X11-BASIC

VERSION 1.14

(kollo@users.sourceforge.net)

(see http://x11-basic.sourceforge.net/)

X11-Basic is a comprehensive non-standard Basic interpreter with full X capability that integrates fratures like shell scripting, cgi programming and full graphical visualisation into the easy to learn basic language on moden computers. The syntax is most similar to the old GFA-Basic on ATARI-ST implementation. Old GFA-programs should run with only few changes.

About this document

This document describes the features of X11-Basic. You will find information about the X11-Basic interpreter (the program `xbasic` under Unix or `xbasic.exe` under Windows) as well as the language itself. For a more compact description you may want to read the `x11basic(1)` man-page or the man-page of the X11-Basic pseudo compiler `xbc(1)`.

The latest information and updates and new versions of X11-Basic can be found at `http://x11-basic.sourceforge.net/`.

Contents

1 ABOUT X11-Basic

X11-Basic is a comprehensive non-standard Basic interpreter with full X capability that integrates features like shell scripting, cgi programming and full graphical visualisation into the easy to learn basic language on modern computers.

The actual implementation runs on Unix workstations (like HP/UX, DEC alpha and maybe others) and Linux-PCs with the X-Window system. A MS-Windows version has been started. Also Hand-held PC Versions or adaptations to VxWorks are actually been thought of.

X11-Basic is as well suited to novices as programming wizards, and is appropriate for virtually all programming tasks. For science and engineering X11-Basic has already prooved its capability of handling complex simulation and control problems. For system programs, X11-Basic has high level language replacements for low level programming features that are much easier to read, understand, and maintain. For all applications, X11-Basic is designed to support rapid development of compact, efficient, reliable, readable, portable, well structured programs.

The X11-Basic environment contains a library of GEM GUI functions. This makes writing GUI programs in X11-Basic faster and easier than programming with native GUI tools.

Structured programming

X11-Basic is a structured programming language. Structure is a form of visual and functional encapsulation in which multiple-line sections of program look and act like single units. The beginning and end of blocks are marked by descriptive keyword delimiters. Lines within blocks are generally indented to make the block visible.

Even if the interpreter is BASIC, there are restraints and extansions to it for a structured programming with a better overall view. There are no line numbers and every line holds only one instruction. Jumps with GOTO are possible but not necessary. All the well-known loops are available including an additional command for discontinuation (–> EXIT IF). Procedures and functions with return values of any type can be defined. So a program can contain a main part to call the subfunctions, for example in side of a loop. Whole libraries can be added with the merge command (–> MERGE).

Copyright information

X11-Basic is free software and distributed under the GNU License. Read the file COPYING for details.

2 Usage

The X11-Basic interpreter is called xbasic (xbasic.exe under Windows). Under Unix it is usually installed in the /usr/bin/ (if installed via the .rpm) or in /usr/local/bin (if installed manually from the source package) path.

2.1 Installing X11-Basic

SuSE-Linux and RedHat

If you got a Redhat-Package (RPM) e.g. the File X11Basic-1.14-4.i386.rpm, then you can install this package (beeing root) with rpm -i X11Basic-1.14-4.i386.rpm.

This is a very convienient way at least for the Linux distributions Feodora, SuSE and RedHat (and many others) of installing the interpreter and its documentation, the X11-Basic pseudo compiler xbc, The bytecode compiler xbbc, the bytecode interpreter (virtual machine) xbvm and the ANSI-Basic to X11-Basic converter bas2x11basic, the man-pages and a small collection of example programs.

After having done so, do a rehash, and then you can execute the interpreter with xbasic or read the man pages with man xbasic or man x11basic.

The documentation will be installed in the /usr/share/doc/packages/X11Basic/ directory (at least on my system) and you can find the following files:

```
-rw-r--r--    1005  ACKNOWLEGEMENTS      -- acknowlegements
-rw-r--r--      46  AUTHORS              -- contact addresses of the author
-rw-r--r--   17982  COPYING              -- copyright information
-rw-r--r--    2960  INSTALL              -- installation instructions
-rw-r--r--    1752  README               -- short description
-rw-r--r--     169  RELEASE_NOTES        -- release notes
-rw-r--r--  164366  X11-Basic-manual.txt -- the manual
drwxr-xr-x    1024  editors/             -- files for editors
drwxr-xr-x    1024  examples/            -- few example programs
```

Debian based distributions

As with Ubuntu linux the popularity of debian based linux distributions has dramatically increased in the recent past. RPM is not used for debian based distributions. But with the RPM to DEB converter alien, a .deb package can be created out of the .rpm files. If you hesitate to do so, there are also ready-made debian packages available on various sources. You just need to look for them on the internet. Al though they are maintained by third party volunteers, the file system structure should be similar to what is described in the previous chapter, so you should expect to find the same files at the same place.

Other linux and UNIX distributions

Maybe RPM is supported for your distribution, then you can follow the instructions above. If not, you might look for a debian package. Also the existence of a package for HPUX is known (unfortunately it contains the very old version 1.04 and seems not to be maintained any more). So far (as I know) there exist no packages for any other system.

For all other systems you will have to get the source-package `X11Basic-1.14.tar.gz` and compile the sources (which is very easy, by the way). This should work for all linux distributions, and probably with little modifications also for HPUX, for DEC/alpha and FreeBSD and maybe others. Also X11-Basic compiles on Cygwin, so you can use X11-basic even on WINDOWS operating systems.

In order to compile X11Basic, you will need the following:

- A C compiler, preferably GNU C (but some ANSI C compilers, like DEC's will do)
- a 'csh'-compatible shell
- X11R5 or R6 libraries (for the graphics)
- the `readline` library
- the `termcap` library

These will suffice to get you started.

Unpack `X11Basic-1.14.tar.gz` with

```
tar xzf X11Basic-1.14.tar.gz
```

go into the `X11Basic-1.14` directory and do a

```
sh ./install.sh
```

Thats all you will have to do (except for something goes wrong).

A `xbasic` binary will be compiled into the `./src/` directory

```
cd src/
make install
```

will copy the binary in `${HOME}/bin/`.

If the 'install.sh' script fails, please contact me (kollousers.sourceforge.net) and send me the output it generated (install.log). I am going to try to help you to fix the problem.

make the pseudo-compiler work, libraries

If you want to use the pseudo compiler, the bytecode compiler and the virtual machine included in this package, you have to make both the static and dynamic libraries.

```
cd ./src
```

If you do not have the file `mathematics.c` please do

```
cp mathemat_dummy.c mathematics.c
```

then:

Libraries

`make lib` will generate `libx11basic.so`

`make x11basic.a` will generate `x11basic.a` – now you have to be root –

`make installroot` will generate a dynamic linked binary of the interpreter, copy it to `/bin/`, install the libraries in `/usr/local/lib/`, install the man-page

The compiler

`make xbc` will make the X11-Basic compiler You can then (beeing root) do a

`make xbcinstall` to install the compiler in `/usr/local/bin/` and the compiler man-page

After doing all this, you will also find a `xbasic.dyn` executabe. This is the interpreter, dynamically linked and much shorter.

The bytecode compiler and the virtual machine Both programs `xbbc` and `xbbc` are new to this package. The idea is to increase the excecution speed of X11-basic programs a lot by compiling it to a bytecode before execution. The bytecode then can be run much faster with the bytecode interpreter, the so-called virtual machine. The conversion to bytecode is a real compilation (in contrast to what the pseudo compiler `xbc` does). The step to assembler or machine code is not far. Also a conversion to C or to JAVA or any other language will be straigt forward, although the code will practically not be readable to humans anymore. As with JAVA, the bytecode is still platform independant and can be run on any system, which has a virtual machine ported to.

One point to mention (whether this is a feature or a disadvantage): X11-basic bytecode can not be converted back to source code (.bas).

Because this is a new feature which is incomplete and still has a lot of bugs the feature is not incorporated into the interpreter `xbasic` itself.

Please try the bytecode compiler out and maybe you want to report errors etc.

Support

If you have trouble with X11-Basic, you may send me a mail or browse the faq.

If you have problems compiling X11-Basic under Unix, you should add the file `install.log`, which is created by the `install.sh`-script and contains all the information needed to pin down the error.

If you have trouble with some X11-Basic-command or program, you should create a minimum sample program to reproduce the error; please keep this sample program as small as possible. Then take the program and send it to me. Add a short description of you problem, containing:

- Which Operating system are you using: Windows (95, 98, me, xp, NT, 2000) or Unix (linux, solaris, FreeBSD, HPUX ...) ?
- How does the program behave on your computer ? What did you expect ?
- Which version of X11-Basic are you using ? Please try the latest one ! You also might want to fill in the bug report form.

2.2 The X11-Basic Interpreter

There are several ways to start the X11-Basic interpreter.

The simplest way is to just start it by the command `xbasic`. Then you can use the interpreter in interactive mode. Just try to enter some X11-Basic commands.

2.2.1 Command line parameters

The X11-Basic Interpreter takes some command line parameters.

Command line parameters are as follows:

xbasic <filename>	run Basic-Programm `[input.bas]`
`-l`	load only, don't execute
`-e <command>`	excecute basic command
`-eval <expression>`	evaluate numerical expression
`-daemon`	swich off prompting and echoing

The program does not insist on its name, the person installing can name it `xbasic`, `x11basic` or any other name. However `xbasic` is the preferred name for the executable. When you start the program without arguments it tells you the different options.

Examples:

```
xbasic testme.bas
xbasic -l dontrunme.bas
xbasic -e 'alert 1,"Hallo !",1," OK ",b'
xbasic --eval 1+3-4*3
```

2.2.2 X11-Basic as daemon

The commandline option `-daemon` forces the interpreter to run in daemon-mode (with no terminal connected). No prompt is given and the input is not echoed back.

Example:

To run the X11-Basic interpreter on a tcp-socket on port 1371 create a new user called `xbasic` and insert

```
--- in /etc/inetd.conf: ---
xbasic stream tcp nowait xbasic /usr/sbin/tcpd /bin/xbasic --daemon
--- in /etc/services: ---
xbasic          1371/tcp
---
```

Please note that this is not recommended since xbasic would open several security holes on your system.

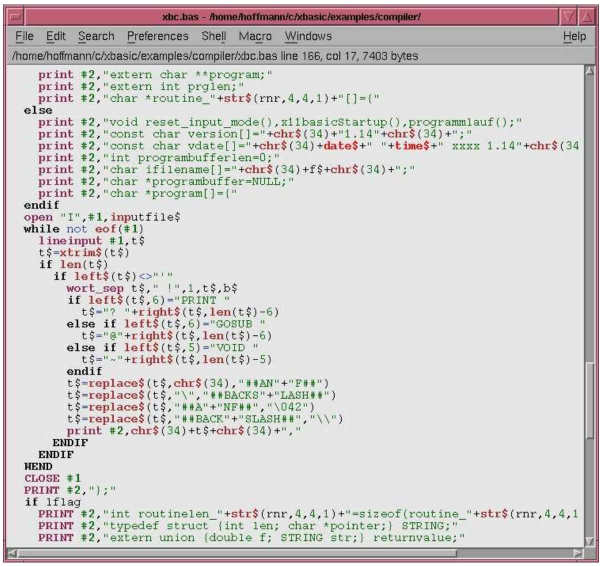
2.2.3 X11-Basic as shell

X11-Basic programs can be excecuted like shell scripts. Make sure that the very first line of your X11-Basic program starts with the characters `'#!'` followed by the full pathname of the X11-Basic inperpreter xbasic (e.g. `'#!/usr/bin/xbasic'`). This she-bang-line ensures, that your Unix will invoke xbasic to execute your program. Moreover, you will need to change the permissions of your X11-Basic program, e.g. `chmod 755 myprog`. After that your program can simply be executed from your shell and the interpreter works in the background like shells do.

2.3 Using Syntax highlighting with nedit

NEdit, the full featured, plain text Nirvana editor[1] is a GUI style text editor for workstations with the X Window System. NEdit provides all of the standard menu, dialog, editing, mouse support, macro extension language, syntax highlighting, and a lot other nice features (and

[1] `http://nedit.org/`



xbc.bas - /home/hoffmann/c/xbasic/examples/compiler/
File Edit Search Preferences Shell Macro Windows Help

/home/hoffmann/c/xbasic/examples/compiler/xbc.bas line 166, col 17, 7403 bytes

```
    print #2,"extern char **program;"
    print #2,"extern int prglen;"
    print #2,"char *routine_"+str$(rnr,4,4,1)+"[]={"
  else
    print #2,"void reset_input_mode(),x11basicStartup(),programmlauf();"
    print #2,"const char version[]="+chr$(34)+"1.14"+chr$(34)+";"
    print #2,"const char vdate[]="+chr$(34)+date$+" "+time$+" xxxx 1.14"+chr$(34
    print #2,"int programbufferlen=0;"
    print #2,"char ifilename[]="+chr$(34)+f$+chr$(34)+";"
    print #2,"char *programbuffer=NULL;"
    print #2,"char *program[]={"
  endif
  open "I",#1,inputfile$
  while not eof(#1)
    lineinput #1,t$
    t$=xtrim$(t$)
    if len(t$)
      if left$(t$)<>"'"
        wort_sep t$," !",1,t$,b$
        if left$(t$,6)="PRINT "
          t$="? "+right$(t$,len(t$)-6)
        else if left$(t$,6)="GOSUB "
          t$="@"+right$(t$,len(t$)-6)
        else if left$(t$,5)="VOID "
          t$="~"+right$(t$,len(t$)-5)
        endif
        t$=replace$(t$,chr$(34),"##AN"+"F##")
        t$=replace$(t$,"\","##BACKS"+"LASH##")
        t$=replace$(t$,"##A"+"NF##","\042")
        t$=replace$(t$,"##BACK"+"SLASH##","\\")
        print #2,chr$(34)+t$+chr$(34)+","
      ENDIF
    ENDIF
  WEND
  CLOSE #1
  PRINT #2,"};"
  if lflag
    PRINT #2,"int routinelen_"+str$(rnr,4,4,1)+"=sizeof(routine_"+str$(rnr,4,4,1
    PRINT #2,"typedef struct {int len; char *pointer;} STRING;"
    PRINT #2,"extern union {double f; STRING str;} returnvalue;"
```

Figure 2.1: The Nirvana Editor with syntax highlighting for a X11-Basic program.

extensions for programmers). In short, it has everything you want to develop your X11-Basic programs.

If you like to use nedit as your favorite editor, a `nedit.defs` file comes with this package. This enables syntax highlighting for X11-Basic programs in **nedit** (see fig. 2.1).

2.4 The pseudo compiler

The X11-Basic package is shipped with a pseudo compiler, which makes stand-alone binaries out of Basic source code. Actually this "compiler" is not a real compiler, since the source code is still interpreted on runtime. But the source code is linked to the X11-Basic library so there results one independant executable. Another advantage is that it is quite hard to extract a full running `*.bas` File from this binary since the code is compressd in a way.

You can find the compiler in `examples/compiler/xbc.bas`. Yes, the compiler compiles itself. Yust make sure you have built the shared library `libx11basic.so` and the library for static linking before (`make lib; make x11basic.a`) and moved it to `/usr/lib`. Then do

```
xbasic xbc.bas
```

Please note: If you link your programs dynamically, you will eventually have to compile them again when the `libx11basic.so` will have changed due to a version update of your X11-Basic package.

See the man page `xbc(1)` for further information on the compiler.

2.5 The WINDOWS Version of X11-Basic

After you have run the setup program, X11-Basic can be invoked in three ways:

1. Choose "X11-Basic" from the start-menu: `xbasic.exe` will come up with a console window and wait for commands to be typed in right away.
2. Click with the right mousebutton on your desktop. Choose "new" from the context-menu that appears; this will create a new icon on your desktop. The context-menu of this icon has three entries "Execute", "Edit" and "View docu" (which shows the embedded documentation, if any); a double-click executes the program.
3. Create a file containing your X11-Basic-program. This file should have the extension ".bas". Double-click on this file then invokes X11-Basic, to execute your program.

2.6 The ANSI-Basic to X11-Basic converter

You now will find a simple ANSI-Basic to X11-Basic converter (`bas2x11basic.bas`) in the `examples/compiler/` directory. It helps convertig old (real) Basic Programs with line numbers and multiple commands per line to the X11-Basic structure. Because there are so many different BASIC Version aroud, in most cases you will have to edit these files produced manually. But most of the work will already have been done by this converter.

Example:

```
xbasic bas2x11basic.bas ansibasic.bas -o newname.bas
```

For further options try `xbasic bas2x11basic.bas --help` and read the man-page `bas2x11basic(1).`

A compiled version of the ANSI-Basic to X11-Basic converter is included in the RPM-packages.

3 Getting Started

This chapter describes all you need to know to write your own programs in X11-Basic.

3.1 Introduction to this Dialect of BASIC

The programming language BASIC has been around since the 1960s. BASIC is an acronym and it stands for *Beginners All Purpose Symbolic Instruction Code*. BASIC was originally designd to be a programming language that is easy to use for a wide range of projects by anyone. X11-Basic is a dialect of this but it is not a BASIC in its original form. It is more a mix of classic BASIC with structured languages like PASCAL and Modula-2. The Syntax of X11-Basic is oriented to the famous GFA-BASIC which was developped for the ATARI ST in 1985.

