loop start
not.l (a0)+    ; Long (4-byte) inversion, also a0=a0+4
dbra  d1,loop  ; Decrement byte count and loop round
rts            ; Return to GFA BASIC
```

The GFA BASIC program to call this routine reads:

```
DO
  READ a%
  EXIT IF a%=-1
  A$=A$+MKI$(a%)
LOOP
DATA 37440,49916,20,21313
DATA 49404,80,53696
DATA 18072,20937,65532,20085, -1
'
DIM r%(16)
xb2%=XBIOS(2)
HIDEM
FOR j%=1 TO 50
  FOR i%=0 TO 190 STEP 10
    r%(0)=i%
    r%(1)=399-i%
    r%(8)=xb2%
    RCALL V:a$,r%()
  NEXT i%
NEXT j%
```

--> The program produces a graphic display. It is also possible, as for C:, to use the INLINE command.

EXEC mod,nam,cmdl,envs
EXEC(mod,nam,cmdl,envs)

mod : **iexp**
nam,cmdl,envs : **sexp**

The EXEC can be used as a command or as a function. It enables the loading and starting of programs from disk, which return to the calling program after completion. Before the EXEC call, the calling program must have allocated enough memory space for the program (see example). The parameter 'mod' specifies the 'Call mode' as follows:

0 --> Load and Start program.
3 --> Load program only.

The string expression 'nam' contains the filename of the program to be loaded (and optionally started). The format of the filename conforms to the rules of the hierarchical filing system, described in Chapter 6 : Input and Output.

The expression 'cmdl' contains the command line, which is registered in the Basepage of the called program. The first character of the command line contains its length (maximum 127). As a rule, this byte is ignored, as if it were zero. Therefore it is normally sufficient to insert a dummy character (e.g. "*").

The string 'envs' contains the Environment. This is a string terminated by CHR$(0). For a C-program, the Environment is a series of strings, each terminated by a CHR$(0), and the whole terminated by two CHR$(0)'s (these are added automatically by GFA BASIC). This Environment is used by many Shell programs in order to store variables and their names, and also by many compilers, in order to determine access paths etc.

If one calls EXEC as a function, then one receives the program's returned value, or, for mod=3, the address of the Basepage of the called program. EXEC 3 is only for special programs which include overlays, debugging programs etc.

Example:

```
FILESELECT "\*.PRG","",f$
RESERVE 100
SHOWM
a%=EXEC(0,f$,"","")
RESERVE
PRINT "Back in GFA BASIC. Returned value=";a%
```

--> A program is loaded and called (if it does not require too much memory) and returns to the interpreter on completion. The return value, which is supplied by the EXEC function, can also be given by GFA BASIC programs by terminating them with QUIT n or SYSTEM n.

Chapter 11
AES-LIBRARIES

This chapter gives an overview of the AES libraries (AES = Application Environment Services). A detailed description of these routines would be beyond the scope of this manual (we refer the reader to the extensive literature already published on GEM), hence information is given in compressed form, as follows:

a) the name of the appropriate library routine.
b) a short description of the function.
c) the complete GFA BASIC 3 syntax for the function call, with the meaning of all the applicable variables explained.

The chapter closes with longer example programs.

Before beginning the descriptions of the functions in the 11 AES Libraries, it is important that the reader becomes familiar with the most important data structures used by the AES. These are the OBJECT, TEDINFO, ICONBLK, BITBLK, USERBLK and PARMBLK structures.

The following AES functions are implemented in a similar way to GFA BASIC 2:

GCONTRL
ADDRIN
ADDROUT
GINTIN
GINTOUT
GB

They represent the addresses of the AES parameter blocks. Specifically:

GCONTRL	-->	address of the AES Control block
ADDRIN	-->	address of the AES Address Input block
ADDROUT	-->	address of the AES Address Output block
GINTIN	-->	address of the AES Integer Input block
GINTOUT	-->	address of the AES Integer Output block
GB	-->	address of the AES Parameter block

With an index in brackets after these functions, the appropriate parameter positions within the block are accessed directly, for instance, x=GCONTRL(3) corresponds to x=DPEEK(GCONTRL+6), or GCONTL(3)=x corresponds to DPOKE GCONTRL+6,x.

GCONTRL, GINTIN and GINTOUT expect words (2 bytes) as their parameters, whereas ADDRIN and ADDROUT expect long words (4 bytes). So ADDRIN(2)=x corresponds to LPOKE ADDRIN+2*4,x.

The GB block address cannot be used with an index, on grounds of compatibility with ST-BASIC; with GINTIN etc. indices are available. For example, the second longword in the GB block ({GB+4}) is the address of the GEM internal global array.

GEMSYS n

n : iexp

The command GEMSYS calls an AES routine by specifying the routine number n. The parameters necessary for the operation of the routine must first be placed in the appropriate AES parameter blocks.

Example:

```
REPEAT
  GINTIN(0)=60
  GINTIN(1)=30
  GINTIN(2)=10
  GINTIN(3)=10
  GINTIN(4)=200
  GINTIN(5)=200
  GEMSYS 72
UNTIL MOUSEK
```

--> A moving rectangle is drawn on the screen, as function number 72 is GRAF_MOVEBOX).

Object Structure

Offset	Contents	Type	Meaning
00	ob_next	word	pointer to the next object
02	ob_head	word	pointer to the first child
04	ob_tail	word	pointer to the last child
06	ob_type	word	type of object.
08	ob_flags	word	object information (see below)
10	ob_state	word	status of the object (see below)
12	ob_spec	long	pointer to further information (see below)
16	ob_x	word	x-position of the object
18	ob_y	word	y-position of the object
20	ob_w	word	width of the object
22	ob_h	word	height of the object

So the memory requirements of an object are these 24 bytes plus further descriptive structures, e.g. TEDINFO or BITBLK structures.

OB_NEXT points to the following object on the same level, or, if it is the last object on that level, to the parent object, or contains -1 if none.

OB_HEAD points to the object's first child, or contains -1 if none.

OB_TAIL points to the object's last child, or contains -1 if none.

The value -1 in this context is also referred to as NIL (Not In List).

Depending on the value in OB_TYPE, OB_SPEC has the address of different data structures, as shown in the following table:

OB_TYPE		OB_SPEC
20	G_BOX	BOX info., see below
21	G_TEXT	Pointer to TEDINFO graphic text
22	G_BOXTEXT	Pointer to TEDINFO text-in-a-box
23	G_IMAGE	Pointer to BITBLK bit image graphic
24	G_USERDEF	Pointer to USERBLK structure.
25	G_IBOX	BOX info., see below.
26	G_BUTTON	Pointer to centred C-string, to go in a box
27	G_BOXCHAR	BOX info., see below.
28	G_STRING	Pointer to C-string menu item structure
29	G_FTEXT	Pointer to TEDINFO editable graphic text
30	G_FBOXTEXT	Pointer to TEDINFO editable text-in-a-box
31	G_ICON	Pointer to ICONBLK structure
32	G_TITLE	Pointer to C-string menu title structure

For G_BOX, G_IBOX and G_BOXCHAR, OB_SPEC contains information concerning character-content, border-type and colour of the appropriate object. The upper 8 bits are only used by G_BOXCHAR and contain the ASCII code of the character to appear in the box.

They contain the following values for the border:

0	=	no border
1 to 128	=	the border extends into the object for between 1 and 128 pixels
-1 to -128	=	the border extends for between 1 and 128 pixels outside the object

The bit allocation for the object colour word is:

```
1111 2222 3444 5555
```

where:

1 = Border colour (0 to 15)
2 = Text colour (0 to 15)
3 = Text mode (0 = transparent, 1 = overwritten)
4 = Fill pattern (0 to 7)
5 = Colour of object interior (0 to 15)

OB_FLAGS	Hex	Bit no.
NORMAL	&H0000	-
SELECTABLE	&H0001	0
DEFAULT	&H0002	1
EXIT	&H0004	2
EDITABLE	&H0008	3
RBUTTON	&H0010	4
LASTOB	&H0020	5
TOUCHEXIT	&H0040	6
HIDETREE	&H0080	7
INDIRECT	&H0100	8

OB_STATE	Hex	Bit no
NORMAL	&H0000	-
SELECTED	&H0001	0
CROSSED	&H0002	1
CHECKED	&H0004	2
DISABLED	&H0008	3
OUTLINED	&H0010	4
SHADOWED	&H0020	5

The structures described in the preceding section are addressed in GFA BASIC 3 with the following syntax (for both reading and writing):

OB_NEXT(tree%,obj&)
OB_HEAD(tree%,obj&)
OB_TAIL(tree%,obj&)

OB_TYPE(tree%,obj&)
OB_FLAGS(tree%,obj&)
OB_STATE(tree%,obj&)
OB_SPEC(tree%,obj&)

OB_X(tree%,obj&)
OB_Y(tree%,obj&)
OB_W(tree%,obj&)
OB_H(tree%,obj&)

..where tree% is the address of the Object Tree and obj& the object number. In addition the address of an individual object can be determined with address=OB_ADR(tree%,obj&).

Text Data Structure (TEDINFO)

Offset	Contents	Type	Meaning
00	te_ptext	long	Pointer to text
04	te_ptmplt	long	Pointer to text mask
08	te_pvalid	long	Pointer to validation string for input
12	te_font	word	Font
14	te_resvd	word	-reserved-
16	te_just	word	Text justification
18	te_colour	word	Colour of the surrounding box
20	te_resvd2	word	-reserved-
22	te_thickness	word	Character thickness
24	te_txtlen	word	Text length
26	te_tmplen	word	Text mask length

Icon Data Structure (ICONBLK)

Offset	Contents	Type	Meaning
00	ib_pmask	long	Pointer to icon mask
04	ib_pdata	long	Pointer to icon data
08	ib_ptext	long	Pointer to icon text
12	ib_char	word	The single character within the icon
14	ib_xchar	word	x-position of the character
16	ib_ychar	word	y-position of the character
18	ib_xicon	word	x-position of the icon
20	ib_yicon	word	y-position of the icon
22	ib_wicon	word	Width of the icon
24	ib_hicon	word	Height of the icon
26	ib_xtext	word	x-position of the text
28	ib_ytext	word	y-position of the text
30	ib_wtext	word	Text width in pixels
32	ib_htext	word	Text height in pixels
34	ib_resvd	word	-reserved-

Bit Image Block Structure (BITBLK)

Offset	Contents	Type	Meaning
00	bi_pdata	long	Pointer to image data
04	bi_wb	word	Width of the image in bytes
06	bi_hl	word	Height of the image in pixels
08	bi_x	word	x-position of the image
10	bi_y	word	y-position of the image
12	bi_colour	word	Image colour

Application Block Structure (USERBLK)

Offset	Contents	Type	Meaning
00	ub_code	long	Pointer to a user assembly language routine to draw the object.
04	up_parm	long	Pointer to a PARMBLK structure.

Parameter Block Structure (PARMBLK)

Offset	Contents	Type	Meaning
00	pb_tree	long	Pointer to object tree
04	pb_obj	word	Object number
06	pr_prevstate	word	Previous status
08	pr_currstate	word	Current status
10	pb_x	word	x-position of object
12	pb_y	word	y-position of object
14	pb_w	word	Width of object
16	pb_h	word	Height of object
18	pb_xc	word	x-position of clipping rectangle
20	pb_yc	word	y-position of clipping rectangle
22	pb_wc	word	Width of the clipping rectangle
24	pb_hc	word	Height of the clipping rectangle
28	pb_parm	long	Parameter from the USERBLK structure

It should be noted that some functions return not only a function value, but also return values in variables made available for that purpose. If the function value is to be ignored, then the function should be called with VOID or its abbreviation '~'. (In the same way as for the INP function: the returned function value may be printed with PRINT INP(2), or ignored with ~INP(2).) When parameters have to indicate addresses, at least 4-byte long variable types must be used, and coordinate entries must take place in variable types of at least 2 bytes (word or greater).

Applications Library

The Applications Library controls the accessing of other AES libraries. The functions APPL_INIT and APPL_EXIT are called automatically by GFA BASIC 3 when a program is started or ended.

APPL_INIT()

The current program is announced as an application. The function APPL_INIT returns an Application ID that acts as the 'handle' of the current GFA BASIC program. This identification number is, for example, important for the installation of additional fonts.

The identification number (ap_id) of the application (program) is returned.

APPL_INIT is used for APPL_READ, APPL_WRITE and MENU_REGISTER.

This function in GFA BASIC is a dummy function, which carries out no operating system call, since APPL_INIT is already implemented by the start of the GFA BASIC interpreter.

APPL_READ(id,len,adr_buffer)

id,len,adr_buffer : iexp

Returns 0 if an error occurred.

With this instruction bytes can be read from an event buffer (message pipe).

id	>	Identification number of the application, from whose buffer reading is to be done.
len	>	Number of bytes to be read.
adr_buffer	>	Address of the buffer.

APPL_WRITE(id,len,adr_buffer)

id,len,adr_buffer : iexp

With APPL_WRITE, bytes are written into an event buffer (message pipe).

Returns 0 if an error occurred.

id	>	Identification number of the application, into whose buffer writing is to be done.
len	>	Number of bytes to be written.
adr_buffer	>	Address of the buffer.

APPL_FIND(fname$)

fname$: sexp

The identification number of another application in the system is determined from its filename, e.g. for information exchange with other current programs.

If no error occurs, the requested ID is returned, otherwise (if the application is not found), -1 is returned.

fname$	>	The 8-character file name (without extension) of the sought-after application. The filename supplied must be exactly 8 characters long, if necessary padded out with spaces, and characters must be in upper case.

Example:

```
PRINT APPL_FIND("CONTROL")
```

--> Prints '65535' (= -1) if the Control desk accessory cannot be found. If it is found, the ID number (e.g. 2) is printed. Besides Accessories, one can also find the ID of GFA BASIC (ap_id=0) and the screen manager "SCRENMGR" (ap_id=1), which is responsible for the menu handling.

APPL_TPLAY(mem,num,speed)
APPL_TRECORD(mem,num)

mem,num,speed : iexp

APPL_TRECORD makes a record of user activities (mouse movement, key presses, etc), and APPL_TPLAY plays it back at a specified speed (speed = 1 to 10000). These functions do not work as specified in the documentation.

With newer ROM versions these functions work approximately as specified, except that instead of 6 bytes per event 8 bytes are used and also the speed factor works differently. Everything else appears reliable, but these functions should be used advisedly. We will not go into them any further here.

APPL_EXIT()

APPL_EXIT informs the system that the program (application) has finished, causing its identification number to be released and made available for other programs.

APPL_EXIT exists in GFA BASIC only as a dummy function, since a QUIT or SYSTEM command accomplishes this automatically.

In case of error, 0 is returned.

Event Library

The Event Library allows a program to react to inputs from the mouse, keyboard, etc, or to time-dependent events.

EVNT_KEYBD()

Waits for a key to be pressed and returns a word-sized value, with the low-order byte containing the ASCII code, and the high-order byte containing the keyboard scan code.

Example:

```
DO
  PRINT HEX$(EVNT_KEYBD(),4)
LOOP
```

--> Prints values corresponding to pressed keys.

EVNT_BUTTON(clicks,mask,state[,mx,my,button,k_state])

clicks,mask,state : iexp
mx,my,button,k_state : ivar

Waits for one or more mouse clicks, e.g. double-click, triple-click, etc.

Returns the number of clicks.

clicks > Maximum allowable clicks
mask > Mask for the desired mouse key:
Bit 0 = 1 : Left button
Bit 1 = 1 : Right button
state > Desired status, in order to terminate the event
Bit allocation as for 'mask'

mx < x-coordinate of mouse pointer when event is terminated
my < y-coordinate of mouse pointer when event is terminated
button < State of mouse buttons when event is terminated,
as for 'mask'.
k_stat < Condition of the keyboard 'shift' keys on event termination:
Bit 0 = Right shift key
Bit 1 = Left shift key
Bit 2 = Control key
Bit 3 = Alternate key

The parameters mx,my,button and k_state are optional, these values can also be found by querying GINTOUT(1) to GINTOUT(4).

Example:

```
DO
  SELECT EVNT_BUTTON(2,1,1,mx%,my%,bu%,kb%)
  CASE 1
    TEXT mx%,my%,"1"
  CASE 2
    TEXT mx%,my%,"2"
  CASE 3
    TEXT mx%,my%,"3"
  ENDSELECT
LOOP UNTIL BTST(kb%,2) ! until CONTROL key also pressed
```

--> Waits for mouse clicks with the left mouse button. The number of clicks appears on the screen at the mouse position. The program is ended by holding down the CONTROL key as well as clicking.

EVNT_MOUSE(flag,mx,my,mw,mh,mcur_x,mcur_y,button,k_state)

flag,mx,my,mw,mh : iexp
mcur_x,mcur_y,button,k_state : ivar

Waits for the mouse pointer to be located inside (or, optionally, outside) a specified rectangular area of the screen.

The returned value is always 1.

flag	>	Presence inside (0) or outside (1) of the area is detected
mx,my	>	Coordinates of top left corner of rectangle
mw	>	Width of rectangle
mh	>	Height of rectangle
mcur_x	<	x-coordinate of the mouse pointer when event occurs
mcur_y	<	y-coordinate of the mouse pointer when event occurs
button	<	Mouse key status when event occurs: Bit 0 = 1 : Right button Bit 1 = 1 : Left button
k_state	<	State of the keyboard 'shift' keys when event occurs: Bit 0 = Right shift key Bit 1 = Left shift key Bit 2 = Control key Bit 3 = Alternate key

Example:

```
DO
  ~EVNT_MOUSE(0,100,100,200,90,mx%,my%,bu%,kb%)
  IF bu%
    PBOX mx%-10,my%-10,mx%+10,my%+10
  ELSE
    PLOT mx%,my%
  ENDIF
LOOP UNTIL BTST(kb%,2)
```

--> The program waits until the mouse pointer is located inside the rectangle. Then a point is plotted at the mouse position, or a small square is drawn if the left button is pressed. The program is ended if the CONTROL key is held down while the pointer is within the rectangle. Compare the effect produced by making the first parameter 1 instead of 0.

Note: While a GEM routine is being executed, the 'break' keys (Control-Shift-Alternate) can be pressed, but will not take effect until execution has returned to the BASIC part of the program. In general though, it is best to exit from programs in the manner provided by the program, otherwise memory restoration, etc, may not take place.

EVNT_MESAG(adr_buffer)

adr_buffer : iexp

Waits for the arrival of a message in the event buffer.

The returned value is always 1.

adr_buffer > address of a 16-byte buffer for the message (cf. MENU(1) to MENU(8)). If 0 is given for adr_buffer, the system message buffer is used, i.e. MENU(1) to MENU(8).

EVNT_TIMER(count)

count : iexp

The function waits for a period of time expressed in 'count' in milliseconds (see DELAY).

The returned value is always 1.

count > Number of milliseconds

EVNT_MULTI(flag,clicks,mask,state,m1_flags,m1_x,m1_y, m1_w,m1_h,m2_flags,m2_x,m2_y,m2_w,m2_h,adr_buffer, count[,mcur_x,mcur_y,button,k_state,key,num_clicks])

flag,clicks,mask,state,m1_flags,m1_x,m1_y,m1_w,m1_h : iexp
m2_x,m2_y,m2_w,m2_h,adr_buffer,count : iexp
mcur_x,mcur_y,button,k_state,key,num_clicks : ivar

Waits for the occurrence of selected events. Returns the event which actually occurred, with bit allocation as for 'flag' below.

flag > Sets the event(s) to be awaited as follows:

Bit 0	keyboard	MU_KEYBD
Bit 1	mouse button	MU_BUTTON
Bit 2	first mouse event	MU_M1
Bit 3	second mouse event	MU_M2
Bit 4	report event	MU_MESAG
Bit 5	timer	MU_TIMER

num_clicks < Number of expected mouse clicks

The parameters were already described for EVNT_MOUSE, EVNT_KEYBD, EVNT_BUTTON and EVNT_MESAG. However, it should be noted that two different mouse events (m1 and m2) can be awaited. With ON MENU, which uses this routine internally, the parameters are installed for the instruction ON MENU xxx GOSUB, e.g. 'count' is specified directly.

MENU(1) to MENU(8)	Message buffer
MENU(9)	Returned value
MENU(10) = mcur_x	x mouse position
MENU(11) = mcur_y	y mouse position
MENU(12) = button	Mouse button state
MENU(13) = k_state	'Shift' keys state (BIOS (11,-1))
MENU(14) = key	ASCII and Scan code
MENU(15) = num_clicks	Number of mouse clicks

EVNT_DCLICK(new,get_set)

new,get_set : iexp

Sets the speed for double-clicks of a mouse button.

Returns the speed.

new > New speed (0 to 4).
get_set > Determines whether the speed is to be set, or just read:

0 = Determines the current speed. (Then 'new' is a dummy).
1 = Sets 'new' as new speed.

Menu Library

Used for drawing the Menu Bar and managing its operation.

MENU_BAR(tree,flag)

tree,flag : iexp

Displays/erases a menu bar (from a Resource file).
Compare: MENU x$() and MENU KILL.

Returns 0 if an error occurred.

tree > Address of the menu object tree
flag > 1 = Display menu bar
0 = Erase menu bar (happens automatically at the end of a program)

MENU_ICHECK(tree,item,flag)

tree,item,flag : iexp

Deletes a tick (see below) against a menu item (which should have at least two spaces reserved for it).

Compare: MENU x,0 and MENU x,1.

Returns 0 if an error occurred.

tree > Address of the menu object tree
item > Object number of the menu item
flag > 0 = Delete tick
1 = Display tick

MENU_IENABLE(tree,item,flag)

tree,item,flag : iexp

This causes the enabling or disabling of menu items. The menu item will then appear with normal characters (selectable), or grey characters (not selectable) respectively.

Compare: MENU x,2 and MENU x,3.

Returns 0 if an error occurred.

tree > Address of the menu object tree
item > Object number of the menu entry
flag > 0 = Disabled
1 = Enabled

MENU_TNORMAL(tree,title,flag)

tree,title,flag : iexp

Menu title switches to inverse or normal video.
Compare: MENU OFF.

Returns 0 if an error occurred.

tree > Address of the menu object tree
title > Object number of the menu title
flag > 0 = Inverse video
1 = Normal

MENU_TEXT(tree,item,new_text$)

tree,item : iexp
new_text$: sexp

Changes the text of a menu item. This function permits the adapting of menu entries while a program is running.

Returns 0 if an error occurred.

tree > Address of the menu object tree
item > Object number of the item to be modified
new_text$ > A string containing the new menu item (may not exceed the length of the old one).

MENU_REGISTER(ap_id,m_text$)

ap_id : iexp
m_text$: sexp

A Desk Accessory name is inserted into the first menu, provided that this does not carry the number of Accessories beyond six (they are numbered 0 to 5). Note: this function can only be used with a Desk Accessory, not a compiled program, or any other program.

Returns the object number of the appropriate menu item:

0 to 5 for the first to sixth Accessory entry
-1 if no further entry is possible

ap_id > Identification number of the Accessory
m_text$ > Name under which the Accessory is to be entered into the menu (normally this is the Accessory name).

Object Library

Contains routines for the definition, drawing and alteration of objects.

OBJC_ADD(tree,parent,child)

tree,parent,child : iexp

An object is added to a given object tree and pointers between the existing objects and the new object are created.

Returns 0 if an error occurred.

tree > Address of the object tree.
parent > Object number of the parent object, of which the new object is to be a child.
child > Object number of the 'child object' to be added

OBJC_DELETE(tree,del_obj)

tree,del_obj : iexp

An object is deleted from an object tree by changing pointers in the tree. The object itself will still be there and can be restored later by restoring the pointers.

Returns 0 if an error occurred.

tree > Address of the object tree
del_obj > Object number of the object to be deleted

OBJC_DRAW(tree,start_obj,depth,cx,cy,cw,ch)

tree,start_obj,depth,cx,cy,cw,ch : iexp

This function draws whole objects or parts of objects on the screen. A clipping rectangle can be specified, to which the drawing is limited.

Returns 0 if an error occurred.

tree > Address of the object tree
start_obj > Object number of the first object to be drawn
depth > Number of object levels to be drawn (0 = only first object)
cx,cy > Coordinates of top left corner of clipping rectangle
cw,ch > Width and height of clipping rectangle

OBJC_FIND(tree,start_obj,depth,fx,fy)

tree,start_obj,depth : iexp
fx,fy : ivar

This function determines the object, if any, which contains the mouse pointer. The mouse coordinates can also be found.

Returns the object number, or -1 if the mouse pointer is not inside any object.

tree > Address of the object tree to be examined
start_obj > Number of the object from which the search is begun
depth > Number of object levels to be searched (0 = only first object)

fx > Mouse x-coordinate
fy > Mouse y-coordinate

The explanation is misleading. Objc_find finds the object at the coordinates fx, fy. You can of course supply the mouse coordinates if you wish, but this call has nothing to do with the mouse.

OBJC_OFFSET(tree,obj,x_abs,y_abs)

tree,obj : iexp
x_abs,y_abs : ivar

Computes the absolute screen coordinates of the specified object.

Returns 0 if an error occurred

tree > Address of the object tree
obj > Object number

x_abs < The computed x-coordinate
y_abs < The computed y-coordinate

OBJC_ORDER(tree,obj,new_pos)

tree,obj,new_pos : iexp

Re-positions an object within a tree.

Returns 0 if an error occurred

tree > Address of the object tree
obj > Object number
new_pos > New level number:
-1 = One level higher
0 = Bottom level
1 = Bottom level + 1
2 = Bottom level + 2
etc.

OBJC_EDIT(tree,obj,char,old_pos,flag,new_pos)

tree,obj,char,old_pos,flag : iexp
new_pos : ivar

Facilitates the input and editing of text in the G_TEXT and G_BOXTEXT object types.

Returns 0 if an error occurred.

tree > Address of the object tree
obj > Object number
char > Input character (including scan code)
old_pos > Current cursor position in input string
flag > Function selection:

0 ED_START = -reserved- (Function call does nothing)
1 ED_INIT = String is formatted and cursor switched on
2 ED_CHAR = Character processed and string re-displayed
3 ED_END = Text cursor switched off

new_pos < New cursor position in input string

OBJC_CHANGE(tree,obj,res,cx,cy,cw,ch,new_status,re_draw)

tree,obj,res,cx,cy,cw,ch,new_status,re_draw : iexp

This function changes the status of an object (OB_STATE) and, if necessary, inverts (displays in inverse video) that section of the object lying inside the clipping rectangle. (Normally OB_STATE would be changed directly and the object redrawn with OBJC_DRAW.)

Returns 0 if an error occurred.

tree	>	Address of the object tree
obj	>	Number of the object to be changed
res	>	-reserved- (Always 0)
cx,cy	>	Coordinates of top left corner of clipping rectangle
cw,ch	>	Width and height of the clipping rectangle
new_status	>	New object status (see OB_STATE):
		1 = Redraw object
		0 = Don't redraw

Form Library

The Form Library contains routines for Form management, i.e. manipulation of objects in Dialog boxes etc.

FORM_DO(tree,start_obj)

tree,star_obj: iexp

This function takes over the complete management of a Form object, until an object with EXIT or TOUCH_EXIT status is clicked.

Returns the number of the object whose clicking or double-clicking caused the function to be ended. If it was a double-click, bit 15 will be set.

tree > Address of the object tree
start_obj > Number of the first editable object in the object tree, where the cursor is initially to be positioned. If there is no editable object the value in this parameter is irrelevant.

FORM_DIAL(flag,mi_x,mi_y,mi_w,mi_h,ma_x,ma_y, ma_w,ma_h)

flag,mi_x,mi_y,mi_w,mi_h,ma_x,ma_y,ma_w,ma_h : iexp

This function serves to reserve (or release) a rectangular screen area and for drawing expanding or shrinking rectangles.

Returns 0 if an error occurred.

flag > Function type:

0	=	FMD_START	reserves a display area
1	=	FMD_GROW	draws an expanding rectangle
2	=	FMD_SHRINK	draws a shrinking rectangle
3	=	FMD_FINISH	releases reserved display area again

mi_x,mi_y > Top left corner of the rectangle at minimum size
mi_w,mi_h > Width and height of the rectangle at minimum size
ma_x,ma_y > Top left corner of the rectangle at maximum size
ma_w,ma_h > Width and height of the rectangle at maximum size

Example:

```
~FORM_DIAL(1,0,0,0,0,100,100,300,100)
```

--> Draws an expanding rectangle. The parameter group '0,0,0,0' means that the rectangle grows from the centre of what will be the full-size rectangle.

FORM_ALERT(button,string$)

button : iexp
string$: sexp

Creates a general-purpose ALERT box.

Returns the number of the clicked button which caused the function to terminate.

button > Number of the default (thick-bordered) button:

0 = None
1 = First
2 = Second
3 = Third

string$ > A string defining the contents of the Alert Box. The string has the following format (note that the square brackets are part of the string):

[i][Message][Buttons]

where:

i = Required symbol in Alert Box:
0 = No icon
1 = Exclamation mark
2 = Question mark
3 = STOP symbol

Message = At the most 5 lines of text, with a maximum of 30 characters per line, with the lines separated with rule (|) symbols.
Button = A maximum of 3 button names, separated by '|' symbols.

Example:

```
a$="[1][This is the first line|+|+|+|"+"This is the
                    fifth line]"+"[One|Two|Three]"
DO
  PRINT FORM_ALERT(1,a$);
LOOP
```

--> Prints 1, 2 or 3, depending on which button is clicked, or pressing RETURN selects the highlighted button. The program can be exited by pressing the break keys (Shift-Control-Alternate) but this will only take effect after the next click.

FORM_ERROR(err)

err : iexp

Displays the warning box associated with DOS Error 'err'.

Returns the number of the button which terminated the function.

err > Error number

Example:

```
FOR x=0 TO 63
  PRINT FORM_ERROR(x);
NEXT x
```

--> The boxed Error messages are displayed.

FORM_CENTER(tree,fx,fy,fw,fh)

tree : iexp
fx,fy,fw,fh : ivar

This function centres the tree, i.e. Dialog box, etc, on the screen. Its position can then be found.

Returns: a reserved value - at the moment always 1.

tree > Address of the object tree

fx,fy < Coordinates of top left corner
fw,fh < Width and height of the centred box

FORM_KEYBD(tree,obj,next_obj,char,new_obj,next_char)

tree,obj,next_obj,char,new_obj : iexp
next_char : ivar

Allows a Form to be filled out via the keyboard (see OBJC_EDIT).

Returns 0 if the Form was left by clicking on an object with EXIT or TOUCH_EXIT status. Returns a value > 0 if the Form is not finished with.

tree	>	Address of the object tree
obj	>	Number of the object to be edited
next_obj	>	Number of the next EDITable object in the tree
char	>	Input character
new_obj	>	Object to be EDITed on the next call
next_char	<	Next character (derived from keyboard, etc.)

Explanation: This routine is a subroutine of FORM_DO and makes it possible to test a character in BASIC before it is passed to GEM, e.g. to test for a RETURN character, which would normally cause an exit from the Form and allow it to be used to terminate an individual entry, etc.

FORM_BUTTON(tree,obj,clicks,new_obj)

tree,obj,clicks : iexp
new_obj : ivar

Makes possible mouse inputs in a Form.

Returns 0 if the Form was exited by clicking an object with EXIT or TOUCH_EXIT status. Returns a value greater than 0 if the Form is not yet finished with.

tree > Address of the object tree
obj > Current object number
clicks > Maximum expected number of mouse clicks

new_obj < Next object to be edited

Explanation: This routine is a subroutine of FORM_DO.

Graphics Library

GRAF_RUBBERBOX(lx,ty,min_w,min_h [last_w,last_h])

lx, ty, min_w, min_h : iexp
last_w, last_h : ivar

The function GRAF_RUBBERBOX draws an outline of a rectangle while the left button is held down. The top left corner is fixed, but the width and height of the rectangle change with the position of the mouse. The function should be called only when a button is pressed, since it terminates when the button is released.

Returns 0 if an error occurred.

lx, ty > Coordinates of top left corner
min_w, min_h > Minimum width of rectangle
min_h > Minimum height of rectangle

last_w < Width of rectangle when function terminates
last_h < Height of rectangle when function terminates

Example:

```
DO
  ~EVNT_BUTTON(1,1,1,mx%,my%,bu%,kb%)
  ~GRAF_RUBBERBOX(mx%,my%,1,1,w%,h%)
  BOX mx%,my%,mx%+w%,my%+h%
LOOP UNTIL BTST(kb%,3)
```

--> EVNT_BUTTON waits for the left button to be pressed, then puts the mouse coordinates into mx% and my%, and the status of the keyboard 'shift' keys into kb%. The variables Mx% and my% are then passed to GRAF_RUBBERBOX, which continuously draws rectangles from (mx%,my%) to the current mouse position. When the function is terminated by releasing the button, GRAF_RUBBERBOX puts the last width and height into w% and h%. A fixed box is then drawn with the standard BOX command. The program then ends if the Alternate key was held down when the mouse button was first pressed (i.e. bit 3 of kb% is set to 1), or otherwise loops round to draw another box.

GRAF_DRAGBOX(iw,ih,ix,iy,rx,ry,rw,rh [last_ix,last_iy])

iw, ih, ix, iy, vx, vy, vw, vh : iexp
lastx, last_iy : ivar

This function allows a rectangle to be moved about the screen with the mouse. Its movement is restricted to the interior of a larger specified rectangle. The function should only be called when the left button is held down, as it terminates when the button is released.

Returns 0 if an error occurred.

iw, ih	>	Width and height of the moving rectangle
ix, iy	>	Initial coordinates of top left corner of moving rectangle
rx, ry	>	Coordinates of top left corner of limiting rectangle
rw, rh	>	Width and height of limiting rectangle
last_ix	<	Coordinates of the top left corner of inside rectangle
last_iy	<	when the function terminated.

Example:

```
REPEAT
UNTIL MOUSEK=1
~GRAF_DRAGBOX(25,25,50,50,10,10,150,150,lx%,ly%)
BOX lx%,ly%,lx%+25,ly%+25
```

--> When the left mouse button is pressed, a small rectangle will move with the mouse pointer, provided that this does not take it outside the specified larger rectangle (in this case 10,10 - 160,160). When the button is released, the function terminates and the smaller rectangle is redrawn at its final position with the standard BOX command.

GRAF_MOVEBOX(w,h,sx,sy,dx,dy)

w, h, sx, sy, dx, dy : iexp

The function GRAF_MOVEBOX draws a moving rectangle with constant width and height.

Returns 0 if an error occurred.

w, h > Width and height of the rectangle
sx, sy > Initial coordinates of top left corner of the rectangle
dx, dy > Final coordinates of the top left corner

Example:

```
~GRAF_MOVEBOX(25,25,0,0,150,150)
```

GRAF_GROWBOX(sx,sy,sw,sh,dx,dy,dw,dh)

sx,sy,sw,sh,dx,dy,dw,dh : iexp

This function draws an expanding rectangle.

Returns 0 if an error occurred.

sx, sy > Initial coordinates of the top left corner of the rectangle
sw, sh > Initial width and height of the rectangle
dx, dy > Final coordinates of the top left corner
dw, dh > Final width and height

Example:

```
BOX 100,100,110,110
PAUSE 25
~GRAF_GROWBOX(100,100,10,10,0,0,300,180)
BOX 0,0,300,180
```

GRAF_SHRINKBOX(sx,sy,sw,sh,dx,dy,dw,dh)

sx, sy, sw, sh, dx, dy, dw, dh : iexp

This function draws a shrinking rectangle.

Returns 0 if an error occurred.

sx, sy > Final coordinates of top left corner of rectangle
sw, sh > Final width and height of the rectangle
dx, dy > Initial coordinates of top left corner
dw, dh > Initial width and height

Note that the FINAL coordinates are given first.

Example:

```
BOX 0,0,300,180
PAUSE 25
~GRAF_SHRINKBOX(100,100,10,10,0,0,300,180)
BOX 100,100,110,110
```

GRAF_WATCHBOX(tree,obj,in_state,out_state)

tree, obj, in_state, out_state : iexp

This function (which really belongs in the Object Library) monitors an object in a tree while a mouse button is pressed, checking whether the mouse pointer is inside it or outside. When the button is released, the status of the object takes one of two specified values (normal selected/normal), depending on whether the pointer was located inside the object, or outside.

Returns 1 if the mouse pointer was inside the object when the button was released, or 0 if it was outside.

tree	>	Address of the object tree
obj	>	Number of the object to be monitored
in_state	>	Status (OB_STATE) to be given to the object if the mouse pointer is found to be within it.
out_state	>	Status (OB_STATE) to be given to the object if the mouse pointer is found to be outside it. The appropriate bit allocation is found under OB_STATE.

GRAF_SLIDEBOX(tree,parent_obj,slider_obj,flag)

tree,parent_obj,slider_obj,flag : iexp

This function also really should be in the Object Library. It moves one rectangular object within another in a similar way to GRAF_DRAGBOX. However, the object can only be moved horizontally or vertically and, in addition, it must be a 'child' of the limiting rectangle (object). The function call should only take place when a mouse button is pressed, since the function terminates when the button is released. The most common application is the movement of slider bars in windows.

Returns the position of the moving rectangle relative to the limiting one:

Horizontally : 0 = far left 1000 = far right
Vertically : 0 = top, 1000 = bottom

tree > Address of the object tree
parent_obj > Object number of the "limiting rectangle"
slider_obj > Object number of the moving rectangle
flag > Direction:
0 = Horizontal
1 = Vertical

GRAF_HANDLE(char_w,char_h,box_w,box_h)

char_w, char_h, box_w, box_h : ivar

Returns the Identification number of the current VDI Workstation, which is used internally for AES calls, and supplies the size of a character from the system character set.

char_w < Width in pixels of a character from the standard set
char_h < Height in pixels of a character from the standard set
box_w < Width of a standard character cell
box_h < Height of a standard character cell

GRAF_MOUSE(m_form,pattern_adr)

m_form, pattern_adr : iexp

This function allows the appearance of the mouse pointer to be changed. Eight pre-defined pointers are available, or one may be defined by the user. (However, the command DEFMOUSE is more convenient to use.)

Returns 0 is an error occurred.

m_form > Number of the mouse pointer shape:

0	=	Arrow
1	=	Double clips
2	=	Busy bee
3	=	Pointing hand
4	=	Open hand
5	=	Thin cross-hairs
6	=	Thick cross-hairs
7	=	Outlined cross-hairs
255	=	User defined
256	=	Hide mouse pointer
257	=	Show mouse pointer

pattern_adr > Address of bit information defining the mouse pointer as desired. 37 word-sized values are expected, as follows:

1 =	x-coordinate of the action point (i.e. the point referred to by MOUSEX, MOUSEY etc.)
2 =	y-coordinate of the action point
3 =	Number of colour levels, always 1
4 =	Mask colour, always 0
5 =	Pointer colour, always 1
6 to 21 =	Mask definition (16 words, i.e = 16x16 bits)
22 to 37 =	Pointer definition (16 words, i.e = 16x16 bits)

GRAF_MKSTATE(mx,my,m_state,k_state)

mx, my, m_state, k_state : ivar

This function supplies the current mouse pointer coordinates and the status of the mouse buttons and the keyboard 'shift' keys.

This is an AES routine to query the mouse. Unlike MOUSEX etc., the function gives valid results if the pointer is within a menu bar.

Returns: a reserved value, at the moment always 1.

mx, my < Current coordinates of the mouse pointer
m_state < Mouse button status:
Bit 0 = left button
Bit 1 = right button
k_state < Status of keyboard 'shift' keys (if key is pressed, bit is set):
Bit 0 = Right shift key
Bit 1 = Left shift key
Bit 2 = Control key
Bit 3 = Alternate key

Scrap Library

Contains routines enabling the exchange of data between different applications.

SCRP_READ(path$)

path$: svar

This function reads data, perhaps left there by another program, from a small internal buffer, thus allowing communication between successively implemented GEM programs. The data would often, but not necessarily, consist of a file specification, indicating a source of data on disc.

Returns 0 if an error occurred.

path$ < Data string or file specification.

SCRP_WRITE(path$)

path$: sexp

The opposite of SCRP_READ. A small amount of data, perhaps a file specification, is written into an internal buffer.

Returns 0 if an error occurred.

path$ > Data string or file specification.

Example:

```
~SCRP_WRITE("c:\TMP\A.X")
```

Quit from BASIC here, and reload it, then...