X11-Basic has a lot of features which make the language different from the original (ANSI-Basic) intention. As with GFA-Basic these modifications help developing programs with having a more structured look and which make use of the more modern graphical user interfaces available on compunters since the mid 1980's:

- one command or declaration per line for better readability
- variables and identifiers in general can have up to 250 characters to distinguish them from each other
- data typing and arrays
- use of subroutines and functions
- powerful loop and program flow constructs
- file operations
- commands to directly access the OS
- commands for using X11-Windows graphics and a port of the AES from the ATARI ST, allowing for easy use of graphics in your program
- commands for direct memory manipulation, allowing you to access the machine almost as with machine language

- possibility to merge source code for libraries and reuse
- a compiler is also available

3.2 Interpreter vs Compiler

X11-Basic programs (or scripts) are interpreted by default. This means the so-called interpreter takes each line of your code and looks what to do with it. The compiler does it differently, it will take your code once, translate it into directly executable machine code resulting in a more speedy program execution as the step for command lookup does not appear anymore. The compiled program just can be executed out of the box. The advantage of an interpreter is that you can directly test and run your program without running a compiler first. This is helpful while developing but of course a compiler is available as well allowing you to produce rather fast machine code from your X11-Basic program.

3.3 Your first X11-Basic program

Open your favorite editor and type the following line of code into the editor.

```
PRINT "Hello X11-Basic!"
```

Now save the file as "’hello.bas"’ and run the interpreter with

```
xbasic hello.bas
```

X11-Basic should not complain. If it does, check carefully for typing mistakes. The Program now should print out your hello message at the console or in the console window the interpreter was started from. It will not return to the shell, but just prompt for aditional commands. Now type

```
> quit
```

and you return to the shell.

Of course you can include the quit command in your hello.bas:

```
PRINT "Hello X11-Basic!"
QUIT
```

Now the program always returns to the shell promt when done.

3.4 General Syntax

Appending lines

Since for a structured programming language like X11-Basic each line must not contain more than a single command, some restrictions apply for the program code.

With many editors a limitation on the maximal line length applies (e.g. 4096 characters/line). In X11-Basic a single command may in very rare cases consist of more than 4096 characters (e.g. by assigning an array constant to an array). Therfor a possibility of plitting lines into two (or more) has been implemented. If the last character of a line is a '\' (it must be really the last character of the line and may not be followed by a space character!), the following line will be appended to this line by replacing the '\' and the following newline character by spaces.

Example:

```
PRINT "Hello,";   \
      " thats it"
```

will be treated as:

```
PRINT "Hello,";" thats it"
```

Please note: The '\' character must be placed at a position within the command where a space would be allowed, too.

Comments

A comment can be inserted into your program code with the REM command or the abbreviation '. Also the '#' as a first character of the program line reserves the rest of the line for a comment. Anything behind the REM will be ignored by X11-Basic.

The REM stands for remark and is a generic comment. Comment your programs and you will be able to understand it later.

If you want to place comments at the end of a line, they have to be prefaced with '!'.

Example:

```
' This is a demonstration of comments
DO    ! endless loop
LOOP  ! with nothing inside
```

These end of line comments can not be used after DATA (and REM).

3.5 The very BASIC commands: PRINT, INPUT, IF and GOTO

The **PRINT**-command is used to put text on the text screen. Text screen means your terminal (under Unix) or the console window (under Windows). PRINT is used to generate basic output, e.g. text, strings, numbers, e.g. the result of a calculation. Some basic formatting is possible.

With the **INPUT** command you let the user input data, p.ex. numbers or text.

The **IF** command let the program do different things depending on the result of a calculation.

With **GOTO** you can branch to a different part of your program. GOTO, despite its bad reputation ([goto considered harmful]), has still its good uses.

Besides these four very basic commands (which are really standard basic commands, and you can already write very handy calculations with only these four commands) X11-Basic has many more features which make life easier and your programs more user friendly.

3.6 Variables

Variables in BASIC programming are analogous to variables in mathematics. Variable identifiers consist of alphanumeric strings. These identifiers are used to refer to values in computer memory. In the X11-Basic program, a variable name is one way to bind a variable to a memory location; the corresponding value is stored as a data object in that location so that the object can be accessed and manipulated later via the variable's name.

X11-Basic uses two scopes for variables: global (which is the default) and local.

Global variables can be modified from anywhere within the program, and any part of the program may depend on it. A global variable therefore has an unlimited potential for creating mutual dependencies, and adding mutual dependencies increases complexity. However, in a few cases, global variables can be suitable for use. They can be used to avoid having to pass frequently-used variables continuously throughout several functions, for example.

The use of global variables makes software harder to read and understand. Since any code anywhere in the program can change the value of the variable at any time, understanding the use of the variable may entail understanding a large portion of the program. They can lead to problems of naming because a global variable makes a name dangerous to use for any other local or object scope variable. A local variable of the same name can shield the global variable from access, again leading to harder to understand code. The setting of a global variable can create side effects that are hard to understand and predict. The use of globals make it more difficult to isolate units of code for purposes of unit testing, thus they can directly contribute to lowering the quality of the code.

Because of all this, X11-Basic also provides local variables, which live only within a certain function or procedure and their context.

You can refer to a variable by giving its name in the place you want the value of the variable to be used. X11-Basic will automatically know where to store the data and how to deal with it. It is also important to tell X11-Basic what sort of data you want to store. You can have variables that store only numbers but also variables that deal with a character or a whole string, a line of text for example. The following valid line of X11-Basic code will create a variable called x for you and assign it the value of 10.

```
x=10
```

You learn here 3 important things. First your variable has a name, x in this case, secondly you state that you want to do something with this variable, an assignment with the = sign, the last thing is what you want to do with this variable. You give it the value 10 here. Such an assignment will overwrite any old data that has been stored before in that variable. As long as you don't assign a value to a variable, it will hold a default value, 0 in most cases.

3.6.1 Data types

Now how you see that this variable x stores a number? How does X11-Basic know that you want to deal with a number? Easy - by the way the name of the variable has been written. To distinguish between different ways of data types X11-Basic appends a special typing sign as a suffix to the variable name to distinguish between several ways to store data in variables.

The X11-Basic interpreter uses 64-bit floating point variables, 32-bit integer variables, character strings and arrays of these variables of arbitrary dimension. A declaration of the variables and of their type is not necessary (except for arrays –> `DIM`), because the interpreter recognizes the type of the variable from the suffix: Integer variables have the suffix %, character strings a $, arrays a (). Variables without suffix are treated as float. Pointers are integers, function calls are marked by @. Logical expressions are also of type integer. It is important that variables with a special suffix are different from those without.

The integers are not yet fully implemented. So please do not use them at the moment.

3.6.2 Variable naming

You can use all letters and numbers for your variable names. Spaces are not allowed but underscores inside the variable name. In general X11-Basic can distinguish upto 64 characters[1] per name. X11-Basic limits you only in the following ways: a variable may not begin with a number or an underscore, only with letters. Avoid to name your variables like X11-Basic commands. It will work but it can also cause troubles while typing. As a rule, never try to assign values to X11-Basic system variables (like TRUE, FALSE, TIMER, PC, TERMINALNAME$). The values indeed will be assigned, but you never can use the assigned values, since always the internal values will be used.

Valid variable names look like the following: x, auto%, lives%, bonus1%, x_1, city_name$, debit Invalid variable names look like this and X11-Basic will complain: _blank, 1x, ?value%,

[1] Names of variables are limited to a number of characters or digits specified in `defs.h` (default 64).

5s$. Always remember: begin your variable names with a letter from A-Z and you are on the safe side!

Variable names and commands are case insensitive. Each name is bound to only one kind of variable; `A$` is a whole different variable(value) than `A` which is different from `A(1,1)` or `A$(1,1)`.

Space between commands will be ignored, but note that no space is allowed between the name of a variable or command and the '(' of its parameter list. So, `ASC("A")` is good, `ASC(      "A"      )` also, but `ASC ("A")` isn't.

Examples:

integer variables:	`i%=25`
	`my_adr%=varptr(b$)`
	`b%=malloc(100000)`
float variables:	`a=1.2443e17`
	`b=@f(x)`
character strings:	`t$="Hello everybody !"`
fields and arrays:	`i%(),a(),t$(), [1,3,5;7,6,2]`

3.6.3 Strings and Arrays

String variables are sequences of characters. Strings generally contain ASCII text, but can hold arbitrary byte sequences. Strings are automatically elastic, meaning they automatically resize to contain whatever number of bytes are put into them. When a string resizes, its location in memory may change, as when a longer string is assigned and there is insufficient room after the string to store the extra bytes.

A wealth of intrinsics and functions are provided to support efficient string processing.

X11-Basic arrays can contain variables of any data type, including strings. All arrays, even multi-dimensional arrays, can be redimensioned without altering the contents. A special feature of X11-Basic is the implicit dimensioning of arrays and the existance of array constants. You may define an array by using the DIM command. You might also define the array by an assignment like `a()=b()` if b() already has been DIMed or by `a()=[1,2,3,4;6,7,8,9]` asigning an array constant (In this example a 2 dimensional array will be created and the rows are separated by ';').

3.7 Arithmetics and Calculations

X11-Basic handles numbers and arithmetic: You may calculate trigonometric functions like `SIN()` or `ATAN()`, or logarithms (with `LOG()`). Bitwise operations, like `AND` or `OR` are available as well as `MIN()` and `MAX()` (calculate the minimum or maximum of its argument) or `MOD` or `INT()` (reminder of a division or integer part or a number). Many other statements give a complete set of math functions.

3.7.1 Expressions and Conditions

Expressions are needed to calculate values. The simplest expression is a numerical or string constant. More complex expressions may contain constants, variables, operators, function calls and possibly parentheses. The expression format used by X11-Basic is identical with that of many other BASIC packages: The operators have precedence of the usual order and you can alter the order of operator evaluation using parentheses. Here is an example numeric expression following after a PRINT statement:

```
PRINT (X-1)*10+SIN(x)
```

Conditions and expression are the same in X11-Basic, FALSE is defined as 0 and TRUE as -1. Those definitions are defined in defs.h and could be changed, but that is not recommended. Because BASIC doesn't have seperate boolean operators for conditions and expressions, using an boolean operator (AND,OR,XOR,NOT) may give spurious results. When on each site a value of an 0 or -1 is used, it's assumed as an condition, if it's not, it considered an expression. Problems could occure if you use boolean operators with negative numbers, but that is also unrecommended because the outcome of such an expression is highly platform depended.

3.7.2 Operators

X11-Basic provides operators for numerical expressions, character strings and arrays of either type and any dimension.

Numerical Operators

Numerical operators are roughly categorized in following categories:

- arithmetical operators: `^ * / + -`
- comparison operators: `= <> < > <= >=`
- logical operators: `NOT AND OR XOR ...`

X11-Basic recognizes the following operators, in order of falling precedence (the precedence of BASIC operators affects the order of expression evaluation):

Order	Operator	Description
1	`( )`	Parenthetical expression
2	`^`	Exponent
3	−	Sign (negation)
3	+	Sign
4	NOT	Bitwise not
5	/	Divide
5	*	Multiply
5	\	Integer division
5	MOD	Modulus (rest of division)
6	+	Add
6	-	Subtract
7	<<	Bitwise shift to the left
7	>>	Bitwise shift to the right
8	=	Logical "equals"
8	<>	Logical "not equal"
8	<	Logical "less than"
8	>	Logical "Greater than"
8	<=	Logical "less than or equal"
8	>=	Logical "greater than or equal"
9	AND	Bitwise and
9	OR	Bitwise or
9	XOR	Bitwise xor
9	IMP	Implies
9	EQV	Equivalence
10	=	assignment

Operators for Character Strings

+ conjunction, links two strings together.

Example: suppose `a$="X11"`, `b$="-"` and `c$="BASIC"`, so `d$=a$+b$+c$` results in `"X11-BASIC"`.

< <= = => > comparison functions belong to numerical (boolean) functions because the result is a number.

& the eval operator evaluates command or expression which is given by the String.

Rules for comparison of strings:

1. Two strings are equal if all the characters inside are identical (also spaces and punctuation marks).
   Example:
   `" 123 v fdh.-," = " 123 v fdh.-,"`

2. The comparison of size operates also character by character until one of them is smaller or a the strings ends first, this is the smaller one.
   Examples:

```
"X11">"X11"     result:  0
"X11"<"x11"     result: -1
"123"<"abc"     result: -1
"123">"1234"    result:  0
```

The eval-Operator &: The &-operator followed by a string evaluates it for program code.
Example:

```
REM generate ten times the command 'print a$'
CLR i
a$="print a$"
label1:
INC i
IF i>10
  b$="label2"
ELSE
  b$="label1"
ENDIF
&a$
GOTO &b$
label2:
END
```

To program like this can produce a really unreadable code but this is BASIC.

3.7.3 String processing

X11-Basic has the usual functions to extract parts from a string: `LEFT$()`, `MID$()` and `RIGHT$()`.

If you want to split a string into tokens you should use the commands `WORT_SEP` or `SPLIT`.

There is quite a bunch of other string-processing functions like `UPPER$()` (converting to upper case), `INSTR()` (finding one string within the other), `CHR$()` (converting an ascii-code into a character), `GLOB()` (testing a string against a pattern) and more.

Array operators

+ : Addition element by element - : subtraction element by element * : Array/Matrix multiplication

muss noch uebersetzt werden ....

Spanning operators

Some operators operate in between of different classes of operands, like the string comparison operators produce a number, or also the array comparison operators produce numbers. Her are ...

3.8 Program structure

An X11-Basic program consist of a main program block and subroutines. The main program block is the shell of the program and is the section between the first line and the keyword END (or QUIT). The code in the main block drives the logic of your program. In a simple programs this is that is needed. In larger and more complex programs, putting all your code in the main block makes the program hard to read and understand. Subroutines let you divide your program in manageable sections, each performing its own specific, but limited, tasks.

In X11-Basic there are two types of subroutines: procedures and functions. The main difference between the two is that a function returns a single value and can be used in expressions, while a procedure never returns a value. A procedure or function must appear after the main program block. Therefore, the structure of an X11-Basic program is as follows:

```
Main program block
END
Procedures and Functions
```

Procedures are blocks of code that can be called from elsewhere in a program. These subroutines can take arguments but return no results. They can access all variables available but also may have local variables (–> `LOCAL`).

Functions are blocks of code that can be called from elsewhere within an expression (e.g `a=3*@myfunction(b)`). Variables are global unless declared local. For local variables changes outside a function have no effect within the function except as explicitly specified within the function. Functions arguments can be variables and arrays of any data types. Functions can return variables of any data type. By default, arguments are passed by value.

3.8.1 Procedures

A procedure starts with the keyword PROCEDURE followed by the procedure name and the parameters being passed to the procedure. All procedures must end with the keyword RETURN. Procedures use the following format:

```
PROCEDURE ProcName(parameters)
  LOCAL vars
  procedure logic
RETURN
```

The parameters of the subroutine are placed between parenthesis behind the subroutine name and must be in the same order as the procedure call from the main program. All variables used within the subroutine should be declared local using the LOCAL statement. The rest of the procedure determines the task the subroutine must perform.

A procedure can be called in two ways: by using the keyword GOSUB or @. For instance, the procedure progress(), which showsa progress bar on the text console given the total amount a and the fraction b, can be called the following ways:

```
GOSUB progress(100,i)
@progress(100,i)

PROCEDURE progress(a,b)
  LOCAL t$
  IF verbose
    PRINT chr$(13);"[";string$(b/a*32,"-");">";
    PRINT string$((1.03-b/a)*32,"-");"| ";str$(int(b/a*100),3,3);"% ]";
    FLUSH
  ENDIF
RETURN
```

3.8.2 Functions

A function starts with a FUNCTION header followed by a function name, and ends with the keyword ENDFUNCTION. The function is either a numeric or a string function. A numeric function defaults to the floating point data type and needs no postfix. A string function returns a string and the function name ends with a $ postfix. A function must contain at least one RETURN statement to return the function value. Functions use this format:

```
FUNCTION FuncName[$](parameters)
  LOCAL vars
  function logic
  RETURN value[$]
ENDFUNCTION
```

The type of the return value must match the function type. A string function must return a string and a numeric function a numeric value. The numeric value is always converted to a floating point variable. The function returns to the caller when the RETURN statement is executed. The ENDFUNCTION statement only indicates the end of the function declaration and will cause an error if the program tries to execute this statement.

A function is called by preceding the function name with @. As an example, the string function Copy$() is called as follows:

```
Right$=@Copy$("X11-Basic",4)
```

where the function Copy$() might be defined as:

```
FUNCTION Copy$(a$,p)
  LOCAL b$
  b$=MID$(a$,p)
  RETURN b$
ENDFUNC
```

Of course you are as well free do define

```
FUNCTION Copy$(a$,p)
  RETURN MID$(a$,p)
ENDFUNC
```

instead.

An alternative for FUNCTION is the DEFFN statement[1], which defines a one line function. The function Copy$() used in the example above, might be used in a DEFFN statement as well:

```
DEFFN Copy$(a$,p)=MID$(a$,p)
```

In contrast with procedures and functions, DEFFN functions may be placed within a procedure or function body, although it doesn't use the local variables of the subroutine.

3.8.3 Parameters and local variables

Any X11-Basic variable type can be passed to a procedure or function.

By default all parameters are passed "'by value"'. Though parameters can also be passed "'by reference"' by using the VAR statement[2].

The keyword VAR precedes the list of variables that are being passed as call by reference parameters. These variables should always be listed at the end of the parameter list in the procedure or function heading. The difference between the two is that a call by value parameter

[1] This is currently not implemented to X11-Basic.

[2] This is currently not implemented in X11-Basic

gets a copy of the passed value and a call by reference does not. A VAR variable references the same variable that is passed to the subroutine. The original variable will change when a subroutine modifies the corresponding VAR variable. In fact, both variable names reference the same piece of memory that contains the variable value.

Internally, X11-Basic maintains a list of all variables. Each entry in the list points to a memory location that contains the variable value. A call by reference variable points to the same location as the passed variable. Therefore, constants or expressions can not be passed to a VAR variable.

A copy of an array cannot be passed directly to a subroutine[1]; arrays should always be passed as a VAR. X11-Basic swaps the pointers to the array descriptors, when an array is passed to a parameter array. The following example shows a simple function, which searches a name in a given string array:

```
Idx=SearchName("Jack",Name$())

FUNCTION SearchName(n$,VAR N$())
  LOCAL Idx
  WHILE NOT N$(Idx)=n$
    INC Idx
  WEND
  RETURN Idx
ENDFUNC
```

The locally used array N$() references the global array Name$(). The array N$() is only valid within the procedure, where it points to the descriptor of the Name$() array. Trying to access N$() outside the procedure and Name$ within the procedure will cause an error.

The LOCAL statement lists the variables only known to a procedure or function. Subroutine parameters are local variables as well. When a subroutine calls another subroutine the local variables of the calling routine are known in the called routine as if they where global variables.

Several local variables separated by commas may be listed after the LOCAL statement. Multiple LOCAL lines are allowed.

3.8.4 The address of a procedure

Some external linked shared libraries require an address of a function that can be called from within the shared library (or other objects of code, which can be linked to the X11-Basic interpreter or compiled excecutable, if it is compiled (callback functions). X11-Basic provides the _CB()= command to generate a pointer to the entry point of a procedure[2].

[1] This may be possible in future versions of X11-Basic

[2] Not yet implemented to X11-Basic

3.9 Simple Input/Output

3.9.1 Printing data to the console

You actually already know a X11-Basic command to write data on screen. This command is PRINT. It is very versatile and you can extend it in various ways.

Syntax is simple, PRINT <data> where <data> is whatever sort of data you want to print on screen. That can be variables, numbers, the result of a calculation, a string or a mix of them all. You can even special user defined commands (using functions) to your PRINT statement for screen control such as cursor positioning. A few examples for the PRINT command can be found here:

```
PRINT 10+5
PRINT x%
PRINT 10;20;30
PRINT 10,20,30
PRINT "Hello!"
PRINT "y= ";y
PRINT "x=";x;" y=";y;" z=";z
PRINT "Your name is ";nam$
PRINT AT(5,5);"AT() is one of my favorites"
PRINT CHR$(27);"[2J This is a cleared console..."
```

These are the most simple variations of the PRINT command. Try them out on your own to see their effects. Try with defining a few sample variables.

Now why do you write PRINT "y =";y instead of PRINT "y =",y? Using ; will add the following data directly behind your text without altering the cursor position while the , will advance the cursor to the next vertical tabular position. You can use that to align your data in tables on screen. In short, if you want to write data directly to some sort of prompt or behind some text, use the ; notation. Put a ; as the last data on your PRINT statement to let the cursor stay on the current line. You can use this to prevent a scrolling on the last line of the screen or if you simply want to split writing of prompt and data into two lines of code. Technically speaking giving the ; last will suppress a carriage return.

If you want to write direct text onto screen which is not contained in a string variable, just enclose it in double quotes " like PRINT "Hello!" which will write Hello! on screen.

3.9.2 Screen control

Now that you know how to write your data on screen, you will also want to know how to handle screen output in detail. How do I leave a line of text blank might you ask? Write simply PRINT without any data behind to output a blank line on screen. Try this 3 lines program:

```
PRINT "Hello!"
PRINT
PRINT "This is the first example for screen control!"
```

As you see it prints the greeting and the other line with an empty line between.

A very important thing is how to clear the screen. For obvious reasons, you'll sometimes prepare a screen layout that requires you not to have other text or old data on screen. You'll simply clear the screen with the following command.

```
CLS
```

A neat thing is to write on screen exactly on a position where you want and not following the listed flow of ordinary `PRINT` statements. If you did try the first example in this chapter, you have noticed the `AT()` command. This special addition for `PRINT` allows you to position the cursor freely on screen so you can write your data where you want. Let's try the following example program:

```
CLS
PRINT AT(1,1);"Top left"
PRINT AT(5,13);"Middle line, text indented 5 chars"
PRINT AT(20,25);"bottom line";
INPUT "",dummy$
```

Press the RETURN key to terminate the program. You'll learn soon about the `INPUT` statement.

Syntax for `PRINT AT();` is `PRINT AT(column, row);`, where row 1 is on top of the screen and column 1 on the left end. Column and row can be variables, expressions or simply a plain number. Valid `PRINT AT()` commands are:

```
PRINT AT(1,5);"Hello"
PRINT AT(5+x%,10);"x"
PRINT AT(4+8,y%);"y = "
```

How many character positions do I have at my hand you may ask. This depends on the current console screen size. You have almost always at least 24 lines of text. 80 columns are standard. If you want to exactly know the number of rows and collums of the text screen, you can use the (system) variables `ROWS` and `COLS`.

```
> PRINT ROWS,COLS
24      80
```

There are more commands you can use with `PRINT` like `SPC()` and `TAB()`. Refer to the command reference on them.

3.9.3 Formatting output with PRINT USING

There are built in commands for formatting data on output. I will only explain how to use that for numeric data. Refer to your command reference to learn how to apply it for strings.

Generic syntax is `PRINT USING "<format string>",expression`. The format string defines how you want your data to be formatted on screen while expression is what you want to write with this format. Expression can be a number, a variable or a full expression.

3.9.4 Gathering user input

Finally you can make your program interactive and allow the user to enter data into the computer. To understand this sub chapter, you have to fully understand variables as explained in chapter 2 because all user input is stored in variables.

To enter numbers into the computer, use the `INPUT` statement in one of the following ways. Your entered data is stored in the named variable. If you give a string variable, you can enter text while you can only enter numeric data if you use a numeric variable. A minus sign and optional decimal point are allowed for numeric input.

```
INPUT "x= ",x
INPUT "What is your name? ",your_name$
```

This will prompt the user to enter a value for x which will be stored into a float variable. You can then use this variable in your program as normal, doing calculations with it. Your program will stop until the RETURN key or the ENTER key has been pressed to terminate the input.

You can read more than one variable with one `INPUT` statement, just list your variables where you want your input to go to with separating commata.

```
PRINT "enter 3 values, separated with commata (eq 3,4,5):"
INPUT x%,y%,z%
```

The user has then to enter commata at the appropriate places to tell which input goes to which variable. To the example above the user would respond with 5,6,7. As always you can then reuse the entered values in expressions and calculations.

```
CLS
INPUT "Enter a value for x:",x
PRINT "x = ";x
INPUT "What is your name?",your_name$
PRINT "Your name is ";your_name$;"."
PRINT "Bye, ";your_name$;"!"
```

While entering strings you may have already noticed that X11-Basic will treat entering a comma again as a delimiter, effectively cutting your string at that comma. Use the command LINEINPUT instead of INPUT to read strings.

```
LINEINPUT txt$
```

You can now enter strings with a comma in and it will be saved to the string variable as well. You can read multiple strings with LINEINPUT as well but the user has to press the RETURN key terminating each string to be entered.

An INPUT statement will erase the former contents of a variable in any case and will place the entered values in them.

3.10 Flow Control

This time you'll finally make your programs do things more then once without having to retype your code. The creation of so called loops is essential for making complex programs work. The concept of looping and simple counting loops

Before going further let me explain you the fundamental idea of looping. The idea is to make your program repeat a section of code for a defined amount of time. You may let X11-Basic count a variable for you and you can then use the value of that variable in an ongoing calculation. Or you can let X11-Basic loop a certain part of code until a special condition has been met. Take a look at the following sample program:

```
FOR i%=1 TO 5
  PRINT i%
NEXT i%
```

This little example program loops 5 times and counts the variable i% from 1 to 5 and prints the current value to the screen. This sort of loop is called a FOR-NEXT-loop. You can use any numerical variable to count. Most often this sort of loop is used to do things a certain amount of time or to iterate over a list. The loop will repeat the code between the FOR and its corresponding NEXT. Each time X11-Basic reaches the NEXT, it will increment the count variable and will stop the loop if the maximum count has been reached.

You can of course have another loop inside the current one. Just make sure not to use the same variable for counting or X11-Basic will do unpredictable things:

```
FOR i%=1 TO 5
  FOR j%=1 TO 10
    PRINT i%;" * ";j%;" = ";i%*j%
  NEXT j%
NEXT i%
```

That sample program has one FOR-NEXT-loop in another and it calculates the product of the both counter variables creating some sort of multiplication table. Some rules and advice to keep in mind with FOR-NEXT-loops:

1. Always terminate an opened `FOR` with a corresponding `NEXT`.
2. Always terminate FOR-loops in the correct order. If you write `FOR i%=...` first and `FOR j%=...` next, make sure to terminate the inner loop first.
3. You can count downwards with the word `DOWNTO` instead of `TO`. Try `FOR i%=5 DOWNTO 1`.
4. You can count in steps not equal 1 with the keyword `STEP`: `FOR i%=1 TO 10 STEP 2` That will increment i% in steps of 2 until it reaches 10.
5. X11-Basic will check for correct loop termination while entering the code into the editor.
6. You can terminate the FOR-NEXT-loop with the `EXIT IF` statement.

Conditions

A very fundamental idea in programming is to create and use conditionals. These will allow you to make decisions when certain conditions are met and let your program take an alternative code segment.

Try to imagine that you count a special variable and want to do something else when the value of your counter is 5:

```
FOR i%=1 to 10
  IF i%=5
    PRINT "i% is now 5"
  ELSE
    PRINT "i% is not 5"
  ENDIF
NEXT i%
```

This program loops 10 times and counts in the variable i%. For each iteration of the loop it checks if i% is 5 in the IF line. If that condition is true, i% is 5, then it executes the program branch until the ELSE and omits the following part. If the condition is not true, X11-Basic will only execute the part behind the ELSE. Make sure to terminate each `IF` conditional with an `ENDIF` or X11-Basic will get lost and produce an error message.

You may leave out the `ELSE` fork. X11-Basic will then do nothing if the condition is not true.

3.10.1 Conditional and endless loops

Sometimes you don't know how far you need to count for a special operation. Or imagine a game. You don't want to let it run just for 10 frames but until the player sprite did collide or something like that. The first new loop will loop until a condition is fulfilled:

```
REPEAT
...
UNTIL <condition>
```

This is a so called REPEAT-UNTIL-loop. It loops at least once and checks for the condition after the loop contents have been executed by X11-Basic. Use it for things that need to be done at least once. You can emulate FOR-NEXT-loops with it if you want trickier counting:

```
i%=1
REPEAT
  PRINT "i%=";i%
  i%=i%+1
UNTIL i%>5
```

Surely you can test the condition before entering a loop. This is useful if you want to loop only when a certain condition is already true:

```
WHILE <condition>
...
WEND
```

This is the so called WHILE-WEND loop. It checks the condition first and it will not execute the loop body if the condition is not fulfilled. Sometimes you want to loop endless. X11-Basic has a special loop construct for this purpose although you can create never ending loops easily with the types above if you use a condition that will never get true. The never ending loop is called DO-loop. The 3 loops in the example are all equal in functionality and will loop endless.

```
DO
  PRINT "endless"
LOOP

i%=0
REPEAT
  PRINT "endless"
UNTIL i%=1
```

```
i%=0
WHILE i%=0
  PRINT "endless"
WEND
```

At this point it is important that you know you can terminate at your X11-Basic program at any point. This is useful if your program gets stuck in an endless loop which was not intended. Press CONTROL-c together and X11-Basic will stop the program. Another CONTROL-c will quit the interpreter.

Sometimes you will want to terminate a running loop at another point than the official loop beginning or loop end. Use the `EXIT IF` statement in your loop for extra conditions. This will also terminate FOR-NEXT-loops if you wish to and it is the only way to terminate a DO-LOOP.

```
i%=1
DO
  PRINT "i%=";i%
  EXIT IF i%=5
  i%=i%+1
LOOP
```

Please note that the `EXIT IF` statement has no `ENDIF` or the like. It just terminates the loop and continues your program behind the loop end.

3.11 Adress spaces

The full accessible Program memory can be accessed by `PEEK/POKE`, `LPEEK/LPOKE`, `DPEEK/DPOKE`. Be careful. You can manipulate all symbols of the interpreter and or dynamically linked libraries and your program. Adressspaces belonging to other programs which are not shared memory blocks can not be accessed. You will get a segmentation fault on trying this.

3.12 Graphics: Drawing and painting

A graphics window will be automatically opened when the first graphic command appears in your program. Without using any graphic commands no X11-Server is needed at all and your programs also runs under a text console or as a daemon or as CGI scripts. But if you want to draw anything with e.g. `LINE`, `CIRCLE` or `BOX`, control the MOUSE pointer, the keyboard or use the graphical user interface with e.g. `ALERT` or `MENU`, a graphic window will open with the default geometry `640x400`. All graphic output can be done in full color which can be set with the `GET_COLOR()` and the `COLOR` statements. Moreover, there can be up to 16 different graphic windows opened at a time. Please note that all graphics is displaied after a `SHOWPAGE` command only. This allows fast animations.

To allow for some animated bitmap graphics, X11-Basic offers the commands `GET` and `PUT`, which retrieve rectangular regions from the graphics-window into a string or vice versa.

3.13 Reading from and writing to files

Before you may read from or write to a file, you need to open it; once you are done, you should close it. Each open file is designated by a simple number, which might be stored within a variable and must be supplied to the `PRINT` and `INPUT` commands if you want to access the file.

If you need more control, you may consider reading and writing one byte at a time, using the multi-purpose commands `INP()` and `OUT`, or reading the whole file as a binary block with `BLOAD`.

3.14 Internet connections, special files and sockets

X11-Basic allows to connect a program to another Program on a different (or the same) host computer via standart internet protocols or pipes.

Basically there are two methods of connections to other computers on a network: The TCP/IP Based connections via strams and the implemention of a connectionless, unreliable datagram packet service (UDP).

A method of passing data between two applications on the same computer is using Pipes. Pipes are special files which are created in the local filesystem.

3.14.1 Local communication: Pipes

3.14.2 World-Wide communication: Sockets

Most interprocess communication uses the client server model. These terms refer to the two processes which will be communicating with each other. One of the two processes, the client, connects to the other process, the server, typically to make a request for information. A good analogy is a person who makes a phone call to another person.

Notice that the client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established. Notice also that once a connection is established, both sides can send and receive information.

When a socket is created, the program has to specify the address domain and the socket type. Two processes can communicate with each other only if their sockets are of the same type and in the same domain. There are two widely used address domains, the unix domain, in which two processes which share a common file system communicate, and the Internet domain, in which two processes running on any two hosts on the Internet communicate. Each of these has its own address format.

The address of a socket in the Unix domain is a character string which is basically an entry in the file system.

The address of a socket in the Internet domain consists of the Internet address of the host machine (every computer on the Internet has a unique 32 bit address, often referred to as its IP address). In addition, each socket needs a port number on that host. Port numbers are 16 bit unsigned integers. The lower numbers are reserved in Unix for standard services. For example, the port number for the FTP server is 21. It is important that standard services be at the same port on all computers so that clients will know their addresses. However, port numbers above 2000 are generally available.

Socket Types There are two widely used socket types, stream sockets, and datagram sockets. Stream sockets treat communications as a continuous stream of characters, while datagram sockets have to read entire messages at once. Each uses its own communciations protocol. Stream sockets use TCP (Transmission Control Protocol), which is a reliable, stream oriented protocol, and datagram sockets use UDP (Unix Datagram Protocol), which is unreliable and message oriented.

TCP/IP Transmission Control Protocol (TCP) provides a reliable byte-stream transfer service between two endpoints on an internet. TCP depends on IP to move packets around the network on its behalf. IP is inherently unreliable, so TCP protects against data loss, data corruption, packet reordering and data duplication by adding checksums and sequence numbers to transmitted data and, on the receiving side, sending back packets that acknowledge the receipt of data.

Before sending data across the network, TCP establishes a connection with the destination via an exchange of management packets. The connection is destroyed, again via an exchange of management packets, when the application that was using TCP indicates that no more data will be transferred. In OSI terms, TCP is a Connection-Oriented Acknowledged Transport protocol.