```
~SCRP_READ(a$)
PRINT a$
```

File Selector Library

The File Selector Library routine enables the user to select a file from a displayed directory, or to specify a file by typing its name.

FSEL_INPUT(path$,name$,[button])

path$, name$: svar
button : ivar

This function invokes the standard Fileselect Box, and corresponds to the FILESELECT instruction. The initial directory path and the default filename are contained in the string variables path$ and name$. After the fileselect box has been used in the normal way, and the function exited by clicking on 'OK' or 'Cancel', these strings contain the last used directory path and the chosen filename respectively. The optional integer variable 'button' contains 1 or 0, depending on whether the 'OK' or 'Cancel' button was clicked.

Returns 0 if an error occurred.

On entry:	path$	>	Initial directory path
	name$	>	Default filename
On exit :	path$	<	Final directory path
	name$	<	Chosen filename
	button	<	1 if 'OK' was clicked to exit 0 if 'Cancel' was clicked on

Example:

```
p$="a:\*.*"
n$=""
DO
  ~FSEL_INPUT(p$,n$,b)
  CLS
  PRINT p$
  PRINT n$
  PRINT b
LOOP UNTIL b=0
```

--> Various filenames can be selected and displayed. The program finishes when the 'Cancel' button is clicked.

Window Library

The Window Library contains all functions relating to Window management.

WIND_CREATE(attr,wx,wy,ww,wh)

attr,wx,wy,ww,wh : iexp

This function allocates a new window, specifying the attributes and maximum size. The window identification number ('handle') is returned, or 0 if an error occurred.

Compare: OPENW #n,x,y,w,h,attr

Returns window ID, or 0 if an error occurred.

attr > Window attributes as follows:

			Bit no.
&H0001	NAME	Title bar with name	0
&H0002	CLOSE	Close box	1
&H0004	FULL	Full box	2
&H0008	MOVE	Move box	3
&H0010	INFO	Information line	4
&H0020	SIZE	Size box	5
&H0040	UPARROW	Up arrow	6
&H0080	DNARROW	Down arrow	7
&H0100	VSLIDE	Vertical slider bar	8
&H0200	LFARROW	Left arrow	9
&H0400	RTARROW	Right arrow	10
&H0800	HSLIDE	Horizontal slider bar	11

wx > Maximum x-position of left edge
wy > Maximum y-position of top edge
ww > Maximum width of the window
wh > Maximum height of the window

WIND_OPEN(handle,wx,wy,ww,wh)

handle,wx,wy,ww,wh : iexp

This function draws on the screen a window previously created with WIND_CREATE.

Compare: OPENW

Returns 0 if an error occurred.

handle > Identification number of the window
wx > Left x-coordinate
wy > Top y-coordinate
ww, wh > Initial width and height

WIND_CLOSE(handle)

handle : iexp

This function is the counterpart of WIND_OPEN and closes the specified window.

Compare: CLOSEW

Returns 0 if an error occurred.

handle > Identification number of the window

WIND_DELETE(handle)

handle : iexp

This function deletes a window allocation and frees the reserved memory and window identification number for re-use.

Compare: CLOSEW

Returns 0 if an error occurred

handle > Identification number of the window

WIND_GET(handle,code,w1,w2,w3,w4)

handle, code : iexp
w1,w2,w3,w4 : ivar

By specifying the function code 'code', this function supplies information about a window.

Returns 0 if an error occurred.

handle > Identification number of the window

code > Depending on the code specified, information is supplied in w1,w2, w3 and w4 as follows:

4 WX_WORKXYWH supplies the size of the window work area:
w1 < left x-coordinate
w2 < top y-coordinate
w3 < width
w4 < height

5 WF_CURRXYWH supplies the total size of the entire window including the borders:
w1 < left x-coordinate
w2 < top y-coordinate
w3 < width
w4 < height

6 WF_PREVXYWH supplies the total size of the previous window:
w1 < left x-coordinate
w2 < upper y-coordinate
w3 < width
w4 < height

7 WF_FULLXYWH supplies the total maximum size of the window (set by WIND_CREATE):
w1 < left x-coordinate
w2 < upper y-coordinate
w3 < width
w4 < height

8 WF_HSLIDE supplies the position of the horizontal slider:
w1 < (1 = far left, 1000 = far right)

9 WF_VSLIDE supplies the position of the vertical slider:
w1 < (1 = top, 1000 = bottom)

10 WF_TOP supplies the identification number of the top (=active) window:
w1 < identification number

11 WF_FIRSTXYWH supplies the coordinates of the first rectangle in the specified window's rectangle list. (The list of rectangles required to build up the window: an unobscured window has one rectangle in its list, a window partially obscured by another window has several, a totally obscured window has none):
w1 < left x-coordinate
w2 < top y-coordinate
w3 < width
w4 < height

12 WF_NEXTXYWH supplies the coordinates of the next rectangle in the specified window's rectangle list:
w1 < left x-coordinate
w2 < top y-coordinate
w3 < width
w4 < height

13 WF_RESVD -reserved-

15 WF_HSLIZE supplies the size of the horizontal slider bar compared to its maximum possible size:
w1 < -1 = minimum size (1 = small, 1000 = full width)

16 WF_VSLIZE supplies the size of the vertical slider bar compared to its maximum possible size:
w1 < -1 = Minimum size
(1 = small, 1000 = full height)

WIND_SET(handle,code,w1,w2,w3,w4)

handle, code : iexp
w1, w2, w3, w4 : ivar

This function changes parts of a window according to the specified function code.

Returns 0 if an error occurred.

handle > Identification number of the window

code > Specifies components to be changed:

1 WF_KIND sets new window components (as with WIND_CREATE).
w1 > new window part.

2 WF_NAME gives a window a new title:
w1, w2 > High-word, low-word of the address of the title string (terminated with two null (0) bytes)

3 WF_INFO specifies a new information line:
w1, w2 > as above
Note: It is best to use the functions TITLEW and INFOW in place of the above, if you are not sure of how to make the strings stay put during the running of a program.

5 WF_CURRXYWH sets the total window size:
w1 > Left x-coordinate
w2 > Top y-coordinate
w3 > Width
w4 > Height

8 WF_HSLIDE positions the horizontal slider:
w1 > 1=far left, 1000=far right

9 WF_VSLIDE positions the vertical slider:
w1 > 1=top, 1000=bottom

10 WF_TOP Sets the top (currently active) window

14 WF_NEWDESK Sets a new desktop Menu tree:
w1, w2 > Low-word, high-word of address of new tree
w3 > ID number of the first object to be drawn
Note: word order is the opposite of the normal order

15 WF_HSLIZE Sets the size of the horizontal slider bar compared to its maximum possible size:
w1 > -1 = minimum size
(1 = small, 1000 = full width)

16 WF_VSLIZE Sets the size of the vertical slider bar compared to its maximum possible size:
w1 > -1 = Minimum size
(1 = small, 1000 = full height)

WIND_FIND(fx,fy)

fx,fy : iexp

This function determines the Identification number of a window within which the the specified coordinates lie.

Returns the Identification number of found window, or 0 if not found.

fx > x-coordinate
fy > y-coordinate

WIND_UPDATE(flag)

flag : iexp

This function coordinates all functions connected with screen redrawing, in particular in combination with Pull-down Menus.

Returns 0 if an error occurred.

flag		Function
0	=	END_UPDATE screen redraw completed
1	=	BEG_UPDATE screen redraw starting
2	=	END_MCTRL application relinquishes mouse supervision
3	=	BEG_MCTRL application takes over mouse supervision; GEM functions for menu and window handling are inactive.

WIND_CALC(w_type,attr,ix,iy,iw,ih,ox,oy,ow,oh)

w_type,attr,ix,iy,iw,ih : iexp
ox,oy,ow,oh : ivar

This function computes the total size of a window (including slider bars, etc.) from the size of the work area, or, conversely, the size of the work area from the total size of the window.

Returns 0 if an error occurred.

w_type > 0 = Compute total size
1 = Compute work area size

attr > Window components:

			Bit no.
&H0001	NAME	Title bar with name	0
&H0002	CLOSE	Close box	1
&H0004	FULL	Full-size box	2
&H0008	MOVE	Movement bar	3
&H0010	INFO	Information line	4
&H0020	SIZE	Size box	5
&H0040	UPARROW	Up-arrow	6
&H0080	DNARROW	Down-arrow	7
&H0100	VSLIDE	Vertical slider	8
&H0200	LFARROW	Left-arrow	9
&H0400	RTARROW	Right-arrow	10
&H0800	HSLIDE	Horizontal slider	11

ix,iy > Known top-left coordinates
iw,ih > Known width and height

ox,oy < Calculated top-left coordinates
ow,oh < Calculated width and height

Resource Library

The Resource Library provides routines for the creation of a graphical user interface (i.e. Dialog boxes, etc.), which allow, independently of the current screen resolution, the exchange of data between user and program.

RSRC_LOAD(name$)

name$: sexp

This function reserves memory and loads a resource file. Then internal pointers are set and the coordinates of characters converted into pixel format. (For Resources which have been defined directly in memory, with POKE,etc, RSRC_OBFIX must be used to do this). If the file cannot be found, RSRC_LOAD automatically does a little searching, as detailed under SHEL_FIND.

Returns 0 if an error occurred.

name$ > File specification of the Resource file.

Example:

```
~RSRC_LOAD("TEST.RSC")
```

RSRC_FREE()

This function releases the memory space reserved by RSRC_LOAD.

Returns 0 if an error occurred.

Example:

```
~RSRC_FREE()
```

RSRC_GADDR(type,index,addr)

type,index : iexp
addr : ivar

This function (Resource Get ADDRess) determines the address of a resource structure after it has been loaded with RSRC_LOAD. Depending on the version of GEM, this function may only work for Object trees and Alerts (ad_frstr).

Returns 0 if an error occurred.

type > Type of structure whose address is to be found:

0	Object tree	(the tree loaded with RSRC_LOAD)
1	OBJECT	(object)
2	TEDINFO	(text information)
3	ICONBLK	(icon information)
4	BITBLK	(bit-mapped graphic information)
5	STRING	(text)
6	image data	(bit-mapped graphic)
7	obspec	(object specification)
8	te_ptext	(string)
9	te_ptmplt	(text template)
10	te_pvalid	(text validation string)
11	ib_pmask	(icon display mask)
12	ib_pdata	(icon bit map)
13	ib_ptext	(icon text)
14	bi_pdata	(image data)
15	ad_frstr	(address of pointer to free string)
16	ad_frimg	(address of pointer to free image)

index > The number (not the Object Number) of the object whose address is required, counting objects of that type one by one from the beginning of the Resource file.

addr < The required address

Example:

```
~RSRC_GADDR(0,0,TREE%)
```

RSRC_SADDR(type,index,addr)

type,index : iexp
addr : ivar

This function sets the address of an object.

Returns 0 if an error occurred.

type > Type of structure (see RSRC_GADDR above)
index > The number (not the Object Number) of the object whose address is to be set, counted object by object from the beginning of the Resource file.
addr > The address

RSRC_OBFIX(tree,obj)

tree, obj : iexp

This function converts the coordinates of an object within a tree, from character coordinates to pixel coordinates, taking into account the current screen resolution. It is called automatically by RSRC_LOAD, but must be used if the object is created directly in memory by POKE, etc.

Returns a reserved value, at the moment always 1.

tree > Address of the appropriate object tree
obj > Object number of the object to be adjusted

Shell Library

The Shell Library routines enable one application to call another, preserving both the original application and its environment.

SHEL_READ(cmd_string$,tail_string$)

cmd_string$,tail_string$: svar

This function allows the program to identify the command by which it was invoked, and supplies the name, e.g. GFABASIC.PRG, and the command line, if any, from a .APP).

Returns 0 if an error occurs.

cmd_string$ < String variable to contain the command line
tail_string$ < String variable to contain the name

SHEL_WRITE(prg,grf,gem,cmd$,nam$)

prg, grf, gem : iexp
cmd$, nam$: sexp

This function informs the AES that another application is to be started after the current one has terminated. In contrast to p_exec (GEMDOS 75), however, the current program does not remain resident in memory.

Returns 0 if an error occurred.

prg > 0 = Back to the Desktop
1 = Load new program
grf > 0 = TOS program
1 = Graphic application

gem > 0 = Not GEM application
1 = GEM application
cmd$ > Command line string
nam$ > Name of next application

Example:

```
~SHEL_WRITE(1,1,1,"","GFABASIC.PRG")
```

--> After this, quitting the BASIC and returning to the Desktop will result in the BASIC being restarted.

SHEL_GET(num,x$)

num : iexp
x$: svar

This function reads data from the GEMDOS environmental string buffer (into which the file DESKTOP.INF is read on start-up).

Returns 0 if an error occurred.

num > Number of bytes to be read

x$ < String variable to contain data

Example:

```
PRINT SHEL_GET(500,x$)
PRINT x$
~INP(2)
```

--> Either the contents of DESKTOP.INF or data for the default desktop is read into x$, and printed.

SHEL_PUT(len,x$)

len : iexp
x$: sexp

This function writes data into the GEMDOS environmental string buffer.

Returns 0 if an error occurred.

x$ > String containing the data to be written
len > Number of bytes to be written

Example:

```
' Register GFA-BASIC
~SHEL_GET(2000,a$)
q%=INSTR(a$,CHR$(26))
IF q%
  a$=LEFT$(a$,q%-1)
  IF INSTR(a$,"GFABASIC.PRG")=0
    a$=a$+"#G 03 04 A:\GFABASIC.PRG@*.GFA@"
                              +MKI$(&HD0A)+CHR$(26)
~SHEL_PUT(LEN(a$),a$)
  ENDIF
ENDIF
```

--> The program 'registers' A:\GFABASIC.PRG, so that with a double-click on a .GFA program file, GFABASIC is loaded which then loads and runs the program which was clicked on.

Note, to save this in the DESKTOP file:

```
~SHEL_GET(3000,a$)
OPEN"O",#1,"A:\ DESKTOP.INF"
PRINT #1,LEFT$(A$,INSTR(A$,CHR$(26)))'
CLOSE #1
' The CHR$(26) is important
```

SHEL_FIND(path$)

path$: svar

This function searches for a specified file and supplies the full file specification. First, the specified path on the current drive is searched, then the root directory of the current drive, then the root directory of drive A:.

Returns 0 if the filename was not found, or 1 if it was.

On entry:
path$ > String containing the sought-after filename

On exit:
path$ < Contains the full file specification if the file was found, otherwise it is unchanged.

SHEL_ENVRN(addr,search$)

addr : avar (32-bit)
search$: sexp

This function determines the values of variables in the GEM environment.

Returns a reserved value, at the moment always 1.

search$ > The string to be sought

addr < Address of the byte following the specified string

Example:

```
PRINT SHEL_ENVRN(a%,"PATH")
PRINT CHAR{a%-4}
```

--> Displays: PATH=; A:\

Sample Programs

In this last section example programs are provided, dealing with the Graphics Library, Dialog boxes, Menus and Window programming. Note that it is important to exit from all these programs in the ways specified, not by just breaking in with Shift-Control-Alternate, otherwise memory re-allocation, etc. will not take place and the programs (or other programs) may subsequently fail to work without resetting the computer.

```
' ** Graphics Library
' GRAF_SMP.GFA
'
REPEAT
  CLS
  PRINT CHR$(27)+"p";
  PRINT "| <F1> rubber | <F2> drag | <F3> move |";
  PRINT " <F4> grow_shrink | <F10> quit |"
  choice|=INP(2)
  '
  SELECT choice|
  CASE 187      ! F1
    rubber
  CASE 188      ! F2
    drag
  CASE 189      ! F3
    move
  CASE 190      ! F4
    grow_shrink
  ENDSELECT
  '
  PRINT CHR$(27)+"q";
  '
UNTIL choice|=196 ! quit with F10
'
EDIT
'
PROCEDURE rubber
  GRAPHMODE 3
```

```
  DEFFILL 1,2,4
  REPEAT
    MOUSE mx%,my%,mk%
    IF mk% AND 1
      x1%=mx%
      y1%=my%
      ~GRAF_RUBBERBOX(x1%,y1%,16,16,lx%,ly%)
      PBOX x1%,y1%,x1%+lx%,y1%+ly%
    ENDIF
  UNTIL mk% AND 2
RETURN
'
PROCEDURE drag
  GRAPHMODE 3
  BOX 40,40,400,300
  lx%=50
  ly%=50
  REPEAT
    BOX lx%,ly%,lx%+150,ly%+100
    '
    REPEAT
      mk%=MOUSEK
    UNTIL mk%                    ! i.e. until mousek<>0
    '
    IF mk% AND 1                          ! left button
      BOX lx%,ly%,lx%+150,ly%+100
      ~GRAF_DRAGBOX(150,100,lx%,ly%,40,40,360,260,lx%,ly
      BOX lx%,ly%,lx%+150,ly%+100
    ENDIF
    BOX lx%,ly%,lx%+150,ly%+100
  UNTIL mk% AND 2                        ! right button
  GRAPHMODE 1
RETURN
'
PROCEDURE move
  GRAPHMODE 1
  DEFFILL 1,2,4
  w%=100
  h%=100
```

```
  FOR i%=0 TO 639-w% STEP w%
    FOR j%=0 TO 399-h% STEP h%
      ~GRAF_MOVEBOX(w%,h%,i%,j%,639-i%,399-j%)
    NEXT j%
  NEXT i%
RETURN
'
PROCEDURE grow_shrink
  GRAPHMODE 1
  ~GRAF_GROWBOX(319,199,16,16,0,0,639,399)
  ALERT 0,"That was a growing box!",1,"Continue",r%
  ~GRAF_SHRINKBOX(319,199,16,16,0,0,639,399)
  ALERT 0,"That was a shrinking box!",1," Yes ",r%
RETURN
'
```

--> Pressing F1 to F4 causes the program to branch to the appropriate procedure. If F1 is pressed, the outline of a rectangle appears on the screen when the left mouse button is pressed and held down. The top left corner is fixed, but the diagonally opposite corner moves with the mouse. When the rectangle has the desired shape and size, and the left button is released, the rectangle is fixed and filled. More rectangles can be drawn, or the procedure exited by pressing the right button.

Pressing F2 causes a small rectangle to be drawn within a larger one. With the left button pressed and held down, the smaller rectangle can be moved about within the limits imposed by the larger one. Pressing the right button ends the procedure.

F3 calls up the demonstration of GRAF_MOVEBOX, where sequences of boxes move round the screen. This procedure ends by itself after a few seconds.

Finally, with F4, a growing box is drawn with GRAF_GROWBOX, and, after pressing RETURN, a shrinking box is drawn with GRAF_SHRINKBOX.

Pressing F10 ends the whole program.