TCP has a multi-stage flow-control mechanism which continuously adjusts the sender's data rate in an attempt to achieve maximum data throughput while avoiding congestion and subsequent packet losses in the network. It also attempts to make the best use of network resources by packing as much data as possible into a single IP packet, although this behaviour can be overridden by applications that demand immediate data transfer and don't care about the inefficiencies of small network packets.

The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. A socket is one end of an interprocess communication channel. The two processes each establish their own socket.

The steps involved in establishing a socket on the client side are as follows:

1. Create a socket with the OPEN command providing a port number

   ```
   open "US",#1,"client",5000
   ```

2. Connect the socket to the address of the server using the CONNECT command

   ```
   connect #1,"ptbtime1.ptb.de",13
   ```

3. Instead of using Steps 1 and 2, you can alternatively use the combined command:

   ```
   open "UC",#2,"ptbtime1.ptb.de",13
   ```

4. Send and receive data. There are a number of ways to do this, but the simplest is to use the PRINT, SEND, WRITE, READ, RECEIVE INPUT commands.

   ```
   print #2,"GET /index.html"
   flush #2
   while inp?(#2)
     lineinput #2,t$
     print "got: ";t$
   wend
   ```

5. close the connection with

   ```
   close #1
   ```

The steps involved in establishing a socket on the server side are as follows:

1. Create a socket with the OPEN command and bind the socket to a port number on the host machine.

   ```
   open "US",#1,"server",5000
   ```

2. Listen for connections with

3. Accept a connection with another OPEN command.

   ```
   open "UA",#2,"",1
   ```

   This call typically blocks until a client connects with the server.

4. Send and receive data on the accepted connection

   ```
   print #2,"Welcome to X11-Basic test-server ..."
   flush #2
   do
     if inp?(#2)
       lineinput #2,t$
       print "got: ";t$
     endif
     exit if t$="quit"
   loop
   print #2,"goodbye..."
   flush #2
   ```

5. close the established connection with

   ```
   close #2
   ```

   and listen to the next connection or

6. close the socket if not further needed

   ```
   close #1
   ```

UDP User Datagram Protocol (UDP) provides an unreliable packetized data transfer service between endpoints on an internet. UDP depends on IP to move packets around the network on its behalf.

First a SOcket has to be crated with the OPEN command:

```
open "UU",#1,"sender",5556
```

When a UDP socket is created, its local and remote addresses are unspecified. Datagrams can be sent immedi ately using SEND with a valid destination address and port as argument:

```
send #1,"This is my message",mkl(chr$(131)+chr$(195)+chr$(15)+chr$(200)),5000
```

UDP uses the IPv4 address format, so a long integer has to be passed.

When CONNECT is called on the socket the default destination address is set and datagrams can now be sent using SEND without specifying an destination address. It is still possible to send to other destinations by passing an address to SEND.

```
connect #1,"localhost",5555
send #1,"This is my message"
```

All receive operations return only one packet.

```
if inp?(#1)
  receive #1,t$,adr
  print "Received Message: ";t$;" from ";hex$(adr)
endif
```

INP?(#n) Returns the size of the next pending datagram in bytes, or 0 when no datagram is pending.

The Socket should be closed when the connection is not goint to be used any more:

```
close #1
```

UDP does not guarantee to actually deliver the data to the destination, nor does it guarantee that data packets will be delivered to the destination in the order in which they were sent by the source, nor does it guarantee that only one copy of the data will be delivered to the destination. UDP does guarantee data integrity, and it does this by adding a checksum to the data before transmission.

3.15 Data within the program

You may store data within your program within DATA-statements; during execution you will probably want to READ it into variables or arrays. Also the assignment of constant to arrays may be used to store data in your program and last but not least the `INLINE$()` function may be used to store huge binary data segments.

The first example shows how to store conventional data (numbers and strings) within the sourcecode of a basic program:

```
' example how to use the DATA statement

RESTORE mydata
READ name$,age,address$,code

mydata:
DATA "Bud Spencer",30,"Holywood Street",890754
DATA "Hannelore Isendahl",15,"Max-Planck-Allee",813775
```

The following example shows hot to store arbitrary binary data, which can be used e.g. to store the bitmapdata for a bitmap ( ).

```
' output of inline.bas for X11-Basic 23.04.2002
' demo 104 Bytes.
demo$=""
demo$=demo$+"5*II@V%M@[4D=*9V,)5I@[4D=*9V,(IR?*IR=6Y*A:]OA*IS?F\.&IAI?J\D8ZII"
demo$=demo$+",*5M=;1I@V%P=;1I?F%OaJ]R=:\P,*5E?J\D>*)X,*9W,*AI>ZUE@+%X/F\R&JAV"
demo$=demo$+"A;1W&HXR&DL$"
a$=INLINE$(demo$)
PRINT len(a$),a$

' show a bitmap
biene$="($$43$%*<(1G,=E5Z&MD%_DVW'b*%H-^,EQ6>VTL$$$$"

CLEARW
t$=INLINE$(biene$)
COLOR GET_COLOR(65535,65535,65535)
FOR i=0 TO 40
  PUT_BITMAP t$,i*16,0,16,16
NEXT i
```

3.16 Dynamic-link libraries

A dynamic-link library (`.so` =*shared object*) is a collection of functions (subroutines) that can be used by programs or by other `.so`'s. A `.so` function must be called, directly or indirectly, from a running application and can not be run as a separate task.

Dynamic link libraries save memory space and reduce memory swapping. Memory is saved, because many applications can use a single `.so` simultaneously, sharing a single copy of the `.so` in memory. Another feature of `.so`'s is the ability to change the functions in a `.so` without modifying the applications that use them, as long as the function's arguments and return values do not change. A disadvantage to using `.so`'s is that an application depends on the existence of a separate `.so` module. If the `.so` is not found, the application is terminated.

All documented functions from the shared objects of other software packages can be used and invoked from within yout X11-Basic program.

X11-Basic will perform no check on the number and type of the API function parameters.

3.16.1 Using shared libraries and C functions

Before an application can use a function from a `.so` (if you want to use your own functions written in C you have to compile them to a shared object file), it must load the `.so` explicitly using the `LINK` statement.

```
LINK #n,"myfile.so"
```

The process of loading a `.so` explicitly is called run-time linking.

For instance, to use the `binit()` function from the `trackit.so` library, an application must include following lines of code (supposing, you want to use your own shared object made out of the c-code trackit.c):

```
IF not exist("./trackit.so")
  system "gcc -O3 -shared -o trackit.so trackit.c"
ENDIF
LINK #11,"./trackit.so"
~EXEC(SYM_ADR(#11,"binit"),L:n,L:200,L:varptr(x(0)),L:varptr(bins(0)))
```

The file `trackit.c` contains:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

void binit(int n,int dn,double *x,double *data) {
  int i,j;
  int over=0,under=0;
    for(i=0;i<n;i++) {
      j=(int)((x[i]+PI)/2/PI*dn);
      if(j<0) under++;
      else if(j>=dn) over++;
      else data[j]++;
    }
}
```

X11-Basic applications can load up to 99 shard object files simultaneously, al though the channel number space is shared with the open files..

To do this parameter n must specify a value between 1 an 99. X11-Basic maintains an internal table with 99 entries to store the handle of the loaded shared object modules. These handles are necessary to unload the `.so` when the application is finished using them.

The `.so`'s are unloaded by invoking the `UNLINK` command:

```
UNLINK #11
```

X11-Basic currently allows only a float (double) type for the return value. This is currently a limitation for the use of the standard libraries. If you have written the library function yourself, you could bypass this limitation by passing pointers to variables.

The following parameter types are possible:

L:	32-bits integers and pointers (long) (%)
W:	16-bits signed (short)
B:	8-bits signed (char)
F:	8 byte float (double)
S:	4 byte float (float)

The `SYM_ADR` function determines the address of the function from its name. The spelling of the function name must therefore be identical to the spelling of the function in the `.so`.

When passing the address of the string, a null byte must be added to the end of the string.

3.17 Memory management

Normally, X11-Basic takes care of most of the memory management for the programmer. When a variable, string or array is declared, X11-Basic allocates the required memory and releases it when the application is terminated. However, there may be situations when a programmer wants to allocate additional memory.

3.17.1 Allocating memory

If an application needs to store small amounts of memory, it should use strings. Strings are often used as a buffer for functions. The Adress of the memory occupied by a string can be obtained by the `VARPTR()` function. Its length by the `LEN()` funciton.

To allocate memory from the global and systemwide proram user space memory pool you might use the function `MALLOC()`. For instance, to allocate 2000 bytes, you might use:

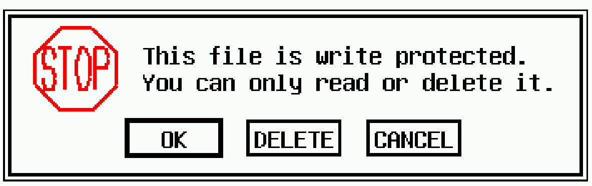
```
Ptr%=MALLOC(2000)
```

A global memory block allocated with `MALLOC` must be freed using the `FREE()` function. An application should always free all memory blocks before exiting. For instance:

```
FREE Ptr%
```

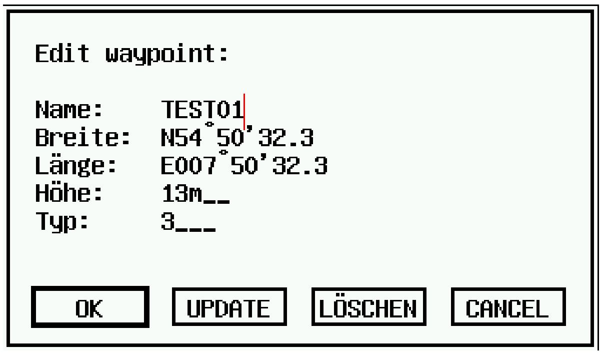
3.18 Other features

- X11-Basic programs may start other programs with the commands `SYSTEM` and `SYSTEM$()`.
- The `ENV$()` function allows access to environment variables.
- The current time or date can be retrieved with `TIME$` and `DATE$`.
- The interpreter allows self modifying code.
- It is possible to link shared library objects and use the fuctions provided from within the X11-basic Program



This file is write protected.
You can only read or delete it.

Figure 3.1: A message box.



Edit waypoint:

Name: TEST01
Breite: N54°50’32.3
Länge: E007°50’32.3
Höhe: 13m__
Typ: 3___

Figure 3.2: A simple input box.

3.19 Using the Graphical User Interface (GUI)

3.19.1 ALERT and FILESELECT

Two most often used Graphic functions are implementet as a full functional graphical user interface dialog: Message boxes and a Fileselector. Arbitrary dialogs can be createt with the object and recource functions. Also a pull down menu function is implemented.

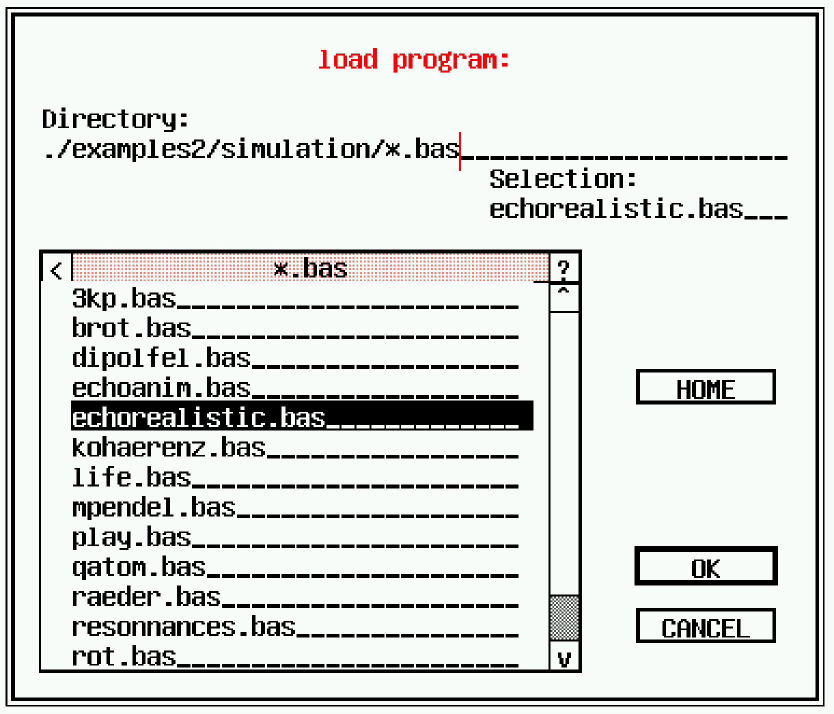
Fig. 3.1 shows a typical messagebox. The command which produces it is:

```
ALERT 3,"This file is write protected.|You can only read or
         delete it.",1,"OK|DELETE|CANCEL",sel
```

ALERT boxes can also be used to manage simple input forms like the one you can see in fig. 3.2. Here is a little exapmle program:

```
CLEARW
i=1
name$="TEST01"
posx$="N54°50’32.3"
posy$="E007°50’32.3"
t$="Edit waypoint:||Name:   "+CHR$(27)+name$+"|"
t$=t$+"Breite: "+chr$(27)+posx$+"|"
t$=t$+"Länge:  "+chr$(27)+posy$+"|"
t$=t$+"Höhe:   "+chr$(27)+str$(alt,5,5)+"|"
t$=t$+"Typ:    "+chr$(27)+hex$(styp,4,4)+"|"
ALERT 0,t$,1,"OK|UPDATE|LÖSCHEN|CANCEL",a,f$
WHILE LEN(f$)
  WORT_SEP f$,CHR$(13),0,a$,f$
  PRINT "Feld";i;": ",a$
  INC i
WEND
QUIT
```

Fig. 3.4 shows the fileselector box. The command which produces it is:



load program:

Directory:
./examples2/simulation/*.bas_____________________

Selection:
echorealistic.bas___

```
< *.bas ?
3kp.bas_______________________
brot.bas______________________
dipolfel.bas__________________
echoanim.bas__________________
echorealistic.bas_____________
kohaerenz.bas_________________
life.bas______________________
mpendel.bas___________________
play.bas______________________
qatom.bas_____________________
raeder.bas____________________
resonnances.bas_______________
rot.bas_______________________
```

Figure 3.3: The fileselcetor

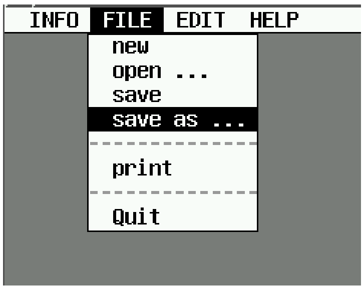


Figure 3.4: A pull down menu

```
FILESELECT "load program:","./*.bas","in.bas",f$
```

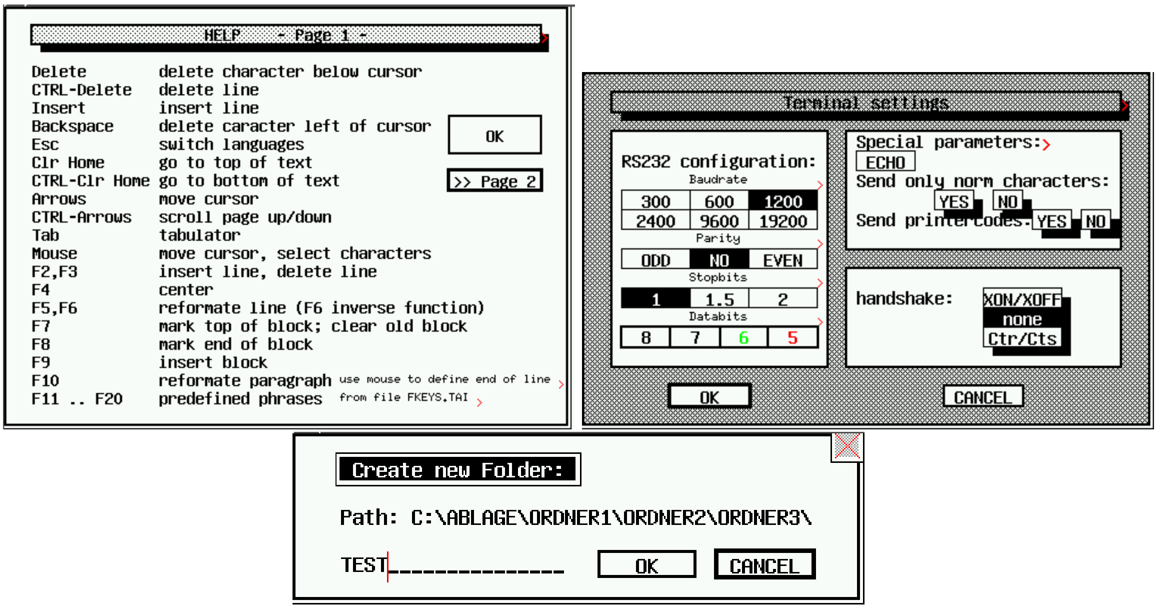
The complete path and filname of the selected file will be returned in `f$`.

3.19.2 Recources

X11-Basic resources consist of object trees, strings, and bitmaps used by a basic program. They encapsulate the user interface and make internationalization easier by placing all program strings in a single file. The dataformat of X11Basic recources is downwards compatible with the Atari-ST GEM implementation.

Resources are generally created using a Resource Construction Set (RCS) and saved to a `.RSC` file which is loaded by `RSRC_LOAD()` at program initialization time.

Resources may also be embedded as data structures in source code (the utility programs `rsc2gui.bas` and `gui2bas.bas` convert `.RSC` files to source code). Resources contain point-



```
HELP     - Page 1 -

Delete          delete character below cursor
CTRL-Delete     delete line
Insert          insert line
Backspace       delete caracter left of cursor
Esc             switch languages
Clr Home        go to top of text
CTRL-Clr Home go to bottom of text
Arrows          move cursor
CTRL-Arrows     scroll page up/down
Tab             tabulator
Mouse           move cursor, select characters
F2,F3           insert line, delete line
F4              center
F5,F6           reformate line (F6 inverse function)
F7              mark top of block; clear old block
F8              mark end of block
F9              insert block
F10             reformate paragraph use mouse to define end of line
F11 .. F20      predefined phrases from file FKEYS.TAI
```

Terminal settings

RS232 configuration:

Baudrate: 300 600 [✓ 1200] 2400 9600 19200

Parity: ODD [✓ NO] EVEN

Stopbits: [✓ 1] 1.5 2

Databits: 8 7 6 5

Special parameters:
ECHO
Send only norm characters: YES NO
Send printercodes: YES NO

handshake: XON/XOFF [✓ none] Ctr/Cts

Create new Folder:

Path: C:\ABLAGE\ORDNER1\ORDNER2\ORDNER3\

TEST

Figure 3.5: Examples of forms in X11-Basic

ers and coordinates which must be fixed up before being used. `RSRC_LOAD()` does this automatically, however if you use an embedded resource you must take care of this by yourself on each object in each object tree to convert the initial character coordinates of to screen coordinates. This allows resources designed on screens with different aspect ratios and system fonts to appear the same. Once a resource is loaded use `rsrc_gaddr()` to obtain pointers to individual object trees which can then be manipulated directly or with the X11-Basic built-in functions.

3.19.3 Objects

Objects can be boxes, buttons, text, images, and more. An object tree is an array of OBJECT structures linked to form a structured relationship to each other. The object itself is a section of data which can be held by a string in X11-Basic.

The OBJECT structure is format is as follows:

```
object$=MKI$(ob_next)+MKI$(ob_head)+MKI$(ob_tail)+
        MKI$(ob_type)+MKI$(ob_flags)+MKI$(ob_state)+
        MKL$(ob_spec)+MKI$(ob_x)+MKI$(ob_y)+MKI$(ob_width)+
        MKI$(ob_height)
```

An Object tree is a collection of objects:

```
tree$=object0$+object1$+ ... +objectn$
```

The first object in an OBJECT tree is called the ROOT object (OBJECT 0). It's coordinates are relative to the upper-left hand corner of the graphics window. The ROOT object can have any number of children and each child can have children of their own. In each case, the OBJECT's coordinates, `ob_x`, `ob_y`, `ob_width`, and `ob_height` are relative to that of its parent. The X11-Basic function `objc_offset()` can, however, be used to determine the exact screen coordinates of a child object. `objc_find()` is used to determine the object at a given screen coordinate.

The `ob_next`, `ob_head`, and `ob_tail` fields determine this relationship between parent OBJECTs and child OBJECTs.

ob_next the index (counting objects from the first object in the object tree) of the object's next sibling at the same level in the object tree array. The ROOT object should set this value to -1. The last child at any given nesting level should set this to the index of its parent.

ob_head the index of the first child of the current object. If the object has no children then this value should be -1.

ob_tail the index of the last child: the tail of the list of the object's children in the object tree array If the object has no children then this value should be -1.

ob_type the object type. The low byte of the ob_type field specifies the object type as follows:

ob_type	Name	Description
20	G_BOX	Box
21	G_TEXT	Formatted Text
22	G_BOXTEXT	Formatted Text in a Box
23	G_IMAGE	Monochrome Image
24	G_PROGDEF	Programmer-Defined Object
25	G_IBOX	Invisible Box
26	G_BUTTON	Push Button w/String
27	G_BOXCHAR	Character in a Box
28	G_STRING	Unformatted Text
29	G_FTEXT	Editable Formatted Text
30	G_FBOXTEXT	Editable Formatted Text in a Box
31	G_ICON	Monochrome Icon
32	G_TITLE	Menu Title
33	G_CICON	Color Icon

ob_flags The ob_flags field of the object structure is a bitmask of different flags that can be applied to any object. You may want to apply one ore more flags at once. Just add the values ob_flags.

ob_flags	Name	Description
0	NONE	No flag
1	SELECTABLE	object is selected. state may be toggled by clicking on it with the mouse.
2	DEFAULT	An EXIT object with this bit set will have a thicker outline and be triggered when the user presses return.
4	EXIT	Clicking on this OBJECT and releasing the mouse button while still over it will cause the dialog to exit.
8	EDITABLE	Set for FTEXT and FBOXTEXT objects to indicate that they may receive edit focus.
16	RBUTTON	This object is one of a group of radio buttons. Clicking on it will deselect any selected objects at the same tree level that also have the RBUTTON fllag set. Likewise, it will be deselected automatically when any other object is selected.
32	LASTOB	This flag signals that the current OBJECT is the last in the object tree. (Required!)
64	TOUCHEXIT	Setting this flag causes the OBJECT to return an exit state immediately after being clicked on with the mouse.
256	HIDETREE	This OBJECT and all of its children will not be drawn.
512	INDIRECT	This flag cause the ob_spec field to be interpreted as a pointer to the ob_spec value rather than the value itself.
1024	FL3DIND	Setting this flag causes the OBJECT to be drawn as a 3D indicator. This is appropriate for radio and toggle buttons.
2048	FL3DACT	Setting this flag causes the OBJECT to be drawn as a 3D activator. This is appropriate for EXIT buttons.
3072	FL3DBAK	If these bits are set, the object is treated as an AES background object. If it is OUTLINED, the outlined is drawn in a 3D manner. If its color is set to WHITE and its fill pattern is set to 0 then the OBJECT will inherit the default 3D background color.
4096	SUBMENU	This bit is set on menu items which have a sub-menu attachment. This bit also indicates that the high byte of the ob_type field is being used by the menu system.

ob_state The ob_state field determines the display state of the object as follows:

ob_state	Name	Description
0	NORMAL	Normal state
1	SELECTED	The object is selected. An object with this bit set will be drawn in inverse video except for G_CICON which will use its 'selected' image.
2	CROSSED	An OBJECT with this bit set will be drawn over with a white cross (this state can only usually be seen over a colored or SELECTED object).
4	CHECKED	An OBJECT with this bit set will be displayed with a checkmark in its upper-left corner.
8	DISABLED	An OBJECT with this bit set will ignore user input. Text objects with this bit set will draw in grey or a dithered pattern.
16	OUTLINED	G_BOX, G_IBOX, G_BOXTEXT, G_FBOXTEXT, and G_BOXCHAR OBJECTs with this bit set will be drawn with a double border.
32	SHADOWED	G_BOX, G_IBOX, G_BOXTEXT, G_FBOXTEXT, and G_BOXCHAR OBJECTs will be drawn with a shadow.

ob_spec The Object-Specific Field

The ob_spec field contains different data depending on the object type as indicated in the table below:

```
+-----------------------------------------------------------------+
|          |The low 16 bits contain a WORD containing color       |
|G_BOX     |information for the OBJECT. Bits 23-16 contain a signed|
|          |BYTE representing the border thickness of the box.    |
+-----------------------------------------------------------------+
|G_TEXT    |The ob_spec field contains a pointer to a TEDINFO     |
|          |structure.                                            |
+-----------------------------------------------------------------+
|G_BOXTEXT |The ob_spec field contains a pointer to a TEDINFO     |
|          |structure.                                            |
+-----------------------------------------------------------------+
|G_IMAGE   |The ob_spec field points to a BITBLK structure.       |
+-----------------------------------------------------------------+
|G_PROGDEF |The ob_spec field points to a APPLBLK structure.      |
+-----------------------------------------------------------------+
|          |The low 16 bits contain a WORD containing color       |
|G_IBOX    |information for the OBJECT. Bits 23-16 contain a signed|
|          |BYTE representing the border thickness of the box.    |
+-----------------------------------------------------------------+
|G_BUTTON  |The ob_spec field contains a pointer to the text to be |
|          |contained in the button.                              |
+-----------------------------------------------------------------+
|          |The low 16 bits contain a WORD containing color       |
|          |information for the OBJECT. Bits 23-16 contain a signed|
```

G_BOXCHAR	BYTE representing the border thickness of the box. Bits 31-24 contain the ASCII value of the character to display.
G_STRING	The ob_spec field contains a pointer to the text to be displayed.
G_FTEXT	The ob_spec field contains a pointer to a TEDINFO structure.
G_FBOXTEXT	The ob_spec field contains a pointer to a TEDINFO structure.
G_ICON	The ob_spec field contains a pointer to an ICONBLK structure.
G_TITLE	The ob_spec field contains a pointer to the text to be used for the title.
G_CICON	The ob_spec field contains a pointer to a CICONBLK structure.

objc_colorword Almost all objects reference a WORD containing the object color as defined below.

```
objc_colorword=bbbbcccctpppcccc

Bits 15-12 contain the border color
Bits 11-8  contain the text color
Bit   7    is 1 if opaque or 0 if transparent
Bits 6-4   contain the fill pattern index
Bits 3-0   contain the fill color
```

Available colors for fill patterns, text, and borders are listed below:

Value	Name	Color
0	WHITE	White
1	BLACK	Black
2	RED	Red
3	GREEN	Green
4	BLUE	Blue
5	CYAN	Cyan
6	YELLOW	Yellow
7	MAGENTA	Magenta
8	LWHITE	Light Gray
9	LBLACK	Dark Gray
10	LRED	Light Red
11	LGREEN	Light Green
12	LBLUE	Light Blue
13	LCYAN	Light Cyan
14	LYELLOW	Light Yellow
15	LMAGENTA	Light Magenta

TEDINFO G_TEXT, G_BOXTEXT, G_FTEXT, and G_FBOXTEXT objects all reference a TEDINFO structure in their ob_spec field. The TEDINFO structure is defined below:

```
tedinfo$=MKL$(VARPTR(te_ptext$))+MKL$(VARPTR(te_ptmplt$))+
         MKL$(VARPTR(te_pvalid$))+MKI$(te_font)+MKI$(te_fontid)+
         MKI$(te_just)+MKI$(te_color)+MKI$(te_fontsize)+
         MKI$(te_thickness)+MKI$(te_txtlen)+MKI$(te_tmplen)
```

The three character pointer point to text strings required for `G_FTEXT` and `G_FBOXTEXT` objects. te_ptext points to the actual text to be displayed and is the only field used by all text objects. te_ptmplt points to the text template for editable fields. For each character that the user can enter, the text string should contain a tilde character (ASCII 126). Other characters are displayed but cannot be overwritten by the user. `te_pvalid` contains validation characters for each character the user may enter. The current acceptable validation characters are:

Caracter	Allows
9	Digits 0-9
A	Uppercase letters A-Z plus space
a	Upper and lowercase letters plus space
N	Digits 0-9, uppercase letters A-Z and space
n	Digits 0-9, upper and lowercase letters A-Z and space
F	Valid DOS filename characters plus question mark and asterisk
P	Valid DOS pathname characters, backslash, colon, question mark, asterisk
p	Valid DOS pathname characters, backslash and colon
X	All characters

`te_font` may be set to any of the following values:

te_font	Name	Description
3	IBM	Use the standard monospaced font.
5	SMALL	Use the small monospaced font.

`te_just` sets the justification of the text output as follows:

te_just	Name	Description
0	TE_LEFT	Left Justify
1	TE_RIGHT	Right Justify
2	TE_CNTR	Center

te_thickness sets the border thickness (positive and negative values are acceptable) of the G_BOXTEXT or G_FBOXTEXT object.

te_txtlen and te_tmplen should be set to the length of the starting text and template length respectively.

BITBLK G_IMAGE objects contain a pointer to a BITBLK structure in their ob_spec field. The BITBLK structure is defined as follows:

```
bitblk$=MKL$(VARPTR(bi_pdata$))+MKI$(bi_wb)+MKI$(bi_hl)+
        MKI$(bi_x)+MKI$(bi_y)+MKI$(bi_color)
```

`bi_pdata` should contain a monochrome bit image. `bi_wb` specifies the width (in bytes) of the image. All BITBLK images must be a multiple of 16 pixels wide therefore this value must be even. `bi_hl` specifies the height of the image in scan lines (rows). `bi_x` and `bi_y` are used as offsets into `bi_pdata`. Any data occurring before these coordinates will be ignored. `bi_color` is a standard color WORD where the fill color specifies the color in which the image will be rendered.

ICONBLK The `ob_spec` field of `G_ICON` objects point to an ICONBLK structure as defined below:

```
iconblk$=MKL$(VARPTR(ib_pmask$))+MKL$(VARPTR(ib_pdata$))+MKL$(VARPTR(ib_ptext$))+
         MKI$(ib_char)+MKI$(ib_xchar)+MKI$(ib_ychar)+
         MKI$(ib_xicon)+MKI$(ib_yicon)+MKI$(ib_wicon)+MKI$(ib_hicon)+
 MKI$(ib_xtext)+MKI$(ib_ytext)+MKI$(ib_wtext)+MKI$(ib_htext)
```

ib_pmask and ib_pdata contain the monochrome mask and image data respectively. ib_ptext is a string pointer to the icon text. ib_char defines the icon character (used for drive icons) and the icon foreground and background color as follows:

ib_char		
Bits 15-12	Bits 11-8	Bits 7-0
Icon Foreground Color	Icon Background Color	ASCII Character (or 0 for no character).

ib_xchar and ib_ychar specify the location of the icon character relative to ib_xicon and ib_yicon. ib_xicon and ib_yicon specify the location of the icon relative to the ob_x and ob_y of the object. ib_wicon and ib_hicon specify the width and height of the icon in pixels. As with images, icons must be a multiple of 16 pixels in width. ib_xtext and ib_ytext specify the location of the text string relative to the ob_x and ob_y of the object. ib_wtext and ib_htext specify the width and height of the icon text area.

CICONBLK The G_CICON object defines its ob_spec field to be a pointer to a CICONBLK structure as defined below:

```
ciconblk$=monoblk$+MKL$(VARPTR(mainlist$))
```

monoblk contains a monochrome icon which is rendered if a color icon matching the display parameters cannot be found. In addition, the icon text, character, size, and positioning data from the monochrome icon are always used for the color one. mainlist contains the first CICON structure in a linked list of color icons for different resolutions. CICON is defined as follows:

```
cicon$=MKI$(num_planes)+MKL$(VARPTR(col_data$))+MKL$(VARPTR(col_mask$))+
       MKL$(VARPTR(sel_data$))+MKL$(VARPTR(sel_mask$))+
       MKL$(VARPTR(cicon2$))
```

num_planes indicates the number of bit planes this color icon contains. col_data and col_mask contain the icon data and mask for the unselected icon respectively. Likewise, sel_data and sel_mask contain the icon data and mask for the selected icon. cicon2$ contains the next color icon definition. Use MKL$(0) if no more are available.

The GUI library searches the CICONBLK object for a color icon that has the same number of planes in the display. If none is found, the GUI library simply uses the monochrome icon.

APPLBLK `G_PROGDEF` objects allow programmers to define custom objects and link them transparently in the resource. The `ob_spec` field of `G_PROGDEF` objects contains a pointer to an APPLBLK as defined below:

```
applblk$=MKL$(SYM_ADR(#1,"function"))+MKL$(ap_parm)
```

The first is a pointer to a user-defined routine which will draw the object. This routine must be a c-Function, which has to be linked to X11-basic with the LINK command. The routine will be passed a pointer to a PARMBLK structure containing the information it needs to render the object. The routine must be defined with stack checking off and expect to be passed its parameter on the stack. `ap_parm` is a user-defined value which is copied into the PARMBLK structure as defined below:

```
typedef struct parm_blk {
        OBJECT         *tree;
        short           pb_obj;
        short           pb_prevstate;
        short           pb_currstate;
        short           pb_x;
        short           pb_y;
        short           pb_w;
        short           pb_h;
        short           pb_xc;
        short           pb_yc;
        short           pb_wc;
        short           pb_hc;
        long            pb_parm;
} PARMBLK;
```

`tree` points to the OBJECT tree of the object being drawn. The object is located at index `pb_obj`.

The routine is passed the old `ob_state` of the object in `pb_prevstate` and the new `ob_state` of the object in `pb_currstate`. If `pb_prevstate` and `pb_currstate` is equal then the object should be drawn completely, otherwise only the drawing necessary to redraw the object from `pb_prevstate` to `pb_currstate` are necessary.

`pb_x`, `pb_y`, `pb_w`, and `pb_h` give the screen coordinates of the object. `pb_xc`, `pb_yc`, `pb_wc`, and `pb_hc` give the rectangle to clip to. `pb_parm` contains a copy of the `ap_parm` value in the APPLBLK structure. The custom routine should return a short containing any remaining `ob_state` bits you wish the GUI Library to draw over your custom object.

Dialogs

Dialog boxes are modal forms of user input. This means that no other interaction can occur between the user and applications until the requirements of the dialog have been met and it is exited. A normal dialog box consists of an object tree with a BOX as its root object and any number of other controls that accept user input. Both alert boxes and the file selector are examples of dialog boxes.

The `form_do()` function performs the simplest method of using a dialog box. Simply construct an OBJECT tree with at least one EXIT or TOUCHEXIT object and call `form_do()`[1]. All interaction with the dialog like editable fields, radio buttons, and selectable objects will be maintained by the X11-Basic library until the user strikes an EXIT or TOUCHEXIT object.

3.19.4 The gui file format

The *.gui file format, which is basically an ASCII representation of the ATARI ST recource files (*.rsc), can be converted to X11-Basic code, which then can handle message boxes and forms. The converter `gui2bas(1)` does this job. For conversion of ATARI ST recource files to *.gui Files see `rsc2gui(1)`.

The *.gui file consists of Lines and Blocks which specify objects and their hirarchical dependencies. The generic format of such an object is:

```
label: TYPE(variables) {
 ... block ...
}
```

The label is optional and gives the object a name. Depending on TYPE of the object, one or more variables are given as a comma separated list in brackets.

Each object may start a block with '{' at the end of the line. Inside this block there might be one or more objects given which then are considered as sub-objects of the one whichopened the block. The block will be closed by a '}' in a single line.

Example:

```
' Little selector box   (c) Markus Hoffmann    07.2003
' convert this with gui2bas !
' as an example for the use of the gui system
' with X11-Basic


  BOX(X=0,Y=0,W=74,H=14, FRAME=2, FRAMECOL=1, TEXTCOL=1, BGCOL=0, PATTERN=0, TEXTMODE=0, STATE=OUTLINED+
    BOXTEXT(X=2,Y=1,W=70,H=1, TEXT="Select option ...", FONT=3, JUST=2, COLOR=4513, BORDER=253, STATE=SH
    BOX(X=2,Y=3,W=60,H=10, FRAME=-1, FRAMECOL=1, TEXTCOL=1, BGCOL=0, PATTERN=0, TEXTMODE=0) {
      FTEXT(X=1,Y=1,W=30,H=1,COLOR=4513,FONT=3,BORDER=1,TEXT="Line 1", PTMP="__________________________
      FTEXT(X=1,Y=2,W=30,H=1,COLOR=4513,FONT=3,BORDER=1,TEXT="", PTMP="________________________________
      FTEXT(X=1,Y=3,W=30,H=1,COLOR=4513,FONT=3,BORDER=1,TEXT="", PTMP="________________________________
      FTEXT(X=1,Y=4,W=30,H=1,COLOR=4513,FONT=3,BORDER=1,TEXT="", PTMP="________________________________
      BOX(X=2,Y=6,W=50,H=3, FRAME=-1, FRAMECOL=1, TEXTCOL=1, BGCOL=1, PATTERN=5, TEXTMODE=0) {
        BUTTON(X=2,Y=1,W=4,H=1, TEXT="ON",STATE=SELECTED, FLAGS=RADIOBUTTON+SELECTABLE,FRAME=2, FRAMECOL=
        BUTTON(X=8,Y=1,W=4,H=1, TEXT="OFF",FLAGS=RADIOBUTTON+SELECTABLE,FRAME=2, FRAMECOL=1, TEXTCOL=1, 
      }
    }
```

[1] Before you should display the dialog box using the `objc_draw()` function. Maybe you also want to center the dialog with `form_center()` andsave and redraw the background with `form_dial()`.

```
    ok:     BUTTON(X=65,Y=4,W=7,H=4, TEXT="OK", FLAGS=SELECTABLE+DEFAULT+EXIT)
    cancel: BUTTON(X=65,Y=9,W=7,H=4, TEXT="CANCEL", FLAGS=SELECTABLE+EXIT+LASTOB+)
  }
```

3.19.5 Menus

Most applications use a menu bar to allow the user to navigate through program options. In addition, future versions of X11-Basic will allow popup menus and drop-down list boxes (a special form of a popup menu).

Here is a simple example program, which demonstrates the handling of a drop down menu.

```
' Test-program for Drop-Down-Menus
'
DIM field$(50)
FOR i=0 TO 50
  READ field$(i)
  EXIT IF field$(i)="***"
NEXT i
oh=0
field$(i)=""
DATA "INFO","  Menutest"
DATA "---------------"
DATA "- Access.1","- Access.2","- Access.3","- Access.4","- Access.5"
DATA "- Access.6",""
DATA "FILE","  new","  open ...","  save","  save as ...","--------------"
DATA "  print","--------------","  Quit",""
DATA "EDIT","  cut","  copy","  paste","----------","  help1","  helper"
DATA "  assist",""
DATA "HELP","  online help","--------------","  edifac","  editor","  edilink"
DATA "  edouard",""
DATA "***"

grau=get_color(32000,32000,32000)
color grau
pbox 0,0,640,400
MENUDEF field$(),menuaction
DO
  pause 0.05
  MENU
LOOP
quit

PROCEDURE menuaction(k)
  local b
  IF (field$(k)="  Quit") OR (field$(k)="  exit")
    quit
  ELSE IF field$(k)="  online help"
    oh=not oh
    MENUSET k,4*abs(oh)
  ELSE IF field$(k)="  Menutest"
    ~form_alert(1,"[0][---- Menutest ----||(c) Markus Hoffmann 2001|X11-Basic V.1.03][ OK ]")
  ELSE
    PRINT "MENU selected ";k;" contents: ";field$(k)
    b=form_alert(1,"[1][--- Menutest ---||You selected item (No. "+str$(k)+"),| for which was n
    if b=2
      MENUSET k,8
    endif
  ENDIF
RETURN
```

3.20 WEB programming with X11-Basic

This chapter explaines, how you can use X11-Basic programs for WEB-interfacing. Especially via the use of so-called **CGI-Scripts**.

3.20.1 What is CGI?

CGI stands for *Common Gateway Interface* — a term you don't really need to know. In short, CGI defines how web servers and web browsers handle information from HTML forms on web pages. This means instead of the WEB server sending static web pages to the clients, it can invoke a program, typically called a cgi-script, to generate the page on the time the request was received. These cgi"-scripts take some action, and then send a results page back to the user's web browser. The results page might be different every time the program is run.

And these programs can be X11-Basic programs.

3.20.2 Configuration

1. **All X11-Basic scripts must begin** with the following statement, on the first line:

   ```
   #!/usr/bin/xbasic
   ```

   Because Unix does not map file suffixes to programs, there has to be a way to tell Unix that this file is a X11-Basic program, and that it is to be executed by the X11-Basic interpreter xbasic. This is seen before in shell scripts, in which the first line tells Unix to execute it with one of the shell programs. The xbasic executable, which will take this file, parse it, and execute it, is located in the directory /usr/bin. This may be different on some systems. If you are not sure where the xbasic executable is, type which xbasic on the command line, and it will return you the path.

2. **All scripts should be marked as executable by the system.**

   Executable files are types that contain instructions for the machine or an interpreter, such as xbasic, to execute. To mark a file as executable, you need to alter the file permissions on the script file. There are three basic permissions: read, write, and execute. There are also three levels of access: owner, group, and anyone. X11-Basic files should have their permissions changed so that you, the owner, has permission to read, write and execute your file, while others only have permission to read and execute your file. This is done with the following command:

   ```
   chmod 755 filename.bas
   ```

   The number 755 is the file access mask. The first digit is your permission; it is 7 for full access. The user and anyone settings are 5 for read and execute.

3. **The very first print statement** in a X11-Basic cgi script that returns HTML should be:

   ```
   print "Content-type: text/html"+chr$(13)
   print ""+chr$(13)
   flush
   ```

   When your X11-Basic script is going to return an HTML file, you must have this as the very first print statement in order to tell the web server that this is an HTML file. There must be two end of line characters (CR+LF) (the aditional `chr$(13)`) in order for this to work. The flush statement ensures, that this statement ist sent to the web-server. After that, you usually `print "<HTML><BODY>"` etc.

4. **End your program with** `quit`

   Do not use `END`. Otherwise the cgi-program will remain is the servers memory as a zombie.

5. **Always use the POST method with HTML forms**

   There are 2 ways to get information from the client to the web server. The GET method takes all of the data from the forms and concatenates it onto the end of the URL. This information is then passed to the CGI program as an environment variable (`QUERY_STRING`). Because the GET method has the limitation of being 1024 characters long, it is best to use the POST method. This takes the data and sends it allong with the request to the web server, without the user seeing the ugly strings in the URL. This information is passed to the CGI program through standard in, which the program can easilly read from. To use the POST method, make sure that your HTML form tag has `METHOD=POST` (no quotes).

6. **HTML forms must reference the cgi script to be executed.**

   In your FORM tag, there is an ACTION attribute. This is like the HREF attribute for a link. It should be the URL of the CGI program you want the form data sent to. Usually this is `ACTION="/cgi-bin/filename.bas"`

7. **X11-Basic-cgi files usually go in the cgi-bin directory of your web server.**

   The web server has a "root" directory. This is the highest directory your HTML files can access. (You don't want clients to be able to snoop around your entire system, so the rest of the system is sealed off) in this directory, there is usually one called cgi-bin, where all the CGI programs go. Some web service providers give each user a cgi-local directory in their home directory where they can put their cgi scripts. If this is the case, use this one instead.

3.20.3 How it works

When a user activates a link to a gateway script, input is sent to the server. The server formats this data into environment variables and checks to see whether additional data was submitted via the standard input stream.

Environment Variables

Input to CGI scripts is usually in the form of environment variables. The environment variables passed to gateway scripts are associated with the browser requesting information from the server, the server processing the request, and the data passed in the request. Environment variables are case-sensitive and are normally used as described in this section. Although some environment variables are system-specific, many environment variables are standard. The standard variables are shown in the following Table:

```
                 Standard environment variables.

+--------------------------------------------------------------+
| Variable          | Purpose                                  |
|-------------------+------------------------------------------|
| AUTH_TYPE         | Specifies the authentication method and  |
|                   | is used to validate a user's access.     |
|-------------------+------------------------------------------|
| CONTENT_LENGTH    | Used to provide a way of tracking the    |
|                   | length of the data string as a numeric   |
|                   | value.                                   |
|-------------------+------------------------------------------|
| CONTENT_TYPE      | Indicates the MIME type of data.         |
|-------------------+------------------------------------------|
| GATEWAY_INTERFACE | Indicates which version of the CGI       |
|                   | standard the server is using.            |
|-------------------+------------------------------------------|
| HTTP_AccEPT       | Indicates the MIME content types the     |
|                   | browser will accept, as passed to the    |
|                   | gateway script via the server.           |
|-------------------+------------------------------------------|
| HTTP_USER_AGENT   | Indicates the type of browser used to    |
|                   | send the request, as passed to the       |
|                   | gateway script via the server.           |
|-------------------+------------------------------------------|
| PATH_INFO         | Identifies the extra information         |
|                   | included in the URL after the            |
|                   | identification of the CGI script.        |
|-------------------+------------------------------------------|
| PATH_TRANSLATED   | Set by the server based on the PATH_INFO |
|                   | variable. The server translates the      |
|                   | PATH_INFO variable into this variable.   |
|-------------------+------------------------------------------|
| QUERY_STRING      | Set to the query string (if the URL      |
|                   | contains a query string).                |
|-------------------+------------------------------------------|
| REMOTE_ADDR       | Identifies the Internet Protocol address |
|                   | of the remote computer making the        |
|                   | request.                                 |
|-------------------+------------------------------------------|
| REMOTE_HOST       | Identifies the name of the machine       |
|                   | making the request.                      |
|-------------------+------------------------------------------|
| REMOTE_IDENT      | Identifies the machine making the        |
|                   | request.                                 |
|-------------------+------------------------------------------|
| REMOTE_USER       | Identifies the user name as              |
|                   | authenticated by the user.               |
|-------------------+------------------------------------------|
| REQUEST_METHOD    | Indicates the method by which the        |
|                   | request was made.                        |
|-------------------+------------------------------------------|
| SCRIPT_NAME       | Identifies the virtual path to the       |
|                   | script being executed.                   |
|-------------------+------------------------------------------|
| SERVER_NAME       | Identifies the server by its host name,  |
|                   | alias, or IP address.                    |
|-------------------+------------------------------------------|
| SERVER_PORT       | Identifies the port number the server    |
|                   | received the request on.                 |
|-------------------+------------------------------------------|
| SERVER_PROTOCOL   | Indicates the protocol of the request    |
|                   | sent to the server.                      |
|-------------------+------------------------------------------|
| SERVER_SOFTWARE   | Identifies the Web server software.      |
+--------------------------------------------------------------+
```

AUTH_TYPE The AUTH_TYPE variable provides access control to protected areas of the Web server and can be used only on servers that support user authentication. If an area of the Web site has no access control, the AUTH_TYPE variable has no value associated with it. If an area of the Web site has access control, the AUTH_TYPE variable is set to a specific value that identifies the authentication scheme being used.

Using this mechanism, the server can challenge a client's request and the client can respond. To do this, the server sets a value for the AUTH_TYPE variable and the client supplies a matching value. The next step is to authenticate the user. Using the basic authentication scheme, the user's browser must supply authentication information that uniquely identifies the user. This information includes a user ID and password.

Under the current implementation of HTTP, HTTP 1.0, the basic authentication scheme is the most commonly used authentication method. To specify this method, set the `AUTH_TYPE` variable as follows: `AUTH_TYPE = Basic`

CONTENT_LENGTH The `CONTENT_LENGTH` variable provides a way of tracking the length of the data string. This variable tells the client and server how much data to read on the standard input stream. The value of the variable corresponds to the number of characters in the data passed with the request. If no data is being passed, the variable has no value.

CONTENT_TYPE The `CONTENT_TYPE` variable indicates the data's MIME type. This variable is set only when attached data is passed using the standard input or output stream. The value assigned to the variable identifies the MIME type and subtype as follows:

Basic MIME types.

Type	Description
application	Binary data that can be executed or used with another application
audio	A sound file that requires an output device to preview
image	A picture that requires an output device to preview
message	An encapsulated mail message
multipart	Data consisting of multiple parts and possibly many data types
text	Textual data that can be represented in any character set or formatting language
video	A video file that requires an output device to preview
x-world	Experimental data type for world files

MIME subtypes are defined in three categories: primary, additionally defined, and extended. The primary subtype is the primary type of data adopted for use as a MIME content type. Additionally defined data types are additional subtypes that have been officially adopted as MIME content types. Extended data types are experimental subtypes that have not been officially adopted as MIME content types. You can easily identify extended subtypes because they begin with the letter x followed by a hyphen. The following Table lists common MIME types and their descriptions.

Common MIME types.

Type/Subtype	Description
application/msword	Microsoft Word document
application/octet-stream	Binary data that can be executed or used with another application
application/pdf	ACROBAT PDF document
application/postscript	Postscript-formatted data
application/rtf	Rich Text Format (RTF) document
application/x-compress	Data that has been compressed using UNIX compress
application/x-dvi	Device-independent file
application/x-gzip	Data that has been compressed using UNIX gzip
application/x-latex	LATEX document
application/x-tar	Data that has been archived using UNIX tar
audio/basic	Audio in a nondescript format
audio/x-wav	Audio in Microsoft WAV format
image/gif	Image in gif format
image/jpeg	Image in JPEG format
image/tiff	Image in TIFF format
image/x-portable-bitmap	Portable bitmap
image/x-portable-graymap	Portable graymap
image/x-portable-pixmap	Portable pixmap
image/x-xbitmap	X-bitmap
image/x-xpixmap	X-pixmap

```
|------------------------------+-------------------------------|
| message/external-body        | Message with external data    |
|                              | source                        |
|------------------------------+-------------------------------|
| message/partial              | Fragmented or partial message |
|------------------------------+-------------------------------|
| message/rfc822               | RFC 822-compliant message     |
|------------------------------+-------------------------------|
| multipart/alternative        | Data with alternative formats |
|------------------------------+-------------------------------|
| multipart/digest             | Multipart message digest      |
|------------------------------+-------------------------------|
| multipart/mixed              | Multipart message with data   |
|                              | in multiple formats           |
|------------------------------+-------------------------------|
| multipart/parallel           | Multipart data with parts     |
|                              | that should be viewed         |
|                              | simultaneously                |
|------------------------------+-------------------------------|
| text/html                    | HTML-formatted text           |
|------------------------------+-------------------------------|
| text/plain                   | Plain text with no HTML       |
|                              | formatting included           |
|------------------------------+-------------------------------|
| video/mpeg                   | Video in the MPEG format      |
|------------------------------+-------------------------------|
| video/quicktime              | Video in the Apple QuickTime  |
|                              | format                        |
|------------------------------+-------------------------------|
| video/x-msvideo              | Video in the Microsoft AVI    |
|                              | format                        |
|------------------------------+-------------------------------|
| x-world/x-vrml               | VRML world file               |
+--------------------------------------------------------------+
```

There are some more common MIME types.

Some MIME content types can be used with additional parameters. These content types include text/plain, text/html, and all multipart message data. The charset parameter, which is optional, is used with the text/plain type to identify the character set used for the data. If a charset is not specified, the default value charset=us-ascii is assumed. Other values for charset include any character set approved by the International Standards Organization. These character sets are defined by ISO-8859-1 to ISO-8859-9 and are specified as follows:

```
CONTENT_TYPE = text/plain; charset=iso-8859-1
```

The boundary parameter, which is required, is used with multipart data to identify the boundary string that separates message parts. The boundary value is set to a string of

1 to 70 characters. Although the string cannot end in a space, it can contain any valid letter or number and can include spaces and a limited set of special characters. Boundary parameters are unique strings that are defined as follows:

```
CONTENT_TYPE = multipart/mixed; boundary=boundary_string
```

GATEWAY_INTERFACE The `GATEWAY_INTERFACE` variable indicates which version of the CGI specification the server is using. The value assigned to the variable identifies the name and version of the specification used as follows:

```
GATEWAY_INTERFACE = name/version
```

The current version of the CGI specification is 1.1. A server conforming to this version would set the `GATEWAY_INTERFACE` variable as follows:

```
GATEWAY_INTERFACE = CGI/1.1
```

HTTP_ACCEPT The `HTTP_ACCEPT` variable defines the types of data the client will accept. The acceptable values are expressed as a type/subtype pair. Each type/subtype pair is separated by commas.

HTTP_USER_AGENT The `HTTP_USER_AGENT` variable identifies the type of browser used to send the request. The acceptable values are expressed as software type/version or library/version.

PATH_INFO The `PATH_INFO` variable specifies extra path information and can be used to send additional information to a gateway script. The extra path information follows the URL to the gateway script referenced. Generally, this information is a virtual or relative path to a resource that the server must interpret.

PATH_TRANSLATED Servers translate the `PATH_INFO` variable into the `PATH_TRANSLATED` variable by inserting the default Web document's directory path in front of the extra path information.

QUERY_STRING The `QUERY_STRING` variable specifies an URL-encoded search string. You set this variable when you use the GET method to submit a fill-out form or when you use an ISINDEX query to search a document. The query string is separated from the URL by a question mark. The user submits all the information following the question mark separating the URL from the query string. The following is an example:

```
/cgi-bin/doit.cgi?string
```

When the query string is URL-encoded, the browser encodes key parts of the string. The plus sign is a placeholder between words and acts as a substitute for spaces:

```
/cgi-bin/doit.cgi?word1+word2+word3
```

Equal signs separate keys assigned by the publisher from values entered by the user. In the following example, response is the key assigned by the publisher, and never is the value entered by the user:

```
/cgi-bin/doit.cgi?response=never
```

Ampersand symbols separate sets of keys and values. In the following example, response is the first key assigned by the publisher, and sometimes is the value entered by the user. The second key assigned by the publisher is reason, and the value entered by the user is I am not really sure:

```
/cgi-bin/doit.cgi?response=sometimes&reason=I+am+not+really+sure
```

Finally, the percent sign is used to identify escape characters. Following the percent sign is an escape code for a special character expressed as a hexadecimal value. Here is how the previous query string could be rewritten using the escape code for an apostrophe:

```
/cgi-bin/doit.cgi?response=sometimes&reason=I%27m+not+really+sure
```

REMOTE_ADDR The `REMOTE_ADDR` variable is set to the Internet Protocol (IP) address of the remote computer making the request.

REMOTE_HOST The `REMOTE_HOST` variable specifies the name of the host computer making a request. This variable is set only if the server can figure out this information using a reverse lookup procedure.

REMOTE_IDENT The `REMOTE_IDENT` variable identifies the remote user making a request. The variable is set only if the server and the remote machine making the request support the identification protocol. Further, information on the remote user is not always available, so you should not rely on it even when it is available.

REMOTE_USER The `REMOTE_USER` variable is the user name as authenticated by the user, and as such is the only variable you should rely upon to identify a user. As with other types of user authentication, this variable is set only if the server supports user authentication and if the gateway script is protected.

REQUEST_METHOD The REQUEST_METHOD variable specifies the method by which the request was made. The methods could be any of GET, HEAD, POST, PUT, DELETE, LINK and UNLINK.

The GET, HEAD and POST methods are the most commonly used request methods. Both GET and POST are used to submit forms.

SCRIPT_NAME The SCRIPT_NAME variable specifies the virtual path to the script being executed. This information is useful if the script generates an HTML document that references the script.

SERVER_NAME The SERVER_NAME variable identifies the server by its host name, alias, or IP address. This variable is always set.

SERVER_PORT The SERVER_PORT variable specifies the port number on which the server received the request. This information can be interpreted from the URL to the script if necessary. However, most servers use the default port of 80 for HTTP requests.

SERVER_PROTOCOL The SERVER_PROTOCOL variable identifies the protocol used to send the request. The value assigned to the variable identifies the name and version of the protocol used. The format is name/version, such as HTTP/1.0.

SERVER_SOFTWARE The SERVER_SOFTWARE variable identifies the name and version of the server software. The format for values assigned to the variable is name/version, such as CERN/2.17.

CGI Standard Input

Most input sent to a Web server is used to set environment variables, yet not all input fits neatly into an environment variable. When a user submits data to be processed by a gateway script, this data is received as an URL-encoded search string or through the standard input stream. The server knows how to process this data because of the method (either POST or GET in HTTP 1.0) used to submit the data.

Sending data as standard input is the most direct way to send data. The server tells the gateway script how many eight-bit sets of data to read from standard input. The script opens the standard input stream and reads the specified amount of data. Although long URL-encoded search strings may get truncated, data sent on the standard input stream will not. Consequently, the standard input stream is the preferred way to pass data.

Which CGI Input Method to use?

You can identify a submission method when you create your fill-out forms. Under HTTP 1.0, two submission methods for forms exist. The HTTP GET method uses URL-encoded search strings. When a server receives an URL-encoded search string, the server assigns the value of the search string to the QUERY_STRING variable.

The HTTP POST method uses the standard input streams. When a server receives data by the standard input stream, the server assigns the value associated with the length of the input stream to the `CONTENT_LENGTH` variable.

Output from CGI Scripts

After the script finishes processing the input, the script should return output to the server. The server will then return the output to the client. Generally, this output is in the form of an HTTP response that includes a header followed by a blank line and a body. Although the CGI header output is strictly formatted, the body of the output is formatted in the manner you specify in the header. For example, the body can contain an HTML document for the client to display.

CGI Headers

CGI headers contain directives to the server. Currently, these three server directives are valid:

```
    * Content-Type
    * Location
    * Status
```

A single header can contain one or all of the server directives. Your CGI script outputs these directives to the server. Although the header is followed by a blank line that separates the header from the body, the output does not have to contain a body.

The **Content-Type** field in a CGI header identifies the MIME type of the data you are sending back to the client. Usually the data output from a script is a fully formatted document, such as an HTML document. You could specify this output in the header as follows:

```
Content-Type: text/html
```

But of course, if your program outputs other data like images etc. you should specify the corresponding content type.

Locations The output of your script doesn't have to be a document created within the script. You can reference any document on the Web using the Location field. The Location field references a file by its URL. Servers process location references either directly or indirectly depending on the location of the file. If the server can find the file locally, it passes the file to the client. Otherwise, the server redirects the URL to the client and the client has to retrieve the file. You can specify a location in a script as follows:

```
Location: http://www.new-jokes.com/
```

Status The Status field passes a status line to the server for forwarding to the client. Status codes are expressed as a three-digit code followed by a string that generally explains what has occurred. The first digit of a status code shows the general status as follows:

```
  1XX Not yet allocated
  2XX Success
  3XX Redirection
  4XX Client error
  5XX Server error
```

Although many status codes are used by servers, the status codes you pass to a client via your CGI script are usually client error codes. Suppose the script could not find a file and you have specified that in such cases, instead of returning nothing, the script should output an error code. Here is a list of the client error codes you may want to use:

```
401 Unauthorized Authentication has failed.
    User is not allowed to access the file and should try again.

403 Forbidden. The request is not acceptable.
    User is not permitted to access file.

404 Not found. The specified resource could not be found.

405 Method not allowed. The submission method used is not allowed.
```

3.20.4 Example cgi-Script `envtest.cgi`

Here is a simple sample cgi-script, which simply returns all the information which it gets from the web server as a html page.

```
#!/usr/bin/xbasic
print "Content-type: text/html"+chr$(13)
print ""+chr$(13)
flush
print "<html><head><TITLE>Test CGI</TITLE><head><body>"
print "<h1>Commandline:</h1>"
i=0
while len(param$(i))
  print str$(i)+": "+param$(i)+"<br>"
  inc i
wend
print "<h1>Environment:</h1><pre>"
flush      ! flush the output before another program is executed !
system "env"
print "</pre><h1>Stdin:</h1><pre>"
length=val(env$("CONTENT_LENGTH"))
if length
  for i=0 to length-1
    t$=t$+chr$(inp(-2))
  next i
  print t$
```