```
' ** Dialog Box Example
' DIAL_SMP.GFA
'
DIM r%(3)
'
form1%=0     ! 0 = Dialog
icon1%=1     ! ICON in tree FORM1
ch_name%=2   ! FTEXT in tree FORM1
sur%=3       ! FTEXT in tree FORM1
str%=4       ! FTEXT in tree FORM1
town%=5      ! FTEXT in tree FORM1
cancel%=6    ! BUTTON in tree FORM1
ok%=7        ! BUTTON in tree FORM1
r%(1)=8      ! BUTTON in tree FORM1
r%(2)=9      ! BUTTON in tree FORM1
r%(3)=10     ! BUTTON in tree FORM1
output%=11   ! STRING in tree FORM1
'
~RSRC_FREE()
~RSRC_LOAD("\DIALOG.RSC")     ! Load Resource file
~RSRC_GADDR(0,0,tree_adr%)    ! Get address of
'                             ! Object tree
~FORM_CENTER(tree_adr%,x%,y%,w%,h%)  ! Centre the
'                          ! coordinates, depending on
'                          !  the current resolution
'
' Define initial editable strings
CHAR{{OB_SPEC(tree_adr%,ch_name%)}}="Sherlock"
CHAR{{OB_SPEC(tree_adr%,sur%)}}="Holmes"
CHAR{{OB_SPEC(tree_adr%,str%)}}="221b Baker Street"
CHAR{{OB_SPEC(tree_adr%,town%)}}="London N1"
'
~OBJC_DRAW(tree_adr%,0,1,x%,y%,w%,h%)
'                                  ! Draw Object tree
'
REPEAT
  ex%=FORM_DO(tree_adr%,0)         ! Clicked an object
'                                  ! with EXIT status?
```

```
'
' Put the texts from the Edit fields
' into the appropriate strings
  ch_name$=CHAR{{OB_SPEC(tree_adr%,ch_name%)}}
  surname$=CHAR{{OB_SPEC(tree_adr%,sur%)}}
  street$=CHAR{{OB_SPEC(tree_adr%,str%)}}
  town$=CHAR{{OB_SPEC(tree_adr%,town%)}}
  '
  FOR i%=1 TO 3
    IF BTST(OB_STATE(tree_adr%,r%(i%)),0)
'                              ! Which radio-button
      radio%=r%(i%)
'                              ! was selected?
    ENDIF
  NEXT i%
UNTIL ex%=ok% OR ex%=cancel%
'
~RSRC_FREE()     ! Release reserved memory
'
CLS
PRINT "Ended with:     : ";ex%
PRINT "Christian name  : ";ch_name$
PRINT "Surname         : ";surname$
PRINT "Street          : ";street$
PRINT "Town            : ";town$
PRINT "Radio:          : ";radio%
'
```

--> The file "DIALOG.RSC" is loaded, the address of the object tree is determined and the object tree coordinates are centred.

Then the function OB_SPEC is used to define the initial editable strings:

OB_SPEC supplies a pointer to a TEDINFO structure, which itself contains pointers to data pertaining to editable text (validation string, colour, text and template addresses, etc.). So the 'pointer path' to the actual information is shown overleaf:

Object --> TEDINFO structure --> String

OB_SPEC(tree%,ch_name%) supplies a pointer to the TEDINFO structure.

{OB_SPEC(tree%,ch_name%)} supplies the address held at that point in TEDINFO.

CHAR{{OB_SPEC(tree%,ch_name%)}} gives the string initially inserted by the Resource Construction Set, in this case: 012345678901 2345678901.

CHAR{{OB_SPEC(tree%,ch_name%)}}="Sherlock" assigns this preset character string a new value (namely 'Sherlock'). This is repeated for the rest of the initial data.

Then the object tree is drawn and, inside the REPEAT-UNTIL loop, the main FORM_DO routine is called. Its returned value, ex%, determines when the loop is exited.

The (edited) strings are then put back into their respective variables, after which the object status (found by OB_STATE) of the 'radio' buttons is examined in a FOR-NEXT loop. Specifically, Bit 0 is tested with BTST. If it is set (=1), then the corresponding button must have been selected (only one radio button can be selected at a time), and the object number of the button is stored in radio%.

Finally the memory space reserved for the Resource is released again with RSRC_FREE. (This is important!!, since otherwise the available free memory shrinks each time the program is started, which soon causes the computer to crash.)

The button which caused the exit of the REPEAT-UNTIL loop is printed (6 = Cancel, 7 = Ok), followed by the data strings as they were when FORM_DO was done for the last time and the object number of the last-selected radio button.

```
'
' ** Menu management
' MENU_SMP.GFA
'
RESERVE FRE(0)-33000              ! Reserve space for
'                                 ! the Resource file
IF RSRC_LOAD("folder\RCS2.RSC")=0  ! File is not in
'           ! 'folder', but RSRC_LOAD looks further
'           ! and should find it on current drive
  ALERT 1,"Resource not found",1,"End",a%
  END
ENDIF
~RSRC_GADDR(0,0,menu_adr%)   ! Get menu address
~MENU_BAR(menu_adr%,1)       ! Display menu bar
'
' Reserve a Message Buffer of 16 bytes
' and give variable names to positions within it:
DIM message_buffer&(7)  ! Eight words (including 0)
mesg_adr%=V:message_buffer&(0)
ABSOLUTE mesg_type&,mesg_adr%
ABSOLUTE m_title&,mesg_adr%+6
ABSOLUTE m_item&,mesg_adr%+8
'
REPEAT
ev%=EVNT_MULTI(&X110000,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                                        mesg_adr%,500)
  IF BTST(ev%,4)          ! a new message has arrived
    IF mesg_type&=10      ! reporting a menu event
      title$=CHAR{OB_SPEC(menu_adr%,m_title&)}
'                         ! find out what it was
      item$=CHAR{OB_SPEC(menu_adr%,m_item&)}
      PRINT AT(3,20);"Menu title: ";title$;SPC(10)
      PRINT AT(3,21);"Menu item : ";item$;SPC(10)
      ~MENU_TNORMAL(menu_adr%,m_title&,1)
'                       ! Un-invert menu title display
    ENDIF
  ENDIF
UNTIL MOUSEK=2
```

```
~MENU_BAR(menu_adr%,0)    ! Remove menu bar
~RSRC_FREE()              ! Release Resource memory
RESERVE        ! Return reserved memory to GFA BASIC
END
```

--> At the beginning of the program a 33 k byte area of memory is reserved. It is returned to the GFA BASIC interpreter at the end of the program.

The Resource file is then loaded into the freed memory area, the address of the menu tree is determined and the menu is displayed.

EVNT_MULTI supervises the menu tree, and, in the event that a menu item is selected, appropriate messages are written into the Message buffer. This is a 16-byte long area of memory, divided into eight 2-byte words, conveniently allocated by means of a word-sized array. Using the ABSOLUTE command, some variables are defined as being located in this array, enabling elements of it to be referred to by name.

In a REPEAT-UNTIL loop (exited by pressing the right mouse button), EVNT_MULTI is called. The first word of the Message buffer (defined as mesg_type&) then contains the value 10 if a menu item was selected. The fourth word (m_title&, the 6th and 7th bytes) contains the object number of the menu title, from beneath which the item was chosen. The object number of the item is in the fifth word (m_item&, bytes 8 and 9). The other elements of the buffer are not required here.

Knowing the object numbers of the menu title and the menu item, and that OB_SPEC(...), in the case of Dialog Box Buttons and Menu items, returns the address of text terminated with a zero byte, CHAR can be used to extract the texts and put them into strings.

After displaying the menu title and item on the screen, the inverted menu title is returned to its normal state with MENU_TNORMAL.

```
' ** Window demonstration
' WIND_SMP.GFA
'
DEFFILL 1,2,4
PBOX 0,19,639,399
DEFFILL 1,0
'
DIM message_buffer&(7)                ! 16 Bytes
adr_mes%=V:message_buffer&(0)
'
ABSOLUTE word0&,adr_mes%
ABSOLUTE x&,adr_mes%+8
ABSOLUTE y&,adr_mes%+10
ABSOLUTE w&,adr_mes%+12
ABSOLUTE h&,adr_mes%+14
'
handle&=WIND_CREATE(&X101111,0,19,639,380)
'
title$="Window"
adr_title%=V:title$
~WIND_SET(handle&,2,CARD(SWAP(adr_title%)),
CARD(adr_title%),0,0)
~WIND_OPEN(handle&,100,100,200,100)
~WIND_GET(handle&,4,wx&,wy&,ww&,wh&)
PBOX wx&,wy&,wx&+ww&,wy&+wh&
'
finish!=FALSE
REPEAT
'
  ~EVNT_MULTI(&X110000,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    adr_mes%,100,d%,d%,d%,d%,d%,d%)
  SELECT word0&
    ' Depending on word0&,
    ' one of the following cases is dealt with:
  CASE 22           ! WM_CLOSED - closed
    finish!=TRUE
  CASE 23           ! WM_FULLED - full size
    ~WIND_SET(handle&,5,1,19,638,380)
    ~WIND_GET(handle&,4,wx&,wy&,ww&,wh&)
    PBOX wx&,wy&,wx&+ww&,wy&+wh&
```

```
    word0&=0
  CASE 27,28        ! WM_SIZED,WM_MOVED - re-sized or
moved
    IF w&<100
      w&=100
    ENDIF
    IF h&<80
      h&=80
    ENDIF
    ~WIND_SET(handle&,5,x&,y&,w&,h&)
    ~WIND_GET(handle&,4,wx&,wy&,ww&,wh&)
    PBOX wx&,wy&,wx&+ww&,wy&+wh&
    word0&=0
  ENDSELECT
UNTIL finish!
~WIND_CLOSE(handle&)
~WIND_DELETE(handle&)
```

--> First a Message Buffer is allocated, as in the example program MENU_SMP.GFA, and some words within it are given variable names.

Then the window is created and its handle number determined with WIND_CREATE, and a title is assigned to it with WIND_SET.

After opening the window with WIND_OPEN the coordinates of the work area of the window are found with WIND_GET, and the whole area is covered with a white rectangle.

In the REPEAT-UNTIL loop, control of the window elements is supervised by EVNT_MULTI. Different events (signalled by word0&, the first word in the message buffer) are dealt with by the associated CASE selection. If the 'Close Window' symbol is clicked, the loop terminates and so does the program, after closing the window and deleting it from memory.

Chapter 12
Appendix

Compatibility with GFA BASIC 2

It is possible in GFA BASIC 3 to use programs from older GFA BASIC versions. For this purpose, programs written in the earlier version must be stored as ASCII files, using the SAVE,A command. They can then be loaded into Version 3 by means of MERGE and, thereafter, treated as GFA BASIC 3 programs being SAVEd and LOADed in the normal way.

Version 3 contains all the commands which are in earlier versions of the BASIC interpreter, although there are some small differences in the interpretation of commands which may make some modification of earlier programs necessary.

MUL, DIV

The commands MUL and DIV work in GFA BASIC 3 with integer variables (I,&,%) and integer parameters only.

In the earlier versions, the program:

```
a%=10
MUL a%,2.5
PRINT a%
```

produced the output 25.

In Version 3, however, MUL does not take account of the fractional parts of inputs, treating them all as integers. Thus, the value to the right of the decimal point is ignored, so, in the above example, the integer a% is set to 20. This incompatibility between versions is the price paid for these commands being many times faster than their counterparts in the earlier versions of GFA BASIC, the increase in speed coming from the use of true integer arithmetic. This incompatibility does, of course, only apply to integer variables.

PRINT USING

PRINT USING displays only the numbers that fit in the designated format. This can mean that it is possible that wrong values will be displayed, instead of wrong formats as in earlier versions, if the length of the number is too great. The actual accuracy is, of course, not affected just the display. Care must therefore be taken when using this command.

CLS - PRINT TAB

At CLS, the string ESC-E-CR is now issued, so that the PRINT command will be able to treat TAB correctly.

KEYPAD

A program, which queries the keys of the numeric key-pads and Alternate and/or Control, requires that a KEYPAD 0 is used to switch this off now. The same is valid for the function keys with Alternate.

MOUSEX - MOUSEY

If windows are active and MOUSEX/MOUSEY are interrogated, then negative coordinates arise above and to the left of the window border (e.g. CLIP OFFSET). In Version 2.X CARD(MOUSEX) or CARD(MOUSEY) returned these values.

Programs written to run in Interpreted mode can be run by means of the Run-only Interpreter which is supplied with the GFA BASIC Interpreter. This contains all the routines necessary for the running of GFA BASIC programs and can be distributed by bona fide GFA BASIC owners along with their programs, thus allowing all ST users to run any software written in GFA BASIC. The BASIC Interpreter itself is sold for use by the owner, on one machine and copies may not be passed on to another user.

Programs written in GFA BASIC Version 3 will be capable of being speeded up in operation by the use of the GFA BASIC 3 compiler. This is designed to produce programs which are executable without the need for supporting code, i.e. they are true stand-alone .PRG files which can be distributed freely by the author. Our only request is that an acknowledgement that they were produced using GFA BASIC be included.

GEMDOS Table

~GEMDOS(0) **pterm0()**

Terminates a program. May not be used in GFA BASIC.

r%=GEMDOS(1) **cconin()**

Reads a character from the keyboard, displaying that character on the screen (see INP(2)).

r% Bit 0-7: ASCII Code, 16-23: Scan-code, 24-31: Shift keys

~GEMDOS(2,z%) **cconout()**

Displays characters on the screen (see OUT 2,z%).

z% Bit 0-7: ASCII Code of the character.

r%=GEMDOS(3) **cauxin()**

Reads a character from the serial interface (see INP(1)).

r% Code of the character read

~GEMDOS(4,z%) **cauxout()**

Outputs a character to the serial interface (see OUT 1,z%).

z% Code of the character to be displayed

~GEMDOS(5,z%) **cprnout(**

Outputs a character to the printer interface (see OUT 0,z%).

z% Code of the character to be output

r%=GEMDOS(6,z%) **crawio()**

Writes a character to the screen or, if z%=255, executes an INKEY routine (see OUT 2,z% as well as INKEY$).

r% with z%=255 a character is read from the keyboard
z% Code of output character, with z%=255 then a character is read in.

r%=GEMDOS(7) **crawcin()**

characters are read from keyboard without screen echo (see INP(2)).

r% Code of the character read.

r%=GEMDOS(8) **cnecin()**

Like GEMDOS(7), however control characters, e.g. CTRL+C, are ignored.

r% Code of the character read.

~GEMDOS(9,L:adr%) **cconws()**

Displays a character string on the screen.

adr% Address of the strings, which must end with a zero byte

~GEMDOS(10,L:adr%) **cconrs()**

Reads in a character string from the keyboard (CTRL + c causes crash).

adr% Buffer for string: first byte is number of characters, second byte number of characters to be read, then the string follows.

r! =GEMDOS(11) **cconis()**

Checks whether there is a character in the keyboard buffer.

r! is TRUE if a is character present, otherwise FALSE

~GEMDOS(14,d%) **dsetdrv()**

Changes the current drive; 0=A, 1=B, etc. (see CHDRIVE).

d% Number of the drive

r! =GEMDOS(16) **cconos()**

Checks whether a character can be displayed on the screen.

r! Output is always TRUE while screen is connected

r! =GEMDOS(17) **cprnos()**

Checks whether the parallel interface is ready to receive.

r! Output is TRUE if ready to receive (normally a printer).

r! =GEMDOS(18) **cauxis()**

Checks whether a character is available on the serial interface.

r! TRUE if character available, otherwise FALSE

r! =GEMDOS(19) **cauxos()**

Checks the output status of the serial interface.

r! TRUE if a character can be output, otherwise FALSE.

r%=GEMDOS(25) **dgetdrv()**

Determines the identity of the current drive.

r% Identity of the current drive (A=0, B=1, etc.).

~GEMDOS(26,L:adr%) **fsetdta()**

Sets the disk transfer address (DTA), normally BASEPAGE+128.

adr% The address to be set

r%=GEMDOS(42) **tgetdate()**

Determines the system date (see DATE$).

r% Date: Bit 0-4: Day, 5-8: Month, 9-15: Year minus 1980

~GEMDOS(43,d%) **tsetdate()**

Sets the system date (see SET TIME).

d% The new date (for format see GEMDOS(42))

r%=GEMDOS(44) **tgettime()**

Determines the system time (see TIME$).

r% Time: Bit 0-4: Seconds, 5-10: Minutes, 11-15: Hours

~GEMDOS(45,t%) **tsettime()**

Sets the system time (see SET TIME).

t% The new time (format see GEMDOS(44))

r%=GEMDOS(47) **fgetdta()**

Determines the current disk transfer address (DTA).

r% The address returned.

r%=GEMDOS(48) **sversion()**

Determines the GEMDOS version number.

r% The GEMDOS version number.

~GEMDOS(49,L:b%,r%) **ptermres()**

Terminates the program and reserves b% bytes in BASEPAGE (Cannot be used in GFA BASIC).

b% Number of bytes to be reserved starting from BASEPAGE
r% Value returned to calling program

r%=GEMDOS(54,L:adr%,d%) **dfree()**

Returns information about free disk space on drive d% (see DFREE).

r% -46 when a wrong drive number specified
adr% Address 4 longwords in length with structure:
Long 1: Number of free Clusters
Long 2: Total number of the Cluster
Long 3: Byte per sector
Long 4: Sectors per Cluster

r%=GEMDOS(57,L:adr%) **dcreate()**

Creates a new directory (see MKDIR).

r% -34 or -36 when a fault arises
adr% Address of the new directory name (must end with zero byte)

r%=GEMDOS(58,L:adr%) **ddelete()**

Deletes a directory (see RMDIR).

r% -34, -36 or -65 when a fault arises
adr% Address of the new directory name (must end with zero byte)

r%=GEMDOS(59,L:adr%) **dsetpath()**

Change the current directory.

r% -34 if new directory not found.
adr% Address of directory name (must end with zero byte).

r%=GEMDOS(60,L:adr%,a%) **fcreate()**

Creates a new file (see OPEN "O").

r% -34, -35 or -36 when a fault arises
adr% Address of the file name (must end with zero byte)
a% Bit 0 set: write protected file, Bit 1: hidden file,
Bit 2: System file (also hidden), Bit 3: Disk name.

r%=GEMDOS(61,L:adr%,m%) **fopen()**

Opens a file (see OPEN).

r% -35 or -36 in the event of an error, otherwise r% -33
adr% Address of the file name (must end with zero byte)
m% 0 for reading, 1 for writing, 2 for reading and writing

r%=GEMDOS(62,h%) **fclose()**

Closes the file with the handle h% (see CLOSE).

r% -37 when faults arise
h% File handle of the file to be closed.

r%=GEMDOS(63,h%,L:l%,L:adr%) **fread()**

Reads l% bytes from the file that was opened with handle h% (see BGET).

r% -37 in the event of an error, otherwise number of bytes read
h% Handle
l% Number bytes to be read
adr% Address to which the bytes are to be written

r%=GEMDOS(64,h%,L:l%,L:adr%) **fwrite()**

Writes bytes write into a file (see BPUT).
r% -36 or -37 in the event of an error, else number of the bytes written
h% Handle.
l% Number of bytes to be written.
adr% Address, starting from which, the bytes are to be written in memory

r%=GEMDOS(65,L:adr%) **fdelete()**

Deletes a file (see KILL).

r% -33 or -36 when a fault arises
adr% Address of the name of the file (must end with zero byte).

r%=GEMDOS(66,L: n%,h%,m%) fseek()

Resets the pointer for file accesses (see SEEK, RELSEEK).

h% is handle
r% -32 or -37 when a fault arises
n% Number ofbytes to be jumped over
m% 0: starting from file beginning
1: starting from current position
2: starting from file end

r%=GEMDOS(67,L:adr%,m%,a%) fattrib()

Reads or writes file attributes.

r% -33 or -34 in the event of an error, otherwise file attributes
adr% Address of the file name (must end with zero byte)
m% 0: File attribute read, 1: File attributes write
a% File attributes:
Bit 0: Write protected 1: Hidden
2: System file 3: Disk name
4: Directory 5: Archive bit

r%=GEMDOS(69,h%) fdup()

Produces a second file handle.

r% -35 or -37 in case a fault arose, otherwise second File handle.
h% Original file handle.

r%=GEMDOS(70,h%,nh%) fforce()

Creates a new output handles for GEMDOS.

r% -37 when a fault arises
h% handle of the data channel to be diverted from
nh% handle of the channel to which the outputs are to be diverted to

r%=GEMDOS(71,L:adr%,d%) **dgetpath()**

Determines the current access path of a drive (see DIR$).

r% -46 in case a fault arose
adr% Address, starting from which the access path is stored
d% Drive identification (0=current, 1=A, 2=B, etc.)

r%=GEMDOS(72,L:b%) **malloc()**

Reserve or determine storage location (see MALLOC).

r% for b%=-1 length of the largest free storage area, otherwise Start address of the reserved area or error message
b% Number of bytes to be reserved, for b%=-1, see r%.

r%=GEMDOS(73,L:adr%) **mfree()**

Reserves the memory starting at adr%. This is returned by GEMDOS(72) (see MFREE).

r% -40 when a fault arises
adr% Address of the storage area to be released.

r%=GEMDOS(74,L:adr%,L:b%) **mshrink()**

Shortens reserved memory block, used with GEMDOS(72). Frees all memory starting at address adr% that exceeds b% (see MSHRINK).

r% -40 or -67 when a fault arises
adr% Address of the storage area to be shortened
b% New length of the storage area in bytes

r%=GEMDOS(75,m%,L:p%,L:c%,L:e%) **pexec()**

Executes a program as a subroutine from the disk (see EXEC).
r% -32, -33, -39, or -66 in event of an error.
m% 0: Load and start, 3: Load 4: start: 5: Return basepage
p% Address of the program name or at m%=4 of the Basepage.
c% Address of the command line (not at m%=4)
e% Address of the environment strings (not at m%=4)

~GEMDOS(76,r%) **pterm()**

terminates the current program (see QUIT, SYSTEM).

r% Returns value to the calling program.

r%=GEMDOS(78,L:adr%,a%) **fsfirst()**

Searches the current indices and searches for files with the specified names. The file name found is returned in DTA.

r% -33: File not found, -49: no further files.
adr% Address of the file name (the Wildcards * and ? may be used)
a% File attributes:
Bit 0: Write protected 1: Hidden
2: System file 3: Disk name
4: Directory 5: Archive bit.

r%=GEMDOS(79) **fsnext()**

Continues a search begun with GEMDOS(78).

r% -49 if no further files on which the search string fits

r%=GEMDOS(86.0,L:o%,L:n%) **frename()**

Names a file (see NAME, RENAME).

r% -34 or -36 when a fault arises
o% Address of the old file name.
n% Address of the new file name.

~GEMDOS(87,l:adr%,h%,m%) **fdatime()**

Determines or sets the time and date of a file (see TOUCH).

adr% Address of the time information (4 byte).
h% File handle.
m% 0: Reading of the file time, 1: To set the file time.

BIOS Table

~BIOS(0,L:adr%) **getmpb()**

Initialization of the MEMORY parameter block.

adr%	Address of the new MPB.

r%=BIOS(1,d%) **constat()**

Interrogates an input buffer (see INP?).

r%	0: no character available, -1: Character available
d%	0: Parallel interface 1: Serial interface
	2: Keyboard 3: MIDI interface

r%=BIOS(2,d%) **bconin()**

Reads in a character from an input buffer (see INP).

r%	Character read in (8 bit).
d%	0: parallel interface 1: Serial interface
	2: Keyboard 3: MIDI interface

~BIOS(3,d%,b%) **bconout()**

Outputs a character to a device (see OUT).

d%	0: Parallel interface 1: Serial interface
	2: Keyboard 3: MIDI interface
	4: IKBD
b%	The character to be output

r%=BIOS(4,f%,L:b%,n%,s%,d%) **rwabs()**

Reads and writes sectors to and from disk.

r%	0 if no fault arose
f%=0:	Reads n% sectors starting at sector s%, on drive d% at buffer address b%
f%=1:	Writes the sectors to the disk drive
f%=2:	As f%=0 but ignores media change
f%=3:	As f%=1 but ignores media change
b%	Address of the data storage area
n%	Number of sectors
s%	Number of the start sectors
d%	Number of the drive (0=A, 1=B, etc.)

r%=BIOS(5,n%,L:adr%) **set EXEC()**

Sets and reads the exception vector.

r%	For adr%=-1 the value of the previous vector is returned
n%	Number of the exception vector
adr%	New address of the vector or -1 (see r%)

r%=BIOS(6) **tickcal()**

Interrogation of the system timer.

r% Number of milliseconds passed, with a 20ms resolution

BIOS(7,d%) **getbpb()**

Returns the address of the disk drive parameter block.

r%	Address of the drive parameter block
d%	Number of the drive (0=A, 1=B, etc.)

r! =BIOS(8,d%) **bcostat()**

Returns the state of an output device (see OUT?).

r!	TRUE if able to transmit characters, otherwise FALSE
d%	0: Parallel interface 1: Serial interface 3: MIDI interface 4: Keyboard chip (IKBD).

r%=BIOS(9,d%) **mediach()**

Determines whether a disk was changed.

r%	0: definitely not changed (Hard disk) 1: perhaps has been changed 2: definitely changed
d%	Number of the drive (0=A, 1=B, etc.)

r%=BIOS(10) **drvmap()**

Checks which drives are attached.

r%	Bit pattern with one bit corresponding to each drive (bit 0=A, bit 1=B, etc.). Thus &x10011 means that drives A, B and E are attached

r%=BIOS(11,c%) **kbshift()**

Determines or sets the status of the keyboard shift keys.

r%	For c%=-1 current status of the shift keys is returned
c%	New status: bit 0: Right Shift bit 1: Left Shift bit 2: Control bit 3: Alternate bit 4: Caps-Lock bit 5: Alternate + Clr/Home (right mouse key) bit 6: Alternate + Insert (left mouse key)

XBIOS Table

~XBIOS(0,t%,l:p%,l:v%) **initmous()**

Initialises the mouse handling routine but is not compatible with GEM.

t%	0: Switches mouse off 1: Switches mouse into relative mode 2: Switches mouse into absolute mode 4: Mouse in keyboard mode
p%	Address of an information structure.
v%	Address of the mouse handling routine

r%=XBIOS(2) **physbase()**

Returns the base address of the physical screen memory currently in use.

r%	Address of the physical screen memory

r%=XBIOS(3) **logbase()**

Returns the address of the logical screen memory when writing to the screen.

r%	Address of the logical screen memory.

r%=XBIOS(4) **getrez()**

Returns the current screen resolution.

r%	0: 320x200, 1: 640*200 2: 640x400, 3: reserved for modified ST's.

~XBIOS(5,l:l%,l:p%,r%) **setscreen()**

Enables resolution to be changed between low res and high res when using the colour monitor. This cannot be used with GEM.

The value -1 means that no parameters are to be changed. Addresses must start on a 256 byte boundary only.

l%	New address of the logical screen memory.
p%	New address of the physical screen memory.
r%	New screen resolution (see XBIOS(4))

~XBIOS(6,L:adr%) **setpalette()**

Allows all colour registers to be reset at one time.

adr% Address of a table of 16 words, which contains the new pallet data.

r%=XBIOS(7,n%,c%) **setcolor()**

Sets or interrogates a colour register (see SETCOLOR).

r%	For c%=-1 the previous specified colour register is returned
n%	Number of the colour register (0 to 15)
c%	New colour, at c%=-1 see r%

r%=XBIOS(8,L:b%,L:f%,d%,sec%,t%,side%,n%) **floprd()**

Reads sectors of the disk.

r%	0 if no fault arose
b%	Address of the area, from which the sectors are read
f%	Unused.
d%	Number of the drive (0=A, 1=B, etc.)
sec%	Number of the sector, starting from one read
t%	Number of the trace (TRACK), from which one reads
side%	Side disk (0 or 1)
n%	Number of sectors to be read (must lie on a TRACK)

r%=XBIOS(9,L:b%,L:f%,d%,sec%,t%,side%,n%) **flopwr()**

Writes sectors to a disk.

r%	0 if no fault arose
b%	Address of the storage area to which the data is to be written
f%	Unused
d%	Number of the drive (0=A, 1=B, etc.)
sec%	Number of the start sector, from which writing will commence.
t%	Number of the track in which writing will take place
side%	Side of the disk (0 or 1).
n%	Number of the sectors to be written (all on one TRACK)

r%=XBIOS(10,L:b%,L:f%,d%,sec%,t%,side%,i%,L:m%,v%)
flopfmt()

a trace of the disk formats.

r%	0 if no fault arose.
b%	Address of an area for the intermediate memory (min. 8KB)
f%	Unused
d%	Number of the drive (0=A, 1=B, etc.).
sec%	Sectors per track (normally 9).
t%	Number of the track to be formatted.
side%	Side of the disk (0 or 1).
i%	Interleave factor (normally 1).
m%	Magic-Number &H87654321
v%	Value in sectors after formatting (normally &HE5E5)

~XBIOS(12,n%,L:adr%) **midiws()**

Outputs the contents of a block of memory to the MIDI interface.

n%	Number of the bytes minus 1, to be output
adr%	Address of the source storage area

~XBIOS(13,n%,L:adr%) **mfpint()**

Sets the MFP interrupt vector on the ST. This can only be used from assembly language or 'C' and is not available from GFA BASIC.

n%	Number of the interrupt (0 to 15)
adr%	New address of the interrupt

r%=XBIOS(14,d%) **iorec()**

Returns the address of the I/O table used by the serial interface.

r%	Address of the data buffer for the serial interface I/O table
d%	0: RS 232 1: IKBD 2: MIDI

~XBIOS(15,b%,h%,ucr%,rsr%,tsr%,scr%) **rsconf()**

Configures the serial interface. The parameters remain unchanged with a value of -1.

b%	Baud rate
h%	Hand shake mode:
	0: without 1: XON/XOFF
	2: RTS/CTS 3: both
ucr%	USART control register of the MFP
rsr%	Receiver status register of the MFP
tsr%	Transmitter status register of the MFP
scr%	Synchronous character register of the MFP

r%=XBIOS(16,L:us%,L:sh%,L:cl%) **keytbl()**

Changes the keyboard translation tables.

r%	Address of the KEYTAB structure
us%	Address of the table for keys without shift
sh%	Address of the table for keys with shift
cl%	Address of the table for keys with Caps-Lock

r%=XBIOS(17) **random()**

Returns a random number (see RAND, RANDOM).

r%	Number returned with 24 bit accuracy (from 0 to 16777215)

~XBIOS(18,L:b%,L:s%,d%,f%) **protobt()**

Creates a boot sector for the disk in memory.

b%	Address of a 512 byte buffer for the producing of the Boot sector
s%	Serial number that forms part of the boot sector: -1: previous serial number retained >24 bits: Random number returned
d%	Disk type (tracks/sides) 0: 40 tracks, single sided (180K) 1: 40 tracks, double sided (360K, IBM) 2: 80 tracks, single sided (360K, SF340) 3: 80 tracks, double sided (720K, SF314)
f%	0: non-executable Bootsector 1: executable -1: leave unchanged

r%=XBIOS(19,L:b%,L:f%,d%,sec%,t%,side%,n%) **flopver()**

Verifies the disk contents.

b%	Address of the memory area with which a comparison is made
f%	Unused
d%	Number of the disk drive (0=A, 1=B, etc.)
sec%	Number of the start sector, from which comparison made
t%	Number of the track
side%	Side of the disk (0 or 1)
n%	Number of the sectors to be compared

~XBIOS(20) **scrdmp()**

Calls the hardcopy routine and, thus, dumps the screen to printer. (See Hardcopy.)

r%=XBIOS(21,c%,s%) **curscon()**

Configure cursor.

r%	when c%=5 returns the cursor blink rate
c%	0: Hide cursor 1: Show cursors 2: Blinking cursor 3: Solid cursor 4: Blink rate set to value in s% 5: see r%.
s%	when c%=4, blink rate set to s%

~XBIOS(22,L:t%) **bsettime()**

Sets date and time (see SET TIME).

t% Bit 0-4: Seconds, 5-10: Minutes, 11-15: Hours, 16-20: Day, 21-24: Month, 25-31: Year minus 1980.

r%=XBIOS(23) **bgettime()**

Returns date and time (see TIME$, DATE$).

r%	For bit allocation see XBIOS(22).

~XBIOS(24) **bioskey()**

Re-installs the original keyboard allocation (see XBIOS(16)).

XBIOS(25,n%,L:adr%) **ikbdws()**

Writes bytes from memory to the keyboard processor (IKBD).

n%	Number bytes minus 1, to be sent
adr%	Address where the data to be sent is stored

~XBIOS(26,i%) **jdisint()**

Disables an MFP interrupt.

i%	Number of the interrupt (0-15) to be disabled.

~XBIOS(27,i%) **jenabint()**

Enables an interrupt of the MFP.

i%	Number of the interrupt

r%=XBIOS(28,d%,reg%) **giaccess()**

Reads and writes from and to the sound chip register.

r%	Returns register value when reading
d%	The value to be written when writing (8 bit)
r%	Register number (0 to 15), bit 7 defines write mode when set

~XBIOS(29,b%) **offgibit()**

Sets the bit of port A on the register of the sound chip to zero.

b% Bit pattern, which is ORed with the existing contents

~XBIOS(30,b%) **ongibit()**

Sets the port A bit of the sound chip register to 1

b%	Bit pattern, which is ANDed with the existing contents

XBIOS(31,t%,c%,d%,L:adr%) **xbtimer()**

Set the MFP timers.

t%	Number of the timer (0 to 3)
c%	Control register
d%	Data register
adr%	Address of the timer interrupt routine

~XBIOS(32,L:adr%) **dosound()**

Starts a sound sequence, which is processed in the interrupt

adr%	Address of the storage area order

r%=XBIOS(33,c%) **setprt()**

Sets or reads the printer parameters.

r% Current configuration, when c%=-1

c%

Bit	set	reset	
0	Dot matrix	daisy wheel	printer
1	Monochrome	colour	printer
2	Atari	Epson	printer
3	Test	NLQ	mode
4	Parallel	RS-232	port
5	Continuous	single sheet	paper

r%=XBIOS(34) **kbdvbas()**

Returns address of table with vectors to the keyboard and MIDI processor.

r% Returned address

r%=XBIOS(35,a%,w%) **kbrate()**

Sets and reads keyboard repeat rate.

r% Current data, bit 0-7: repeat rate, 8-15: Time of repeat delay
a% Repeat delay
w% Repeat rate

~XBIOS(36,L:adr%) **prtblk()**

Hardcopy routine, returns parameter block address

adr% Address of a parameter block for the hardcopy routine

~XBIOS(37) **vsync()**

Waits for the next vertical blank interrupt (see VSYNC).

~XBIOS(38,L:adr%) **supexec()**

Calls an assembler routine in supervisor mode (without GEMDOS system calls)

adr% Address of the assembler routine

~XBIOS(39) **puntaes()**

Turns off AES if it is not in the ROM.

r%=XBIOS(64,b%) **blitmode()**

Controls and interrogates the Blitter (only in the Blitter TOS).

r% Current blitter-status, if b%=-1, bit 1: Blitter there?. b% -1: see r%, otherwise bit 0: set Blitter, otherwise Blitter out, bit 1-14: reserved (-1), bit 15: 0.

Table of LINE-A Variables

The base addresses of the line A variables are returned by means of L~A:

{L~A-906}	Address of the current Font-header
L~A-856.	37 Words, active DEFMOUSE
{L~A-460}	Pointer to the current Font-headers
L~A-456	Array from four pointers, of which the last must be zero.

Each pointer points to a concatenated list of character sets. The first two pointers are valid for resident Fonts. Third is for the GDOS-Fonts, of this with each VDI-CALL is put back.

INT{L~A-440}	Total number of this Fonts
INT{L~A-46}	Text line height
INT{L~A-44}	Maximum splits
INT{L~A-42}	Maximum cursor line
INT{L~A-40}	Length of a text line in bytes
INT{L~A-38}	Text background colour
INT{L~A-36}	Text foreground colour
{L~A-34}	Address of the cursor in screen memory INT{la-30}
	Distance of first text line from the upper screen edge
INT{la-28}	CRSCOL
INT{L~A-26}	CRSLIN
BYTE{L~A-24}	Cursor blink period
BYTE{L~A-23}	Cursor blink count
{L~A-22}	Address of the data for the mode
INT{L~A-18}	Last ASCII character of the font
INT{L~A-16}	First ASCII character of the font
INT{L~A-12}	Horizontal resolution in pixels
{L~A-10}	Address of the table
INT{L~A-4$}	Vertical resolution in pixels
INT{L~A-0}	Number of bit planes
INT{L~A+2}	Number of bytes per screen line
{L~A+4}	Pointer for the CONTRL field
{L~A+8}	Pointer for the INTIN field
{L~A+12}	Pointer for the PTSIN field
{L~A+16}	Pointer for the INTOUT field
{L~A+20}	Pointer for the PTSOUT field
INT{L~A+24}	Colour value for bit level 0

INT{L~A+26}	Colour value for bit level 1
INT{L~A+28}	Colour value for bit level 2
INT{L~A+30}	Colour value for bit level 3
INT{L~A+32}	Flag, do not draw last pixel of a line
INT{L~A+34}	Line pattern
INT{L~A+36}	Graph mode
INT{L~A+38}	until
INT{L~A+44}	2 coordinate pairs
{L~A+46}	Pointer to current fill pattern
{L~A+50}	Pointer to the current fill pattern mask
INT{L~A+52}	Flag for multi-coloured fill pattern
INT{L~A+54}	Clipping-flag
INT{L~A+56}	until
INT{L~A+64}	Clipping-coordinates
INT{L~A+66}	Enlargement factor
INT{L~A+68}	Enlargement direction
INT{L~A+70}	Flag for proportional script
INT{L~A+72}	x-Offset for Textblt
INT{L~A+74}	y-Offset for Textblt
INT{L~A+76}	x-coordinate of a character on the screen INT{L~A+78} y-coordinate of a character on the screen
INT{L~A+80}	Width of a character
INT{L~A+82}	Height of a character
{L~A+84}	Pointer to character set image
{L~A+88}	Width of character set image
INT{L~A+90}	Text style
INT{L~A+92}	Mask for shaded text output
INT{L~A+94}	Mask for italic script
INT{L~A+96}	Additional width for wide script
INT{L~A+98}	Italic script offset on the right
INT{L~A+100}	Italic script offset on the left
INT{L~A+102}	Enlargement flag
INT{L~A+104}	Text rotation angle
INT{L~A+106}	Text colour
{L~A+108}	Pointer on buffer for text effects
INT{L~A+112}	Offset for a second text effects buffer
INT{L~A+114}	Colour of the text background
INT{L~A+116}	Flag for copy screen form, <>0 for transparent
{ L~A+118}	Pointer to routine, which terminates filling procedure - with V 3 by means of Shift-Alternate-Control.

Table of Input Parameters for V_OPN(v) WK

In addition to the values given below, the following default values are used:
x Device identification number (standard)

1. : Screen	11. : Plotter
21. : Printer	31. : Metafile
41. : Camera	51. : Graphic tablet

1 Line type
1 Line colour
1 Mark type
1 Mark colour
1 Text style
1 Text colour
1 Fill type
1 Fill style
1 Fill colour
2 Coordinate system (0: NDC, 1: reserved, 2: RC)

Table of WORK_OUT Array of the VDI

With V_OPN(v)WK, the resultant values are returned in INTOUT(0) to INTOUT(44) and in PTSOUT(0) to PTSOUT(11).

WORK_OUT(0)	Maximum picture width in pixels
WORK_OUT(1)	Maximum picture height in pixels
WORK_OUT(2)	0: Exact screen memory possible, 1: Not possible
WORK_OUT(3)	Width of a pixel in micrometer
WORK_OUT(4)	Height of a pixel in micrometer
WORK_OUT(5)	Number of character heights (0: Modificable)
WORK_OUT(6)	Number of line types
WORK_OUT(7)	Number line widths (0: Modifiable).
WORK_OUT(8)	Number of mark symbols
WORK_OUT(9)	Number of mark symbol sizes (0: Modificable)
WORK_OUT(10)	Number of character sets
WORK_OUT(11)	Number of patterns
WORK_OUT(12)	Number of hatching patterns
WORK_OUT(13)	Number of pre-defined colours
WORK_OUT(14)	Number of basic graphic functions (GDP)
WORK_OUT(15)	until
WORK_OUT(24)	List of basic graphic functions (GDP) Ten basic functions are supported:

1: bar 2: arc 3: pie 4: circle 5: ellipse
6: elliptical arc 7: elliptical pie 8:rounded rectangle
9: filled rounded rectangle 10: justified graphic text
The end of this list is marked with a -1.

WORK_OUT(25)	until
WORK_OUT(34)	List of attributes of basic graphic functions:

0: Line 1: Marker 2: Text 3: Filled out area 4: No attribute

WORK_OUT(35)	0: Colours are not representable 1: Are representable

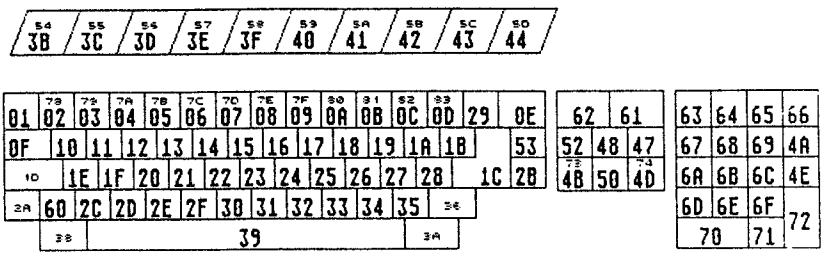
WORK_OUT(36)	0: Text cannot be rotated 1: Can be rotated
WORK_OUT(37)	0: Fill functions are not possible 1: Fill functions are Possible
WORK_OUT(38)	0: CELLARRAY unavailable 1: CELLARRAY available
WORK_OUT(39)	Number of representable colours, 0: More than 32767 2: Monochrome 512: Medium res. or Low res.
WORK_OUT(40)	1: Graphic cursor positioning only with keyboard 2: With keyboard and mouse
WORK_OUT(41)	Device input value: 1: Keyboard 2: Other device
WORK_OUT(42)	Selection keys: 1: Function keys 2 : O t h e r
WORK_OUT(43)	Number of the string input device: 1: Keyboard
WORK_OUT(44)	Work station type: 0: Only output 1: Only input 2: In/output, 3: Reserved 4: Metafile
WORK_OUT(45)	Minimum character width
WORK_OUT(46)	Minimum character height.
WORK_OUT(47)	Maximum character width.
WORK_OUT(48)	Maximum character height.
WORK_OUT(49)	Minimum visible line width.
WORK_OUT(50)	Reserved, always 0.
WORK_OUT(51)	Maximum line width in x-direction.
WORK_OUT(52)	Reserved, always 0.
WORK_OUT(53)	Minimum mark width.
WORK_OUT(54)	Minimum mark height.
WORK_OUT(55)	Maximum mark width.
WORK_OUT(56)	Maximum mark height.

Table of VT 52 Control Codes

The St contains a VT 52 emulator which is modeled on a widely-used terminal and by means of this, can be used for screens that do not contain windows. The routines of this emulator can be called by the output of the strings given in the following table by means of PRINT and these all start with the ESC code (CHR$(27)).

CHR$(27)+"A"; Cursor moves up one line (stops at the upper edge of the screen)
CHR$(27)+"B"; Cursor moves down one line (stops at the lower edge of the screen)
CHR$(27)+"C"; Cursor moves to the right (stops at the right-hand side)
CHR$(27)+"D"; Cursor moves to the left (stops at the left-hand side)
CHR$(27)+"E"; Clear screen (CLS)
CHR$(27)+"H"; Cursor moves to Home position (see LOCATE 1,1)
CHR$(27)+"I"; Cursor moves up one line and scrolls at the upper edge
CHR$(27)+"J"; Erases from cursor to the end of page
CHR$(27)+"K"; Erases from cursor to the end of line
CHR$(27)+"L"; Inserts blank line at cursor position
CHR$(27)+"M"; Deletes line at cursor position (lines below scroll up)
CHR$(27)+"Y"+CHR$(s+32)+CHR$(z+32); corresponds to
LOCATE z,s. where s=chr$(row+32) z=chr$(column+32)
CHR$(27)+"b"+CHR$(f); Selects f as text writing colour
CHR$(27)+"c"+CHR$(f); Selects f as a background colour
CHR$(27)+"d"; Erase from top of page to cursor
CHR$(27)+"e"; Enable cursor
CHR$(27)+"f"; Disable cursors
CHR$(27)+"j"; Store cursor position
CHR$(27)+"k"; Restore cursor to position stored with ESC j
CHR$(27)+"l"; Erase line in which the cursor lies
CHR$(27)+"o"; Erase line from beginning to cursor position
CHR$(27)+"p"; Select reverse video
CHR$(27)+"q"; Switch off reverse video
CHR$(27)+"v"; Switch on word wrap at line end
CHR$(27)+"w"; Switch off word wrap at line end

Scan-code Table



ASCII Table

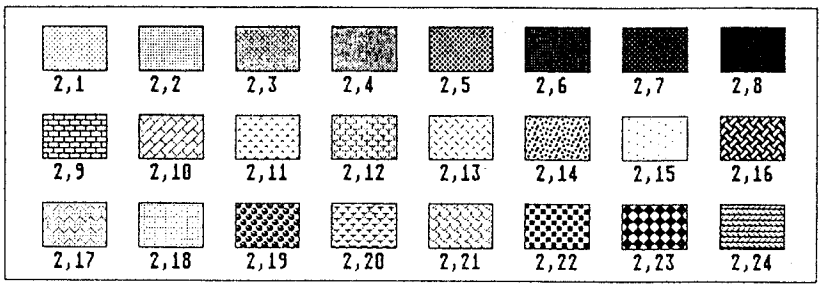

0 1 2 3 4 5 6 7 8 9 A B C D E F
0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
2 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
3 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
4 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
5 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
6 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111
7 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
8 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
9 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
A 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175
B 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
C 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
D 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223
E 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
F 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255

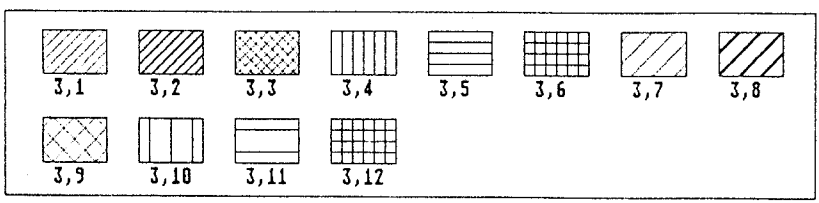
Special ASCII Characters

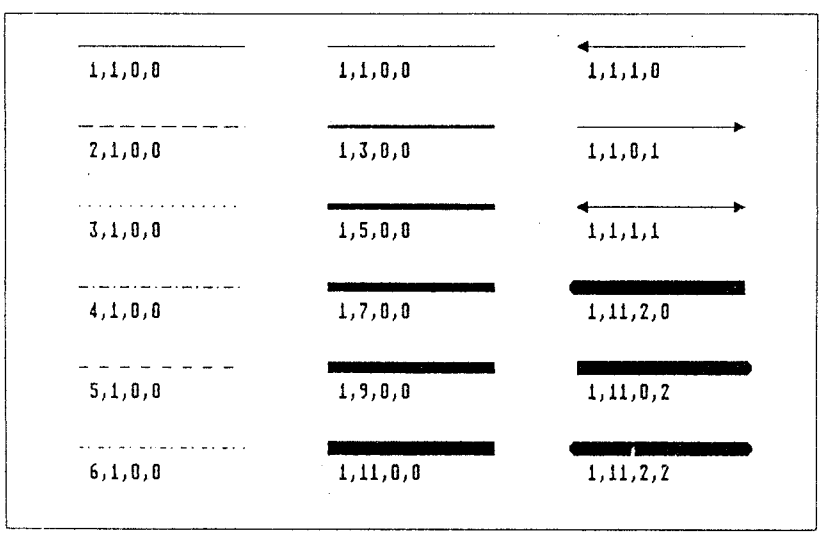
0	NUL	1	SOH	2	STX	3	ETX
4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	LF	11	VT
12	FF	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3
20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC
28	FS	29	GS	30	RS	31	US
127	DEL						

ä	Ä	á	à	â	À
132	142	160	133	131	182
ë		é	è	ê	É
137		130	138	136	144
ï		í	ì	î	
139		161	141	140	
ö	Ö	ó	ò	ô	
148	153	162	149	147	
ü	Ü	ú	ù	û	
129	154	163	151	150	

Fill Pattern and Line Style Table







Error Messages

GFA BASIC Error Messages

0 Division by zero
1 Overflow
2 Not Integer -2147483648 .. 2147483647
3 Not Byte 0 .. 255
4 Not Word -32768 .. 32767
5 Square root only for positive numbers
6 Logarithm only for numbers greater than zero
7 Undefined error
8 Out of memory
9 Function or command not yet implemented
10 String too long max. 32767 characters
11 Not GFA-BASIC 3.00 program
12 Program too long memory full NEW
13 Not GFA-BASIC program file too short NEW
14 Array dimensioned twice
15 Array not dimensioned
16 Array index too large
17 Dim index too large
18 Wrong number of indices
19 Procedure not found
20 Label not found
21 On Open only "I"nput "O"utput "R"andom "A"ppend "U"pdate allowed
22 File already open
23 File # wrong
24 File not open
25 Input wrong not numeric
26 End of file reached
27 Too many points for Polyline/Polyfill/8polymark max. 128
28 Array must have one dimension
29 Number of points too large for array
30 Merge - Not an ASCII file
31 Merge - Line too long aborted

32 ==> syntax error program aborted
33 Undefined label
34 Out of data
35 Data not numeric
36 Undefined error 036
37 Disk full
38 Command not allowed in direct mode
39 Program error Gosub not possible
40 Clear not allowed in For-Next-loops or procedures
41 Cont not possible
42 Parameter missing
43 Expression too complex
44 Undefined function
45 Too many parameters
46 Parameter wrong must be a number
47 Parameter wrong must be a string
48 Open "R" Record length wrong
49 Too many "R"-files (max 31)
50 Not an "R"-File
52 Fields larger than record length
54 GET/PUT Field string length changed
55 GET/PUT record number wrong
60 Sprite string length wrong
61 Error while RESERVE
62 MENU error
63 RESERVE error
64 Pointer (*x) error
65 Array too small (<256)
66 No VAR-Array
67 ASIN/ACOS error
68 VAR-Type mismatch
69 ENDFUNC without RETURN
71 Index too large
90 LOCAL error
91 FOR error
92 Resume (next) not possible Fatal, For or Local
93 Stack error

Bomb Error Messages

100 GFA BASIC Version 3.xx copyright 1986-1988 GFA Systemtechnik GmbH
102 2 bombs - bus error peek or poke possibly wrong
103 3 Bombs - address error Odd word address! Possibly at Dpoke, Dpeek, Lpoke or Lpeek
104 4 Bombs - illegal instruction executed in machine code
105 5 bombs - divide by zero in 68000 Machine Code
106 6 bombs - CHK exception 68000 interrupted by CHK
107 7 Bombs - TRAPV exception 68000 interrupted by TRAPV
108 8 Bombs - privilege violation by 68000 Machine Code
109 9 bombs - trace exception

TOS Error Messages

-1 * General error
-2 * Drive not ready
-3 * Unknown command
-4 * CRC error disk check sum wrong
-5 * Bad request
-6 * Seek error track not found
-7 * Unknown media boot sector wrong
-8 * Sector not found
-9 * Out of paper
-10 * Write fault
-11 * Read fault
-12 * General error 12
-13 * Write protected
-14 * Media change detected
-15 * Unknown device
-16 * Bad sector (verify)
-17 * Insert other disk (request)
-32 * Invalid function number
-33 * File not found
-34 * Path not found
-35 * Too many open files
-36 * Access denied
-37 * Invalid 1andle
-39 * Out of memory
-40 * Invalid memory block address
-46 * Invalid drive specification
-49 * No more files
-64 * GEMDOS range error seek wrong?
-65 * GEMDOS internal error
-66 * Invalid executable file format
-67 * Memory block growth failure

Editor Error Messages

Case without Select
Select without endselect
While without Wend
Repeat without Until
Do without loop
For without Next
Wend without While
Until without Repeat
Loop without Do
NEXT without For
If without Endif
Endif without If
Else without If
Else without Endif
Exit without loop
Procedure without Return
Procedure in loop
Procedure defined twice
Function without Endfunc
Function in loop
Function defined twice
Return without Procedure
Mark defined twice
Local only in Procedure
Local not in loop
Function defined twice
Goto into/outof For-Next, Procedure or Function
Resume in For-Next loop
Resume without Procedure
No Resume in Function
Endfunc without Function

Chapter 13
New Features in GFA BASIC 3.5

1. The editor uses two more bytes per program now than the old editor (Versions 3.0 to 3.07). This accelerates the "backward scrolling" and makes it possible to fold functions, too.

2. You can now also use the "Search" function to search in the header rows of closed procedures or functions.

3. Listing now prints "labels" two characters to the left (as with CASE).

4. Tab functions

Tab	**Cursor jumps to next tabulator position.**
Ctrl+Tab	**Cursor jumps the last previous tab position.**
LeftShift+Tab	**Inserts blank spaces to the next tab position.**
RightShift+Tab	**Deletes all blank spaces in one row up to or from cursor.**

Linear operations with vectors and matrix

All functions described in this chapter relate only to one and/or two-dimensional fields with floating point variables.

System commands

MAT BASE 0

MAT BASE 1

The MAT BASE command can only sensibly be used when OPTION BASE 0 has been activated. In this case, MAT BASE 1 can be used to set the offset for the start of the row and column indexing of one or two-dimensional fields with floating point variables to 1 for the matrix operations. MAT BASE 0 resets this offset to 0 after a MAT BASE 1.

Abbreviation: m b 0/m b 1

The setting made with MAT BASE n affects the following commands

MAT READ
MAT PRINT
MAT CPY
MAT XCPY
MAT ADD
MAT SUB
MAT MUL

The default is MAT BASE 1.

Example:

```
OPTION BASE 0
MAT BASE 1
DATA 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
DIM a(3,3)
MAT READ a()
PRINT a(1,1)
```

Outputs the value 1

Generating commands

MAT CLR a()
MAT SET a()=x
MAT ONE a()

a: Name of field with numeric variables
x: aexp

MAT CLR a() corresponds to an ARRAYFILL a(),0, i.e. the command sets all elements in the field (matrix or vector) a() to a value of 0.

Abbreviation: m cl a()

MAT SET a()=x corresponds to an ARRAYFILL a(),x, i.e. the command sets all elements in the field a() (matrix or vector) to the value x.

Abbreviation: m se a()=x

MAT ONE a() generates from a square matrix a() a uniform matrix, i.e. a square matrix in which elements a(1,1),a(2,2),...,a(n.n) are all equally 1 and all other elements equally 0.

Abbreviation: m o a()

Example:

```
DATA 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
DIM a(3,3)
MAT READ a()
PRINT a(1,1)
MAT CLR a()
PRINT a(1,1)
```

Outputs the value 1, then 0.

```
DIM a(5,7)
FOR i%=1 TO 5
     FOR j%=1 TO 7
          a(i%,j%)=RAND(10)
     NEXT j%
NEXT i%
MAT SET a(),5.3
FOR i%=1 TO 5
     FOR j%=1 TO 7
          PRINT a(i%,j%)
     NEXT j%
NEXT i%
```

Outputs the value 3.5 35 times.

```
DIM a(3,3)
MAT ONE a()
MAT PRINT a()
```

Gives:

```
     1,0,0
     0,1,0
     0,0,1
```

Write and Read commands

MAT READ a()
MAT PRINT [#i]a[,g,n]
MAT INPUT #i,a()

i,g,n: **iexp**
a: **Name of field with numerical variables**

MAT READ a() reads a previously dimensioned matrix or vector from DATA rows.

Abbreviation: m r a()

Example:

```
DATA 1,2,3,4,5,6,7,8,9,10
DIM a(2,5)
MAT READ a()
PRINT a(2,4)
```

outputs the value 9.

MAT PRINT [#i,]a()[,g,n] outputs a matrix or a vector. Vectors are output on one row, the elements being separated by commas. With matrix, each row is followed by a rowfeed.

The output can optionally be redirected with #i, as with PRINT.

If g and n are specified, the numbers are formatted as with STR$(x,g,n).

Abbreviation: m p [#i]a()[g,n] or m ? [#i,]a()[g,n]

Example:

```
DATA 1,2.33333,3
DATA 7,5.25873,9.376
DATA 3.23,7.2,8.999
DIM a(3,3)
```

```
MAT READ a()
MAT PRINT a()
PRINT"------------------------"
MAT PRINT a(),7,3
```

Gives:

```
 1,       2.33333,  3
 7,       5.25873,  9.376
 3.23,    7.2,      8.999
 ------------------------
 1.000,   2.333,    3.000
 7.000,   5.259,    9.376
 3.230,   7.200,    8.999
```

MAT INPUT #1,a() reads a matrix or vector from a file in ASCII format (the format being the reverse of MAT PRINT, commas and rowfeeds may be varied as with INPUT #).

Abbreviation: m i #i,a().

Example:

```
OPEN "o",#1,"Test.DAT"
DIM a(3,3)
MAT ONE a()
MAT PRINT #1,a()
CLOSE #1
MAT CLR a()
OPEN "i",#1,"Test.DAT"
MAT INPUT #1,a()
CLOSE #1
MAT PRINT a()
```

Gives:

```
1,0,0
0,1,0
0,0,1
```

Copy and Transposition commands

MAT CPY a([i,j])=b([k,l])[,h,w]
MAT XCPY a([i,j])=b([k,l])[,h,w]
MAT TRANS a()[=b()]

a,b: Name of fields with numerical variables
i,j,k,l,h,w: iexp

MAT CPY a([i,j])=b([k,l])[,h,w] copies h rows with w elements each from matrix b to the row and column offset of matrix a defined by i,j, starting from the row and column offset of matrix b defined by l,k.

Abbreviation: m c a(i,j)=b(k,l),h,w,
m x a(i,j)=b(k,l),h,w,
m t a()=b().

Example:

```
DIM a(5,5),b(4,4)
MAT SET a()=1
FOR i%=1 TO 4
     FOR j%=1 TO 4
          b(i%,j%)=SUCC(i%)
     NEXT j%
NEXT i%
'
MAT PRINT a()
PRINT "--------"
MAT PRINT b()
PRINT "--------"
MAT CPY a(2,2)=b(2,2),3,3
MAT PRINT a()
```

Gives:

```
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
-----------
2,2,2,2
3,3,3,3
4,4,4,4
5,5,5,5
-----------
1,1,1,1,1
1,3,3,3,1
1,4,4,4,1
1,5,5,5,1
1,1,1,1,1
```

Special cases

MAT COPY a()=b() copies the complete matrix b into matrix a if the matrix are of the same order.

Only those elements are copied in this process for which identical indices are given in both the source and the destination matrix.

Abbreviation: m c a()=b().

Example:

```
DIM a(5,3),b(4,4)
MAT SET a()=1
FOR i%=1 TO 4
     FOR j%=1 TO 4
     b(i%,j%)=SUCC(i%)
     NEXT j%
NEXT i%
'
MAT PRINT a()
```

```
PRINT "-------"
MAT PRINT b()
PRINT "-------"
MAT CPY a()=b(),3,3
MAT PRINT a()
```

Gives:

```
      1,1,1
      1,1,1
      1,1,1
      1,1,1
      1,1,1
      -------
      2,2,2,2
      3,3,3,3
      4,4,4,4
      5,5,5,5
      -------
      2,2,2
      3,3,3
      4,4,4
      1,1,1
      1,1,1
```

MAT COPY a(i,j)=b() copies matrix b, starting from the row and column offset defined by MAT BASE, to the row and column offset of matrix a defined by i,j. Only those elements are copied for which identical indices are given in both the source and the destination matrix.

Abbreviation: m c a(i,j)=b()

Example:

```
DIM a(5,3),b(4,4)
MAT SET a()=1
FOR i%=1 TO 4
      FOR j%=1 TO 4
            b(i%,j%)=SUCC(i%)
      NEXT j%
NEXT i%
'
```

```
MAT PRINT a()
PRINT "------"
MAT PRINT b()
PRINT "------"
MAT CPY a(2,2)=b(2,2),3,3
MAT PRINT a()
```

Gives:

```
 1,1,1
 1,1,1
 1,1,1
 1,1,1
 1,1,1
------
 2,2,2,2
 3,3,3,3
 4,4,4,4
 5,5,5,5
------
 1,1,1
 1,3,3
 1,4,4
 1,5,5
 1,1,1
```

MAT COPY a()=b(i,j) copies matrix b, starting from the row and column offset defined by i,j, to the offset of matrix a defined by MAT BASE. Only those elements are copied for which identical indices are given in both the source and the destination matrix.

Abbreviation: m c a()=b(i,j).

Example:

```
DIM a(5,3),b(4,4)
MAT SET a()=1
FOR i%=1 TO 4
     FOR j%=1 TO 4
          b(i%,j%)=SUCC(i%)
     NEXT j%
```

```
NEXT i%
'
MAT PRINT a()
PRINT "-------"
MAT PRINT b()
PRINT "-------"
MAT CPY a()=b(2,2),3,3
MAT PRINT a()
```

Gives:

```
 1,1,1
 1,1,1
 1,1,1
 1,1,1
 1,1,1
 -------
 2,2,2,2
 3,3,3,3
 4,4,4,4
 5,5,5,5
 -------
 3,3,3
 4,4,4
 5,5,5
 1,1,1
 1,1,1
```

MAT COPY a(i,j)=b(k,l) copies matrix b, starting from the row and column offset defined by k,l, to the offset i,j of matrix a. Only those elements are copied for which identical indices are given in both the source and the destination matrix.

Abbreviation: m c a(i,j)=b(k,l).

Example:

```
DIM a(5,3),b(4,4)
MAT SET a()=1
FOR i%=1 TO 4
     FOR j%=1 TO 4
```

```
            b(i%,j%)=SUCC(j%)
        NEXT j%
NEXT i%
'
MAT PRINT a()
PRINT "-------"
MAT PRINT b()
PRINT "-------"
MAT CPY a(2,2)=b(2,2)
MAT PRINT a()
```

Gives:

```
 1,1,1
 1,1,1
 1,1,1
 1,1,1
 1,1,1
 -------
 2,3,4,5
 2,3,4,5
 2,3,4,5
 2,3,4,5
 -------
 1,1,1
 1,3,4
 1,3,4
 1,3,4
 1,1,1
```

MAT COPY a()=b() copies h rows with w elements each from the matrix b, starting from the row and column offset defined by MAT BASE, the row and column offset of matrix a defined by MAT BASE. Only those elements are copied for which identical indices are given in both the source and the destination matrix.

Abbreviation: m c a()=b().

Example:

```
DIM a(5,3),b(4,4)
MAT SET a()=1
FOR i%=1 TO 4
     FOR j%=1 TO 4
          b(i%,j%)=SUCC(j%)
     NEXT j%
NEXT i%
'
MAT PRINT a()
PRINT "------"
MAT PRINT b()
PRINT "------"
MAT CPY a()=b()
MAT PRINT a()
```

Gives:

```
     1,1,1
     1,1,1
     1,1,1
     1,1,1
     1,1,1
     -------
     2,3,4,5
     2,3,4,5
     2,3,4,5
     2,3,4,5
     -------
     2,3,4
     2,3,4
     2,3,4
     2,3,4
     1,1,1
```

MAT XCPY a([i,j])=b([k,l])[,h,w] works basically in the same manner as MAT CPY a([i,j])=b([k,l])[,h,w], except that matrix b is being transposed while being copied to matrix a, i.e. the rows and columns of matrix b are swapped while it is copied to matrix a. Array b remains unchanged, however. Only those elements are copied for which identical indices are given in both the source and the destination matrix.

Abbreviation: m x a(i,j)=b(k,l),h,w.

Example:

```
DIM a(5,3),b(4,4)
MAT SET a()=1
FOR i%=1 TO 4
     FOR j%=1 TO 4
          b(i%,j%)=SUCC(j%)
     NEXT j%
NEXT i%
MAT PRINT a()
PRINT "------"
MAT PRINT b()
PRINT "------"
MAT XCPY a(2,2)=b(2,2),3,3
MAT PRINT a()
```

Gives:

```
     1,1,1
     1,1,1
     1,1,1
     1,1,1
     1,1,1
     ------
     2,3,4,5
     2,3,4,5
     2,3,4,5
     2,3,4,5
     ------
     1,1,1
     1,3,3
     1,4,4
     1,5,5
     1,1,1
```

Further special cases

As with MAT CPY a(i,j)=b(k,l),w,h.

If MAT CPY or MAT XCPY are applied to vectors, j and l may be ignored. Following a DIM a(n),b(m), a() and b() are interpreted as row vectors, i.e. as matrix of the (1,n) or (1,m) types.

For a and b to be treated as column vectors, they must be dimensioned as matrix of the (n,1) or (m,1) type, ie. DIM a(n,1),b(n,1).

If both vectors are of the same order (both are row or column vectors), MAT CPY must be used. Irrespective of the type of vectors a and b, MAT CPY always treats both vectors syntactically as column vectors, so that the correct syntax to be used for MAT CPY is always

MAT CPY a(n,1)=b(m,1)!

Example:

```
DIM a(10),b(5)    ' a() und b() are row vectors
MAT SET a()=1
FOR i%=1 TO 5
     b(i%)=SUCC(i%)
NEXT i%
PRINT "a():    ";
MAT PRINT a()
PRINT "b():    ";
MAT PRINT b()
PRINT STRING$(45,"-")
```

MAT CPY a(3,1)=b(1,1) ! interprets a() and b() as column vectors

```
PRINT "MAT CPY a(3,1)=b(1,1):  ";
MAT PRINT a()
```

Gives:

```
     a(): 1,1,1,1,1,1,1,1,1,1
     b(): 2,3,4,5,6
     ---------------------------------------
     MAT CPY a(3,1)=b(1,1): 1,1,2,3,4,5,6,1,1,1
```

For **MAT XCPY**, one of the two vectors a and b must be explicitly dimensioned as a row vector, the other as a column vector: for example

```
DIM a(1,10),b(5,1),
```

Since MAT XCPY first transposes the second vector before copying it to the first. For this reason, MAT XCPY can only be used for DIM a(1,n),b(m,1): a()=row vector, b()=column vector and DIM a(n,1),b(1,m): a()=column vector, b()=row vector.

Example:

```
DIM a(1,10),b(5,1)
MAT SET a()=1
FOR i%=1 TO 5
     b(i%,1)=SUCC(i%)
NEXT i%
MAT PRINT a()
PRINT
MAT PRINT b()
MAT XCPY a(1,3)=b(1,1)
PRINT
MAT PRINT a()
```

Gives:

```
     1,1,1,1,1,1,1,1,1,1
     2
     3
     4
     5
     6
     1,1,2,3,4,5,6,1,1,1
```

Optionally, the parameters h and w can also be used when copying vectors with MAT CPY or MAT XCPY. However, the following applies: with MAT CPY, only the h parameter is used for w=1. No copying takes place with w=0.

With MAT XCPY, only h is used for w=1 if b is a column vector to be copied into a row vector after transposition. No copying takes place when w=0. On the other hand, only w is used for h=1 if b is a row vector which is to be copied to a column vector after transposition. In this case, no copying takes place if h=0.

MAT TRANS a()=b() copies the transposed from matrix b to matrix a if a and b are dimensioned accordingly, i.e. the number of rows from a must correspond to the number of columns in b, and the number of columns from a to the number of rows of n: for example, DIM a(n,m),b(m,n).

Example:

```
DIM a(3,4),b(4,3)
MAT SET b()=4
MAT SET a()=1
MAT PRINT a()
PRINT STRING$(10,"-")
MAT PRINT b()
PRINT STRING$(10,"-")
MAT TRANS a()=b()
MAT PRINT a()
```

Gives:

```
 1,1,1,1
 1,1,1,1
 1,1,1,1
----------
 4,4,4
 4,4,4
 4,4,4
 4,4,4
----------
 4,4,4,4
 4,4,4,4
 4,4,4,4
```

In the case of a square matrix, i.e. one with equal numbers of rows and columns, MAT TRANS a() may be used. This command swaps the rows and columns of matrix a and writes the matrix thus changed back to a.

(The original matrix is lost in the process but can be restored with another MAT TRANS a()).

Abbreviation: m t a().

Example:

```
DIM a(5,5)
FOR i%=1 TO 5
     FOR j%=1 TO 5
          a(i%,j%)=j%
     NEXT j%
NEXT i%
MAT PRINT a()
PRINT STRING$(10,"-")
MAT TRANS a()
MAT PRINT a()
```

Gives:

```
 1,2,3,4,5
 1,2,3,4,5
 1,2,3,4,5
 1,2,3,4,5
 1,2,3,4,5
----------
 1,1,1,1,1
 2,2,2,2,2
 3,3,3,3,3
 4,4,4,4,4
 5,5,5,5,5
```

Operation commands

MAT ADD a()=b()+c()
MAT ADD a(),b()
MAT ADD a(),x

MAT SUB a()=b()-c()
MAT SUB a(),b()
MAT SUB a(),x

MAT MUL a()=b()*c()
MAT MUL x=a()*b()
MAT MUL x=a()*b()*c()
MAT MUL a(),x

MAT NORM a(),0
MAT NORM a(),1

MAT DET x=a([i,j])[,n]
MAT QDET x=a([i,j])[,n]
MAT RANG x=a([i,j])[,n]
MAT INV a()=b()

a,b,c: Names of numerical floating point fields
x: aexp; scalar value
i,j,n: aexp

MAT ADD a()=b()+c() is only defined for matrix (vectors) of the same order, e.g. DIM a(n,m),b(m,m),c(n,m) or DIM a(n),b(n),c(n). Array c is added to matrix b, element by element, and the result is written to matrix a.

Abbreviation: m a()=b()+c().

Example:

```
DIM a(3,5),b(3,5),c(3,5)
MAT SET b()=3
MAT SET c()=4
```

```
MAT PRINT b()
PRINT STRING$(10,"-")
MAT PRINT c()
PRINT STRING$(10,"-")
MAT ADD a()=b()+c()
MAT PRINT a()
```

Gives:

```
 3,3,3,3,3
 3,3,3,3,3
 3,3,3,3,3
 ----------
 4,4,4,4,4
 4,4,4,4,4
 4,4,4,4,4
 ----------
 7,7,7,7,7
 7,7,7,7,7
 7,7,7,7,7
```

MAT ADD a(),b() is only defined for matrix (vectors) of the same order, e.g. DIM a(n,m),b(n.m) or DIM a(n),b(n). Array b is added to matrix a, element by element, and the result is written to matrix a.

The original matrix a is lost in the process.

Abbreviation: m a a(),b().

Example:

```
DIM a(3,5),b(3,5)
MAT SET a()=1
MAT SET b()=3
MAT PRINT a()
PRINT STRING$(10,"-")
MAT PRINT b()
PRINT STRING$(10,"-")
MAT ADD a(),b()
MAT PRINT a()
```

Gives:

```
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
---------
3,3,3,3,3
3,3,3,3,3
3,3,3,3,3
---------
4,4,4,4,4
4,4,4,4,4
4,4,4,4,4
```

MAT ADD a(),x is defined for all matrix (vectors). Here, the scalar x is added to matrix a, element by element, and the result is written to matrix a. The original matrix a is lost in the process.

Abbreviation: m a a(),x.

Example:

```
DIM a(3,5)
MAT SET a()=1
MAT PRINT a()
PRINT STRING$(10,"-")
MAT ADD a(),5
MAT PRINT a()
```

Gives:

```
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
---------
6,6,6,6,6
6,6,6,6,6
6,6,6,6,6
```

MAT SUB a()=b()+c() is only defined for matrix (vectors) of the same order, e.g. DIM a(n,m),b(n,m),c(n,m) or DIM a(n),b(n),c(n). Array c is subtracted from matrix b, element by element, and the result is written to matrix a.

Abbreviation: m a()=b()-c().

Example:

```
DIM a(3,5),b(3,5),c(3,5)
MAT SET b()=5
MAT SET c()=3
MAT PRINT b()
PRINT STRING$(10,"-")
MAT PRINT c()
PRINT STRING$(10,"-")
MAT SUB a()=b()-c()
MAT PRINT a()
```

Gives:

```
 5,5,5,5,5
 5,5,5,5,5
 5,5,5,5,5
----------
 3,3,3,3,3
 3,3,3,3,3
 3,3,3,3,3
----------
 2,2,2,2,2
 2,2,2,2,2
 2,2,2,2,2
```

MAT SUB a(),b() is only defined for matrix (vectors) of the same order, e.g. DIM a(n,m),b(n,m) or DIM a(n),b(n). Array b is subtracted from matrix a, element by element, and the result written to matrix a.

The original matrix a is lost in the process.

Abbreviation: m s a(),b().

Example:

```
DIM a(3,5),b(3,5)
MAT SET a()=3
MAT SET b()=1
MAT PRINT a()
PRINT STRING$(10,"-")
MAT PRINT b()
PRINT STRING$(10,"-")
MAT SUB a(),b()
MAT PRINT a()
```

Gives:

```
 3,3,3,3,3
 3,3,3,3,3
 3,3,3,3,3
----------
 1,1,1,1,1
 1,1,1,1,1
 1,1,1,1,1
----------
 2,2,2,2,2
 2,2,2,2,2
 2,2,2,2,2
```

MAT SUB a(),x is defined for all matrix (vectors). Here, the scalar x is subtracted from matrix x, element by element, and the result is written to matrix a. The original matrix a is lost in the process.

Abbreviation: m s a(),x

Example:

```
DIM a(3,5)
MAT SET a()=6
MAT PRINT a()
PRINT STRING$(10,"-")
MAT SUB a(),5
MAT PRINT a()
```

Gives:

```
6,6,6,6,6
6,6,6,6,6
6,6,6,6,6
-----------
5,5,5,5,5
5,5,5,5,5
5,5,5,5,5
-----------
1,1,1,1,1
1,1,1,1,1
1,1,1,1,1
```

MAT MUL a()=b()*c() is defined for matrix of an "appropriate" order. Arrays b and c are multiplied with each other. The result of this multiplication is written to matrix a. In order for the result to be defined, the matrix on the left (matrix b in this case) must have the same number of columns as the matrix on the right (c in this case) has rows. Array a, in this case, must have as many rows as b and as many columns as c, for example:

```
DIM a(2,2),b(2,3),c(3,2)
```

Arrays are multiplied as "row by column", i.e. element a(i.j) is obtained by multiplying the elements in the ith row of matrix b with the elements in the jth column of matrix c, element by element, and then adding up the individual products.

Abbreviation: m a()=b()*c()

Example:

```
DIM a(2,2),b(2,3),c(3,2)
MAT SET b()=1
DATA 1,2,-3,4,5,-1
MAT READ c()
MAT PRINT b(),5,1
PRINT STRING$(18,"-")
MAT PRINT c(),5,1
PRINT STRING$(18,"-")
MAT MUL a()=b()*c()
MAT PRINT a(),5,1
```

Gives:

```
1.0, 1.0, 1.0
1.0, 1.0, 1.0
-------------
1.0, 2.0
-3.0, 4.0
5.0, -1.0
-------------
3.0, 5.0
3.0, 5.0
```

With vectors instead of matrix, **MAT MUL a()=b()*c()** results in the dyadic (or external) product of two vectors.

Example:

```
DIM a(3,3),b(3),c(3)
DATA 1,2,-3,4,5,-1
MAT READ b()
MAT READ c()
MAT PRINT b(),5,1
PRINT STRING$(18,"-")
MAT PRINT c(),5,1
PRINT STRING$(18,"-")
MAT MUL a()=b()*c()
MAT PRINT a(),5,1
```

Gives:

```
1.0,  2.0, -3.0
----------------
4.0,  5.0, -1.0
----------------
4.0,  5.0, -1.0
8.0, 10.0,-2.0
-12.0,-15.0, 3.0
```

MAT MUL x=a()*b() is only defined for vectors with an equal number of elements. The result x is the scalar product (the so-called interior product) of vectors a and b. The scalar product of two vectors is defined as the sum of n products a(i)*b(i),i=1,...,n.

Abbreviation: m x=a()*b().

Example:

```
DIM b(3),c(3)
DATA 1,2,-3,4,5,-1
MAT READ b()
MAT READ c()
MAT PRINT b(),5,1
PRINT STRING$(18,"-")
MAT PRINT c(),5,1
PRINT STRING$(18,"-")
MAT MUL x=b()*c()
PRINT x
```

Gives:

```
 1.0,  2.0, -3.0
------------------
 4.0,  5.0, -1.0
------------------
 17.0
```

MAT MUL x=a()*b()*c() is defined for qualified Vectors a and c as well as qualified Matrix b().

Abbreviation: m x=a()*b()*c().

Example:

```
DIM a(2),b(2,3),c(3)
DATA 1,2,-3,4,5
MAT READ a()
MAT READ c()
MAT SET b()=1
MAT PRINT a(),5,1
PRINT STRING$(18,"-")
MAT PRINT b(),5,1
PRINT STRING$(18,"-")
MAT PRINT c(),5,1
PRINT STRING$(18,"-")
MAT MUL x=a()*b()*c()
PRINT x
```

Gives:

```
 1.0,  2.0
-------------------
 1.0,  1.0,  1.0
 1.0,  1.0,  1.0
 1.0,  1.0,  1.0
-------------------
-3.0,  4.0,  5.0
-------------------
 18.0
```

MAT NORM a(),0 or MAT NORM a(),1 are defined for matrix and vectors. MAT NORM a(),0 normalises a matrix (a vector) by rows, MAT NORM a(),1 by columns. This means that after a normalisation by rows (by columns) the sum of the squares of all elements in each row (column) is identical at 1.

Abbreviation: m no a(),0 bzw. m no a(),1.

Example:

```
DIM a(10,10),b(10,10),v(10)
DATA 1,2,3,4,5,6,7,8,9,-1
DATA 3.2,4,-5,2.4,5.1,6.2,7.2,8.1,6,-5
DATA -2,-5,-6,-1.2,-1.5,-6.7,4.5,8.1,3.4,10
DATA 5,-2.3,4,5.6,12.2,18.2,14.1,16,-21,-13
DATA 4.1,5.2,16.7,18.4,19.1,20.2,13.6,14.8,19.4,18.6
DATA 15.2,-1.8,13.6,-4.9,5.4,19.8,16.4,-20.9,21.4,13.8
DATA -3.6,6,-8.2,-9.1,4,-2.5,2,3.4,6.7,8.4
DATA 4.7,8.3,9.4,10.5,11,19,15.4,18.9,-20,12.6
DATA 5.3,-4.7,6.1,6.5,6.9,-9.2,-10.8,4.3,5.6,9.1
DATA 21.4,19.5,28.4,19.3,24.6,14.9,71.3,23.5,14.5,-12.3
'
CLS
MAT READ a()
MAT CPY b()=a()           ! Source matrix stored
PRINT "Source matrix"
PRINT
MAT PRINT a(),7,2
~INP(2)
```

```
CLS
MAT NORM a(),0
PRINT
PRINT "Row: "
PRINT
MAT PRINT a(),7,2
~INP(2)
'
'
PRINT
PRINT "Test : "
PRINT
FOR i%=1 TO 10
     MAT XCPY v()=a(i%,1)
     MAT MUL x=v()*v()
     PRINT x'
NEXT i%
PRINT
~INP(2)
'
'
CLS
MAT CPY a()=b()
MAT NORM a(),1
PRINT "Column : "
PRINT
MAT PRINT a(),7,2
~INP(2)
'
PRINT
PRINT "Test : "
PRINT
FOR i%=1 TO 10
     MAT CPY v()=a(1,i%)     !Copy column a() in the
vector v()
     MAT MUL x=v()*v()
     PRINT x'
NEXT i%
~INP(2)
```

Gives:

```
Source matrix
1.00,   2.00,   3.00,   4.00,   5.00,   6.00,   7.00,
     8.00,   9.00,  -1.00
3.20,   4.00,  -5.00,   2.40,   5.10,   6.20,   7.20,
     8.10,   6.00,  -5.00
-2.00,  -5.00,  -6.00,  -1.20,  -1.50,  -6.70,   4.50,
     8.10,   3.40,  10.00
5.00,  -2.30,   4.00,   5.60,  12.20,  18.20,  14.10,
     16.00, -21.00, -13.00
4.10,   5.20,  16.70,  18.40,  19.10,  20.20,  13.60,
     14.80,  19.40,  18.60
15.20,  -1.80,  13.60,  -4.90,   5.40,  19.80,  16.40,
     -20.90,  21.40,  13.80
-3.60,   6.00,  -8.20,  -9.10,   4.00,  -2.50,   2.00,
     3.40,   6.70,   8.40
4.70,   8.30,   9.40,  10.50,  11.00,  19.00,  15.40,
     18.90, -20.00,  12.60
5.30,  -4.70,   6.10,   6.50,   6.90,  -9.20, -10.80,
     4.30,   5.60,   9.10
21.40,  19.50,  28.40,  19.30,  24.60,  14.90,  71.30,
     23.50,  14.50, -12.30

0.06,   0.12,   0.18,   0.24,   0.30,   0.35,   0.41,
     0.47,   0.53,  -0.06
0.18,   0.23,  -0.29,   0.14,   0.29,   0.36,   0.42,
     0.47,   0.35,  -0.29
-0.11,  -0.28,  -0.34,  -0.07,  -0.09,  -0.38,   0.26,
     0.46,   0.19,   0.57
0.12,  -0.06,   0.10,   0.14,   0.30,   0.45,   0.35,
     0.40,  -0.52,  -0.32
0.08,   0.10,   0.33,   0.36,   0.38,   0.40,   0.27,
     0.29,   0.38,   0.37
0.32,  -0.04,   0.29,  -0.10,   0.11,   0.42,   0.35,
     -0.44,   0.45,   0.29
-0.19,   0.32,  -0.44,  -0.48,   0.21,  -0.13,   0.11,
     0.18,   0.36,   0.45
```

```
0.11,   0.19,   0.21,   0.24,   0.25,   0.43,   0.35,
     0.43,  -0.46,   0.29
0.23,  -0.21,   0.27,   0.29,   0.31,  -0.41,  -0.48,
     0.19,   0.25,   0.40
0.23,   0.21,   0.30,   0.21,   0.26,   0.16,   0.76,
     0.25,   0.15,  -0.13

Test :

1 1 1 1 1 1 1 1 1 1

0.04,   0.08,   0.08,   0.12,   0.13,   0.14,   0.09,
     0.18,   0.20,  -0.03
0.11,   0.16,  -0.13,   0.07,   0.14,   0.14,   0.09,
     0.18,   0.13,  -0.14
-0.07,  -0.21,  -0.15,  -0.04,  -0.04,  -0.15,   0.06,
     0.18,   0.07,   0.28
0.18,  -0.09,   0.10,   0.17,   0.33,   0.41,   0.18,
     0.35,  -0.46,  -0.36
0.14,   0.21,   0.42,   0.57,   0.51,   0.46,   0.17,
     0.33,   0.42,   0.52
0.53,  -0.07,   0.35,  -0.15,   0.15,   0.45,   0.21,
     -0.46,   0.47,   0.38
-0.13,   0.25,  -0.21,  -0.28,   0.11,  -0.06,   0.03,
     0.08,   0.15,   0.23
0.17,   0.34,   0.24,   0.33,   0.30,   0.43,   0.20,
     0.42,  -0.44,   0.35
0.19,  -0.19,   0.15,   0.20,   0.19,  -0.21,  -0.14,
     0.10,   0.12,   0.25
0.75,   0.80,   0.72,   0.60,   0.66,   0.34,   0.90,
     0.52,   0.32,  -0.34

Test :

1 1 1 1 1 1 1 1 1 1
```

MAT DET x=a([i,j])[,n] calculates the determinants of a square matrix of the (n,n) type. The row and column offsets are preset to a(0,0) or a(1,1), depending on MAT BASE 0 or MAT BASE 1, assuming that OPTION BASE 1 is enabled. It is also possible, however, to calculate the determinant of a square part matrix. To do this, the row and column offsets of a() must be specified as i and j, and the number of elements in the part matrix as n. A part matrix of the (n,n) type is then created internally starting from the "position" ith row, jth column.

Abbreviation: m d x=a([i,j])[,n].

Example:

```
DIM a(10,10),b(4,4)
DATA 1,2,3,4,5,6,7,8,9,-1
DATA 3.2,4,-5,2.4,5.1,6.2,7.2,8.1,6,-5
DATA -2,-5,-6,-1.2,-1.5,-6.7,4.5,8.1,3.4,10
DATA 5,-2.3,4,5.6,12.2,18.2,14.1,16,-21,-13,3.8
DATA 4.1,5.2,16.7,18.4,19.1,20.2,13.6,14.8,19.4,18.6
DATA 15.2,-1.8,13.6,-4.9,5.4,19.8,16.4,-20.9,21.4,13.8
DATA -3.6,6,-8.2,-9.1,4,-2.5,2,3.4,6.7,8.4,10.9
DATA 4.7,8.3,9.4,10.5,11,19,15.4,18.9,-20,12.6
DATA 5.3,-4.7,6.1,6.5,6.9,-9.2,-10.8,4.3,5.6,9.1
DATA 21.4,19.5,28.4,19.3,24.6,14.9,71.3,23.5,14.5,-12.3
'
CLS
MAT READ a()
PRINT "Source matrix"
PRINT
MAT PRINT a(),7,2
PRINT
PRINT "Determinant : ";
MAT DET x=a()
PRINT x;
MAT DET y=a(1,4),4
PRINT
PRINT "Determinant from a(1,4),4 : ";
PRINT y
PRINT
PRINT "Test :"
```

```
PRINT
MAT CPY b()=a(1,4),4,4 ! copy to b()
MAT PRINT b(),7,2
MAT DET z=b()          ! Determinant from b()
PRINT
PRINT z
```

Gives:

```
Source Matrix
1.00,   2.00,   3.00,   4.00,   5.00,   6.00,   7.00,
          8.00,   9.00,  -1.00
3.20,   4.00,  -5.00,   2.40,   5.10,   6.20,   7.20,
          8.10,   6.00,  -5.00
-2.00,  -5.00,  -6.00,  -1.20,  -1.50,  -6.70,   4.50,
          8.10,   3.40,  10.00
5.00,  -2.30,   4.00,   5.60,  12.20,  18.20,  14.10,
          16.00, -21.00, -13.00
3.80,   4.10,   5.20,  16.70,  18.40,  19.10,  20.20,
          13.60,  14.80,  19.40
18.60,  15.20,  -1.80,  13.60,  -4.90,   5.40,  19.80,
          16.40, -20.90,  21.40
13.80,  -3.60,   6.00,  -8.20,  -9.10,   4.00,  -2.50,
          2.00,   3.40,   6.70
8.40,  10.90,   4.70,   8.30,   9.40,  10.50,  11.00,
          19.00,  15.40,  18.90
-20.00,  12.60,   5.30,  -4.70,   6.10,   6.50,   6.90,
          -9.20, -10.80,   4.30
5.60,   9.10,  21.40,  19.50,  28.40,  19.30,  24.60,
          14.90,  71.30,  23.50

Determinant : -2549840202186
Determinant from a(1,4),4 : -57.61200000001

Test:
4.00,   5.00,   6.00,   7.00
2.40,   5.10,   6.20,   7.20
-1.20,  -1.50,  -6.70,   4.50
5.60,  12.20,  18.20,  14.10

-57.61200000001
```

MAT QDET x=a([i,j])[,n] works in the same manner as MAT DET x>a([i,j])[,n], except that it has been optimised for speed rather than accuracy. Both will normally produce identical results. With "critical" matrix, whose determinant is close to 0, you should always use MAT DET, though.

Abbreviation: M qd x=a([i,j])[,n].

Example:

```
DIM a(10,10)
DATA 1,2,3,4,5,6,7,8,9,-1
DATA 3.2,4,-5,2.4,5.1,6.2,7.2,8.1,6,-5
DATA -2,-5,-6,-1.2,-1.5,-6.7,4.5,8.1,3.4,10
DATA 5,-2.3,4,5.6,12.2,18.2,14.1,16,-21,-13,3.8
DATA 4.1,5.2,16.7,18.4,19.1,20.2,13.6,14.8,19.4,18.6
DATA 15.2,-1.8,13.6,-4.9,5.4,19.8,16.4,-20.9,21.4,13.8
DATA -3.6,6,-8.2,-9.1,4,-2.5,2,3.4,6.7,8.4,10.9
DATA 4.7,8.3,9.4,10.5,11,19,15.4,18.9,-20,12.6
DATA 5.3,-4.7,6.1,6.5,6.9,-9.2,-10.8,4.3,5.6,9.1
DATA 21.4,19.5,28.4,19.3,24.6,14.9,71.3,23.5,14.5,-12.3
'
CLS
MAT READ a()
PRINT "Source matrix"
PRINT
MAT PRINT a(),7,2
PRINT
PRINT "Determinant with MAT DET : ";
MAT DET x=a()
PRINT x;
PRINT
PRINT "Determinant with MAT QDET : ";
MAT DET y=a()
PRINT y;
PRINT
PRINT "Deviation : ";x-y
```

Gives:

```
Source Matrix

1.00,   2.00,   3.00,   4.00,   5.00,   6.00,   7.00,
     8.00,   9.00,  -1.00
3.20,   4.00,  -5.00,   2.40,   5.10,   6.20,   7.20,
     8.10,   6.00,  -5.00
-2.00,  -5.00,  -6.00,  -1.20,  -1.50,  -6.70,   4.50,
     8.10,   3.40,  10.00
5.00,  -2.30,   4.00,   5.60,  12.20,  18.20,  14.10,
     16.00, -21.00, -13.00
3.80,   4.10,   5.20,  16.70,  18.40,  19.10,  20.20,
     13.60,  14.80,  19.40
18.60,  15.20,  -1.80,  13.60,  -4.90,   5.40,  19.80,
     16.40, -20.90,  21.40
13.80,  -3.60,   6.00,  -8.20,  -9.10,   4.00,  -2.50,
     2.00,   3.40,   6.70
8.40,  10.90,   4.70,   8.30,   9.40,  10.50,  11.00,
     19.00,  15.40,  18.90
-20.00,  12.60,   5.30,  -4.70,   6.10,   6.50,   6.90,
     -9.20, -10.80,   4.30
5.60,   9.10,  21.40,  19.50,  28.40,  19.30,  24.60,
     14.90,  71.30,  23.50

Determinant with MAT DET : -2549840202186
Determinant with MAT QDET : -2549840202186
Deviation : 0
```

MAT RANG x=a([i,j])[,n] outputs the rank of a square matrix. As with MAT DET or MAT QDET, you can select any row and column offset. The number of elements in the part matrix must be specified with n. This creates a part matrix of the (n,n) type internally, starting from the position ith row, jth column.

Abbreviation: m ra x=a([i,j])[,n].

Example:

```
DATA 1,2,3,4,5
DATA 3.2,4,-5,2.4,5.1
DATA -2,4,-5,2.4,5.1
DATA 5,-2.3,4,5.6,12.2
DATA 4.1,5.2,16.7,18.4,19.1
'
CLS
MAT READ a()
PRINT "Source matrix"
PRINT
MAT PRINT a(),7,2
PRINT
PRINT "Rang from a(): ";
MAT RANG x=a()
PRINT x;
PRINT
PRINT "Rang from a(1,2),3 : ";
MAT RANG y=a(1,2),3
PRINT y;
PRINT
```

Gives:

```
Source matrix
1.00,    2.00,    3.00,    4.00,    5.00
3.20,    4.00,   -5.00,    2.40,    5.10
-2.00,   4.00,   -5.00,    2.40,    5.10
5.00,   -2.30,    4.00,    5.60,   12.20
4.10,    5.20,   16.70,   18.40,   19.10

Rang from a(): 5
Rang from a(1,2),3 : 2
```

MAT INV b()=a() is used to determine the inverses of a square matrix. The inverse of matrix a() is written to matrix b(), hence b() must be of the same type as a().

Abbreviation: m inv b()=a().

Example:

```
DIM a(5,5),b(5,5),c(5,5)
DATA 1,2,3,4,5
DATA 3.2,4,-5,2.4,5.1
DATA -2,4,-5,2.4,5.1
DATA 5,-2.3,4,5.6,12.2
DATA 4.1,5.2,16.7,18.4,19.1
'
CLS
MAT READ a()
PRINT "Source matrix a() : "
PRINT
MAT PRINT a(),7,2
'
MAT INV b()=a()
PRINT
PRINT "Inverse from a() : "
PRINT
MAT PRINT b(),7,2
PRINT
PRINT "Test b()*a()>Unity matrix ? "
PRINT
MAT MUL c()=b()*a()
MAT PRINT c(),7,2
```

Gives:

```
Source matrix a() :
1.00,    2.00,    3.00,    4.00,    5.00
3.20,    4.00,   -5.00,    2.40,    5.10
-2.00,   4.00,   -5.00,    2.40,    5.10
5.00,   -2.30,    4.00,    5.60,   12.20
4.10,    5.20,   16.70,   18.40,   19.10
```

```
Inverse from a() :
0.00,    0.19,  -0.19,  -0.00,  -0.00
0.97,    0.02,  -0.09,  -0.10,  -0.17
0.71,   -0.10,  -0.10,  -0.01,  -0.12
-1.65,   0.17,   0.11,  -0.06,   0.39
0.71,   -0.12,   0.04,   0.09,  -0.17

Test b()*a()>Unity matrix ?
1.00,    0.00,   0.00,   0.00,   0.00
0.00,    1.00,   0.00,   0.00,  -0.00
0.00,   -0.00,   1.00,   0.00,  -0.00
-0.00,  -0.00,  -0.00,   1.00,   0.00
-0.00,   0.00,   0.00,   0.00,   1.00
```

Further new commands in Version 3.5

In addition to the commands described in Chapter 6, Version 3.5 of GFA BASIC also implements three commands from the field of combinatorics, two commands for the operation of DATA pointers,

Commands from the field of combinatorics

These commands are:

x=FACT(n)

y=VARIAT(n,k)

z=COMBIN(n,k)

x,y,x: aexp
n,k: iexp

x=FACT(n) calculates the factorial (n!) of n and writes this value to the variable x. The factorial of an integer number is defined as the product of a multiplication with the first n integer numbers, with 0!=1.

y=VARIAT(n,k) calculates the number of variations of n elements to the kth class without repetition, and writes this value to the variable y.

The number of variations of n elements to the kth class without repetitions is defined as

VARIAT(n,k)=n!/(n-k)!

z=COMBIN(n,k) calculates the number of combinations of n elements to the kth class without repetitions and writes this value to the variable z. The number of combinations of n elements to the kth class without repetitions is defined as

COMBIN(n,k) =n!/((n-k)!*k!)

Example:

```
x=FAXT(6)
y=VARIAT(6,2)
z=COMBIN(6,2)
PRINT x,y,z
~INP(2)
```

Gives:

```
720    30    15
```

Command for the operation of DATA pointers

_DATA
_DATA=

_DATA specifies the position of the DATA pointer. _DATA is 0 if the next READ would result in an "out of data".

_DATA= permits the setting of the DATA pointer to a value which has been previously determined with _DATA.

Example:

```
DIM dp%(100)
DATA 1,2,3,4,5,6,7,8,9
DATA 13,24,328,3242,1,0
'
i%=0
DO WHILE _DATA
      dp%(i%)=_DATA
      INC i%
      READ a
LOOP
'
DEC i%
'
FOR j%=i% DOWNTO 0
      _DATA=dp%(j%)
      READ a
      PRINT a
NEXT j%
~INP(2)
```

Gives:

```
0 1 3242 328 24 13 9 8 7 6 5 4 3 2 1
```

STE Support

Version 3.5E has some new commands specific for use on the Atari STE computer:

STICK(i) The values for 'i' on the STE can now be in the range 0 to 5. 0 and 1 act the same as normally found on the ST but 2 - 5 are exclusively for the STE, and DO NOT check for the mouse at the same time. This effectively speeds up the joystick polling.

STRIG(i) This is the eqivalent Joystick Button command.

The following commands are available for reading the STE's paddle controllers.

PADX(i) This command gives the X position of one of the 2 paddles (i can be 0 or 1).
PADY(i) The equivalent Y-Position.
PADT(i) This command reads the paddle buttons.

The following commands are available for reading the STE's lightpen socket.

LPENX The X position of a connected lightpen.
LPENY The Y position of a connected lightpen.

You now have the ability to detect which computer your GFA programs are running on with the commands:

STE? Returns -1 for STE(or TT), otherwise 0.
TT? Returns -1 for 68020 or 68030 Processor, otherwise 0.

DMACONTROL ctrlvar

ctrlvar	>0	Stop sound.
	1	Play sound once.
	3	Play sound in loop.

The following STE command allows sampled DMA sound to be played:

DMASOUND beg,end,rate[,ctrl]
beg>Sample start address.
end>Sample end address.
rate>Sample rate (0=6.25 kHz, 1=12.5 kHz, 2=25 kHz, 3=50 kHz).
ctrl>See command DMACONTROL, (above)

MWOUT mask,data
This command controls the STE-Interne Micro-Wire-Interface, and is currently used for controlling sound.

MWOUT &H7FF,x
x=&X10 011 ddd ddd Set Master Volume

000 000 -80 dB
010 100 -40 dB
101 xxx 0 dB

The value of the last 5 Bits is eqivalent to HALF of the volume in dB.

x=&x10 101 xdd ddd Set Front Left Channel
x=&x10 100 xdd ddd Set Front Right Channel
x=&x10 111 xdd ddd Set Rear Left Channel (Reserved)
x=&x10 110 xdd ddd Set Rear Right Channel (Reserved)

00 000 -40 dB
01 010 -20 dB
10 1xx 0 dB

The last 4 Bits*2>dB

x=&X10 010 xxd ddd Set Treble
x=&X10 001 xxd ddd Set Bass
0 000 -12dB
0 110 0 dB (flat)
1 100 +12 dB

x=&X10 000 xx0 ddd Set Mix (input select)
000 Off (-100 Db with volume at -80dB)
010 Mix GI sound (normal ST)
011 Reserved
100 Reserved

Example:

```
MWOUT &H7FF,&X10000000010 Switches the ST's sound off.
```

In the following example, try each of the DMASOUND lines for different effects.

```
PRINT STE?                    ! Prints -1 if an STE
n%=360*32
DIM a|(n%)
```

```
' DMASOUND V:a|(0),V:a|(n%),0
' DMASOUND V:a|(0),V:a|(n%),1
' DMASOUND V:a|(0),V:a|(n%),2
DMASOUND V:a|(0),V:a|(n%),3,3
FOR i%=0 TO n%
     a|(i%)=128+SINQ(i%*i%/7200)*127
NEXT i%
REPEAT
UNTIL MOUSEK
DMACONTROL 0
```

Editor

For the STE, MERGE'ing Basic files will stop when a right arrow, (Control-D), is found at the start of the line.

SETCOLOR

The STE's enhanced colour is now also supported, and the available 4096 colours can be accessed by use of SETCOLOR as normal, but the values can now be in the range of 0 to 15, e.g.:

```
SETCOLOR 2,13,13,13
```

INDEX

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X