```
endif
print "</pre>"
print "<FORM METHOD=POST ACTION=/cgi-bin/envtest.cgi>"
print "Name: <INPUT NAME=name><BR>"
print "Email: <INPUT NAME=email><BR>"
print "<INPUT TYPE=submit VALUE="+chr$(34)+"Test POST Method"+chr$(34)+">"
print "</FORM>"
print "<hr><h6>(c) Markus Hoffmann cgi with X11-basic</h6></body></html>"
flush
quit
```

4 Quick reference

4.1 reserved variable names

There are some reserved variables. Some Keywords may not work as varable names as well. Although there is no checking done, parsing errors could occure. Please try the command LET in such cases. In general, as long as an ending of a variable name is different from any command or keyword, it's usable as name.

Reserved and system variables are:

`TRUE`	-1
`FALSE`	0
`PI`	3.14159265359...
`TIMER`	unix system timer, float, seconts
`STIMER`	integer system timer
`CTIMER`	system timer in units of CPU-time
`PC`	line number of next line to be processed
`SP`	internal stack pointer
`ROWS`	number of rows of the terminal
`COLS`	number of columns of the terminal
`ERR`	error number of last error
`MOUSEX`	x coordinate of mouse position relative to window
`MOUSEY`	y coordinate of mouse position
`MOUSEK`	mouse button state
`MOUSES`	state of the shift,alt,ctrl,caps keys
`INKEY$`	content of the keyboard-buffer
`TERMINALNAME$`	device name of the standard terminal
`TIME$`	current time
`DATE$`	current date

4.2 Abbreviations

In direct mode in the interpreter every command can be abbreviated as long as the command parser can identify uniquely the command. So you may use `q` instead of `QUIT`.

In addition there are abbreviations which are actually alternate commands like:

'	REM
?	PRINT
@	GOSUB
~	VOID
&	EVAL

4.3 Commands

`AFTER n,procedure`	execute procedure after n seconds
`ARRAYFILL a(),b`	fills array with value
`ARRAYCOPY dest(),souce()`	copies array including Dimensionierung
`BEEP`	Beep (on TTY/console)
`BELL`	same as BEEP
`BGET #f,a,n`	read n bytes from file #f to adress a
`BLOAD f$,a[,l]`	reads entire file (given by name) to adress a
`BMOVE q,z,n`	copies a block of n bytes from adress q to z
`BPUT #f,a,n`	writes n bytes from adress a to file/channel f
`BREAK`	same as EXIT IF true
`BSAVE f$,a,l`	saves l bytes in memory at adress a to file f$
`CALL adr[,par,...]`	see EXEC
`CASE const`	see SELECT * CASE * DEFAULT * ENDSELECT
`CHAIN bas$`	executes another basic program
`CLEAR`	clear and remove all variables
`CLOSE [[#]n]`	close file, I/O channel or link
`CLR a,b,c(),f$`	clear variables
`CLS`	clear (text)screen
`CONT`	continue (after STOP)
`DATA 1,"Hallo",...`	define constants
`DEFAULT`	see SELECT * CASE * DEFAULT * ENDSELECT
`DIM`	Declarate array
`DO * LOOP`	Loop
`DPOKE adr,word`	write short int word to adr
`DUMP`	lists all used variable names
`DUMP "@"`	list of Funktionen und Prozeduren
`DUMP ":"`	list of all labels
`DUMP "#"`	list of open Files
`DUMP "K"`	list of implementierten Kommandos
`DUMP "F"`	list of internal functions
`ECHO ON/OFF`	same as TRON * TROFF
`EDIT`	call default editor to edit program

ELSE	see IF * ELSE * ENDIF
END	program end, enter interactive mode
ENDFUNCTION	see FUNCTION * ENDFUNCTION
ENDIF	see IF * ELSE * ENDIF
ENDSELECT	see SELECT * CASE * DEFAULT * ENDSELECT
ERASE a()[,b$(),...]	erase arrays
ERROR n	execute error number n
EVAL t$	execute X11-Basic command contained in t$
EVERY n,procedure	invokes procedure every n seconds
EXEC adr[,var[,...]]	call a C subroutine at pointer adr.
EXIT IF a	exit loop if condition a is TRUE
FLUSH [#n]	flush output
FOR * NEXT	For Next loop
FORM_INPUT t$	input string with default value
FUNCTION * ENDFUNC	define function
GOSUB procedure(varliste)	call subroutine
GOTO label	goto label
HELP <expr>	prints short help on expr
HOME	Textcursor home
IF * ELSE IF * ELSE * ENDIF	conditions
INC a	increments variable a
INPUT [#n,]["text";] varlist	read values for variables
LET a=b	enforces assignment
LINEINPUT [#n,]t$	read entire line from channel/file/console
LINK #n,t$	load shared object file t$
LIST [s,e]	List programm code (from line s to e)
LOAD a$	load Program
LOCAL var[,var2,...]	specifies a list of vars to be local in Procedure or function
LOCATE column,row	Place cursor on column and row
LOOP	see DO * LOOP
LPOKE adr,long	writes long int value to pointer adr
LSET t$=a$	
MERGE f$	Merges bas-file to actual program code
MUL a,b	same as a=a*b
NEW	clear and erase all variables and stop.
NEXT	see FOR * NEXT
NOP	no operation do nothing
NOOP	no operation do nothing
ON * GOSUB proc1[,proc2,...]	
ON * GOTO label1[,label2,...]	
ON BREAK GOSUB proc	
ON ERROR GOSUB proc	

`OPEN mode$,#n,filename$`	open a file or socket for input and/or output
`OUT #n,a`	out byte a to channel n
`PAUSE sec`	pauses sec seconds
`PLIST`	formatted listing
`POKE adr,byte`	write byte to pointer adr
`PRINT a;b$`	console output
`PRINT #n;`	output to channel/file
`PRINT AT(x,y);`	locate textcursor at row y and column x
`PRINT a USING f$`	print number with formatter
`PROCEDURE procname [(p1 [,p2] ... )] * RETURN`	
`PSAVE a$`	writes the reformatted BASIC-program into file a$
`PUTBACK [#n,]a`	put back a char to channel/file/console
`QUIT`	quits the X11-BASIC-Interpreter
`RANDOMIZE [seed]`	Sets seed for random generator, default is TIMER
`READ var`	reads constant from DATA statement
`RELSEEK #n,d`	Place filepointer on new relative position d
`REM comment`	comment
`REPEAT`	see REPEAT * UNTIL
`RESTORE [label]`	(re)sets pointer for READ-statement to label
`RESUME`	
`RETURN`	define the end of a PROCEDURE
`RETURN expr`	return value from FUNCTION
`RSRC_LOAD filename$`	loads GEM rsc-File (ATARI ST)
`RSRC_FREE`	frees GEM rsc-File (ATARI ST)
`RUN`	start program
`SAVE [a$]`	writes the BASIC-program into file with the name a$
`SEEK #n,d`	Place filepointer on new absolute position d
`SETENV t$=a$`	Sets environmentvar t$ using value a$
`SORT a(),n[,b()]`	Sort array
`SOUND freq`	Sound the internal speaker
`SWAP`	
`SYSTEM t$`	excecute shell with command t$
`TROFF`	Trace mode off
`TRON`	Trace mode on (for debugging)
`UNLINK #n`	unlinks shared object #n
`UNTIL exp`	if exp is false goto REPEAT
`VERSION`	shows X11-Basic version number and date
`VOID a`	claculates expresion a and discard result
`WORT_SEP t$,d$,mode,a$,b$`	separates string t$ by deliminator d$ in a$ and b$

4.4 Graphic commands

ALERT a,b$,c,d$,var[,ret$]	Infobox
BOX x1,y1,x2,y2	draw a frame
CIRCLE x,y,r	draw a circle
CLEARW [[#]n]	clear graphic window
CLOSEW [[#]n]	close graphic window
COLOR f[,b]	Set foreground color (and background color)
DEFFILL c,a,b	set fill style and pattern
DEFLINE a,b	set line width and type
DEFMARK c,a,g	set colour, size, type (POLYMARK)
DEFMOUSE i	set mouse cursor type
DEFTEXT c,s,r,g	set text properties for ltext
DRAW [[x1,y1] TO] x2,y2	draw line
ELLIPSE x,y,a,b[,a1,a2]	draw an ellipse
FILESELECT tit$,pfad$,default$,f$	display a fileselector-box
GET x,y,w,h,g$	Grafik-Ausschnitt in g$ speichern
GPRINT	like PRINT, but the output goes to the graphic window
GRAPHMODE mode	set graphic-mode
KEYEVENT a,b	Waits until key is pressed
LINE x1,y1,x2,y2	draw a line
LTEXT x,y,t$	Linegraphic-Text
MENUDEF array$(),proc	read text for menu-header from array$()
MENUKILL	deletes menue
MENUSET n,x	change menu-point #n with value x
MENU STOP	switch off the menu
ONMENU	execute the menu and
MENU	wait for menue-events
MOUSE x,y,k	gets position and state of mouse
MOUSEEVENT	wait for mouse event
MOTIONEVENT	wait for mouse movement
MOVEW n,x,y	move window
OPENW n	open window
PBOX x1,y1,x2,y2	draw filled box
PCIRCLE x,y,r[,a1,a2]	draw filled cirle
PELLIPSE x,y,a,b[,a1,a2]	draw filled ellipse
PLOT x,y	draw point
POLYLINE n,x(),y()	draw polygon in (x(),y())
POLYFILL n,x(),y()	draw filled polygon
POLYMARK n,x(),y()	draw polygon points
PRBOX x1,y1,x2,y2	draw filled rounded box

`PUT x,y,g$`	map graphic at position
`PUT_BITMAP t$,i,i,i,i`	map bitmap
`RBOX x1,y1,x2,y2`	draws a rounded box
`SCOPE a(),typ,ys,yo`	fast plot a()
`SCOPE y(),x(),typ,ys,yo,xs,xo`	fast 2D plot
`SGET screen$`	capture graphic and store it in screen$
`SHOWPAGE`	maps all graphic to window
`SPUT screen$`	maps (xwd-)graphic to window
`TEXT x,y,t$`	draw text
`TITLEW n,t$`	set window title
`VSYNC`	same as SHOWPAGE
`XLOAD`	load a program, with FILESELECTOR
`XRUN`	load and run a program, with FILESELECTOR

4.5 Math commands

`ADD a,b`	same as a=a+b but faster
`DEC var`	same as var=var-1 but faster
`DIV a,b`	same as a=a/b but faster
`FFT a(),i`	fast fourier transformation on 1D array.
`FIT x(),y()[,yerr()],n,func(x,a,b,c,...)`	fits function to data
`FIT_LINEAR x(),y()[,[xerr(),]yerr()],n,a,b[,siga,sigb,chi2,q]`	linear regression with errors
`INC var`	same as var=var+1 but faster
`MUL a,b`	same as a=a*b but faster
`SORT a(),n[,b()]`	sorts n values of a() to incrementing order
`SUB a,b`	same as a=a-b but faster

4.6 Math functions

The math function library contains a comprehensive set of mathematics functions, including:

- trigonometric
- arc-trigonometric
- hyperbolic

- arc-hyperbolic
- logarithmic (base e and base 10)
- exponential (base e and base 10)
- miscellaneous (square root, power, etc.)

Some math functions are defined on Vectors and Matrices.

`b=ABS(a)`	absolut value
`c=ADD(a,b)`	add
`a=CINT(b)`	Truncate number (NOTE: differs from INT() !)
`a=RND(dummy)`	random number between 0 and 1
`a=GASDEV(dummy)`	random number Gauss distribution
`a=RAND(dummy)`	random integer number between 0 and a large number
`a=RANDOM(n)`	random integer number between 0 and n
`i=SGN(a)`	sign of a (-1,0,1)
`b=SQR(a)`	square root
`b=SQRT(a)`	square root
`b=TRUNC(a)`	round a to the nearest integer not larger in absolute value
`b=FRAC(a)`	fractional (non integer) part of a
`b=INT(a)`	convert to integer
`b=LN(a)`	base e logarithm (natural log)
`b=LOG(a)`	base e logarithm (natural log)
`b=LOG10(a)`	base 10 logarithm
`b=EXP(a)`	base e "anti-log" (e to the x)
`b=FAK(a)`	Fakultaet

4.6.1 Angles

Angles are always radians, for both, arguments and return values.

`b=RAD(a)`	convert degrees to radians
`b=DEG(a)`	convert radians to degrees

4.6.2 Trigonometric functions

`b=SIN(a)`	sine
`b=COS(a)`	cosine
`b=TAN(a)`	tangent
`b=COT(a)`	cotangent
`b=SEC(a)`	secant
`b=CSC(a)`	cosecant
`b=ASIN(a)`	arc-sine
`b=ACOS(a)`	arc-cosine
`b=ATAN(a)`	arc-tangent
`b=ATAN2(a,c)`	extended arc-tangent
`b=ACOT(a)`	arc-cotangent
`b=ASEC(a)`	arc-secant
`b=ACSC(a)`	arc-cosecant
`b=SINH(a)`	hyperbolic sine
`b=COSH(a)`	hyperbolic cosine
`b=TANH(a)`	hyperbolic tangent
`b=COTH(a)`	hyperbolic cotangent
`b=SECH(a)`	hyperbolic secant
`b=CSCH(a)`	hyperbolic cosecant
`b=ASINH(a)`	hyperbolic arc-sine
`b=ACOSH(a)`	hyperbolic arc-cosine
`b=ATANH(a)`	hyperbolic arc-tangent
`b=ACOTH(a)`	hyperbolic arc-cotangent
`b=ASECH(a)`	hyperbolic arc-secant
`b=ACSCH(a)`	hyperbolic arc-cosecant

4.6.3 Boolean functions

`a=EVEN(d)`	TRUE if d is even number
`a=ODD(d)`	TRUE if d is odd number

4.7 String functions

`b$=BIN$(a[,n])`	convert to binary number
`t$=CHR$(a)`	convert ascii code to string

`t$=ENV$(n$)`	read value of environment variable n$
`t$=HEX$(a[,n])`	a as Hexadecimal number
`t$=INLINE$(a$)`	7Bit-ASCII to Binary conversion, can be used to include Binary data in
`t$=INPUT$(#n,num)`	reads num bytes from file/channel n
`d$=juldate$(a)`	date$ by julian day a
`u$=LCASE$(t$)`	converts t$ to lower case
`t$=LEFT$(a$[,n])`	extraxts from string a$ the first (left) n characters
`u$=LOWER$(t$)`	converts t$ to lower case
`m$=MID$(t$,s[,l])`	extraxts from string t$ a string from position s with l characters
`t$=MKI$(i)`	convert Integer to 2-Byte String
`t$=MKL$(i)`	convert integer to 4-Byte String
`t$=MKF$(a)`	convert float to 4 Byte String
`t$=MKD$(a)`	convert float to 8 Byte String
`o$=OCT$(d,n)`	convert integer d to string with octal number
`t$=PRG$(i)`	Program line
`t$=RIGHT$(a$[,n])`	returns right n characters of a$
`t$=SPACE$(i)`	returns string consisting of i spaces
`t$=STR$(a[,b,c])`	convert number to String of length b with c signifikant digits
`t$=STRING$(w$,i)`	returns string consisting of i copys of w$
`t$=SYSTEM$(n$)`	execute shell with command n$
`t$=TERMINALNAME$(#n)`	returns device name of terminal connected to #n
`u$=UCASE$(t$)`	converts t$ to upper case
`t$=unixtime$(i)`	give time$ from TIMER value
`d$=unixdate$(i)`	give date$ from TIMER value
`u$=UPPER$(t$)`	converts t$ to upper case

4.8 Graphic functions

`a=FORM_ALERT(n,t$)`	message box with default button n
`a=FORM_DIAL(i,i,i,i,i,i,i,i,i)`	complex function
`a=FORM_DO(i)`	do dialog
`c=GET_COLOR(r,g,b)`	allocate color by rgb value
`dummy=OBJC_DRAW(i,i,i,i,i)`	draw object tree
`ob=OBJC_FIND(tree,x,y)`	return object number by coordinates
`c=POINT(x,y)`	returns color of pixel of graphic in window
`a=RSRC_GADDR(typ,nr)`	get pointer to object tree

4.9 Other functions

`a=ARRPTR(b())`	pointer to array descriptors
`a=ASC(t$)`	ASCII code of first letter of string
`b=CVI(a$)`	convert 2-byte string to integer
`b=CVL(a$)`	convert 4-byte string to integer
`b=CVS(a$)`	convert 4-byte string to float
`b=CVF(a$)`	convert 4-byte string to float
`b=CVD(a$)`	convert 8-byte string to double
`a=DIM?(a())`	returns number of elements of array a()
`i=DPEEK(adr)`	read word from pointer adr
`b=EOF(#n)`	TRUE if file pointer reached end of file
`a=EVAL(t$)`	evaluate expression contained in t$
`b=EXIST(fname$)`	TRUE if file fname$ exist
`ret=EXEC(adr[,var])`	see command EXEC, returns int
`a=FREEFILE()`	Returns first free filenumber or -1
`f=GLOB(a$,b$[,flags])`	TRUE if a$ matches pattern b$
`b=GRAY(a)`	Gray code. if a<0: inverse Gray code
`a=HYPOT(x,y)`	returns $\sqrt{x^2+y^2}$
`c=INP(#n)`	reads character (Byte) from channel/file.
`c=INP?(#n)`	number of chars which can be read from channel/file
`a=INP&(#n)`	reads word (2 Bytes) from channel/file.
`i=INP%(#n)`	reads long (4 Bytes) from channel/file.
`a=INSTR(s1$,s2$[,n])`	tests if s2$ is contained in s1$
`a=julian(date$)`	julian day
`l=LEN(t$)`	length of string
`p=LOC(#n)`	Returns value of file position indicator
`l=LOF(#n)`	length of file
`b=LPEEK(adr)`	reads long (4 Bytes) from adress
`m=MAX(a,b,c,...)`	returns biggest value
`m=MAX(f())`	not implemented jet
`m=MIN(a,b,c,...)`	returns smallest value
`m=MIN(array())`	not implemented jet
`m=MIN(function())`	not implemented jet
`d=PEEK(a)`	reads Byte from address a
`a=RINSTR(s1$,s2$[,n])`	tests (starting from right) if s2$ is contained in s1$
`adr=SYM_ADR(#n,s$)`	return pointer to symbol with name s$ from shared Object file #n
`a=VAL(t$)`	converts String/ASCII to number
`i=VAL?(t$)`	returns number of chars which can be converted to number
`a=VARPTR(v)`	returns pointer to variable

4.10 Subroutines and Functions

4.10.1 Subroutines

Subroutines are blocks of code that can be called from elsewhere in the program. Subroutines can take arguments but return no results. They can access all variables available but also may have local variables (–> LOCAL). Subroutines are defined with

```
PROCEDURE name(argumentlist)
  ...  many commands
RETURN
```

4.10.2 User defined functions

X11-Basic functions are blocks of code that can be called from elsewhere within an expression (e.g `a=3*@myfunction(b)`). Functions can take arguments and must return a result. Variables are global unless declared local. For local variables changes outside a function have no effect within the function except as explicitly specified within the function. Functions arguments can be variables and arrays of any types. Functions can return variables of any type. By default, arguments are passed by value. Functions can be executed recursively. A function will be defined by:

```
FUNCTION name(argumentlist)
  .. many more calculations
  RETURN returnvalue
ENDFUNCTION
```

4.11 Error Messages

X11-Basic can produce a number of internal errors, which are refferred to by a number (ERR) (see also ERROR).

The meaning of this errors and their text expression is as follows:

0	Divide by zero
1	Overflow
2	Value not integer $-2147483648\dots2147483647$
3	Value not byte $0\dots255$
4	Value not short $-32768\dots32767$
5	Square root: only positive numbers
6	Logarithmen nur für Zahlen größer Null
7	Unknown Error

8	Out of Memory
9	Function or command not yet implemented
10	String too long
12	Program too long, buffer size exeeded –> NEW
14	Array () already dimensioned
15	Array () not dimensioned
16	Field index too large
17	Dim too large
18	Wrong number of indicies
19	Procedure not found
20	Label not found
21	Open only "I"nput "O"utput "A"ppend "U"pdate
22	File already opened
23	Wrong file #
24	File not opened
25	Wrong input, no number
26	EOF - reached end of file
27	Too many points for Polyline/Polyfill
28	Array must be one dimensional
30	Merge - no ASCII file
31	Merge - line too long - CANCEL
32	==> Syntax error
33	Marke nicht definiert
34	Zu wenig Data
35	Data nicht numerisch
36	Programmstruktur Fehlerhaft
37	Diskette voll
38	Befehl im Direktmodus nicht möglich
39	Programmfehler Kein Gosub möglich
40	Clear nicht möglich in For-Next-Schleifen oder Proceduren
41	Cont nicht möglich
42	Zu wenig Parameter
43	Ausdruck zu komplex
44	Funktion nicht definiert
45	Zu viele Parameter
46	Parameter falsch, keine Zahl
47	Parameter falsch, kein String
48	Open "R" - Satzlänge falsch
49	Zu viele "R"-Files (max. 31)
50	Kein "R"-File
51	Parser: Syntax Error <>
52	Fields größer als Satzlänge

54	GET/PUT Field-String Länge falsch
55	GET/PUT Satznummer falsch
56	Falsche Anzahl Parameter
57	Variable noch nicht initialisiert
58	Variable ist vom falschen Typ
60	Sprite-String-Länge falsch
61	Fehler bei RESERVE
62	Menu falsch
63	Reserve falsch
64	Pointer falsch
65	Feldgröße < 256
66	Kein VAR-Array
67	ASIN/ACOS falsch
68	Falsche VAR-Type
69	ENDFUNC ohne RETURN
70	Unbekannter Fehler 70
71	Index zu groß
80	Matrizenoperationen nur für ein- oder zweidimensionale Felder
81	Matrizen haben nicht die gleiche Ordnung
82	Vektorprodukt nicht definiert
83	Matrizenprodukt nicht definiert
84	Scalarprodukt nicht definiert
85	Transposition nur für zweidimensionale Matrizen
86	Matrix nicht quadratisch
87	Transposition nicht definiert
88	FACT/COMBIN/VARIAT nicht definiert
90	Fehler bei Local
91	Fehler bei For
92	Resume (next) nicht möglich Fatal, For oder Local
93	Stapel-Fehler
100	X11BASIC Version 1.14 Copyright (c) 1997-2007 Markus Hoffmann
101	** 1 - Segmentation fault : Speicherschutzverletzung
102	** 2 - Bus Error Peek/Poke falsch?
103	** 3 - Adress error Ungerade Wort-Adresse! Dpoke/Dpeek, Lpoke/Lpeek?
104	** 4 - Illegal Instruction : ungültiger Maschinenbefehl
105	** 5 - Divide by Zero : Division durch Null
106	** 6 - CHK exeption : CHK-Befehl
107	** 7 - TRAPV exeption : TRAPV-Befehl
108	** 8 - Privilege Violation : Privilegverletzung
109	** 9 - Trace exeption : Trace ohne Monitor
110	** 10 - Broken pipe : Ausgabeweitergabe abgebrochen
131	* Number of hash collisons exceeds maximum generation counter value.

132	* Wrong medium type : Andere Diskette einlegen
133	* No medium found : Bitte Disk einlegen
134	* Quota exceeded
135	* Remote I/O error
136	* Is a named type file
137	* No XENIX semaphores available
138	* Not a XENIX named type file
139	* Structure needs cleaning
140	* Stale NFS file handle
141	* Operation now in progress
142	* Operation already in progress
143	* No route to host
144	* Host is down
145	* Connection refused : Verbindungsaufbau verweigert
146	* Connection timed out : Zeitüberschreitung bei Verbindung
147	* Too many references: cannot splice
148	* Cannot send after transport endpoint shutdown
149	* Transport endpoint is not connected : Keine Verbindung, Verbindung unterbrochen ?
150	* Transport endpoint is already connected : Verbindung schon geöffnet
151	* No buffer space available : Speicher voll
152	* Connection reset by peer
153	* Software caused connection abort : Verbindungsabbruch durch Anwender
154	* Network dropped connection because of reset
155	* Network is unreachable
156	* Network is down
157	* Cannot assign requested address : Verbindungsaufbau nicht möglich
158	* Address already in use : Besetzt, Verbindung nicht möglich
159	* Address family not supported by protocol
160	* Protocol family not supported
161	* Operation not supported on transport endpoint
162	* Socket type not supported
163	* Protocol not supported
164	* Protocol not available
165	* Protocol wrong type for socket
166	* Message too long
167	* Destination address required
168	* Socket operation on non-socket : Operation nur mit Sockets erlaubt
169	* Too many users
170	* Streams pipe error
171	* Interrupted system call should be restarted
172	* Illegal byte sequence
173	* Cannot exec a shared library directly

174	* Attempting to link in too many shared libraries
175	* .lib section in a.out corrupted
176	* Accessing a corrupted shared library
177	* Can not access a needed shared library
178	* Remote address changed
179	* File descriptor in bad state
180	* Name not unique on network
181	* Value too large for defined data type
182	* Not a data message
183	* RFS specific error
184	* Try again : Operation zur Zeit nicht möglich
185	* Too many symbolic links encountered
186	* File name too long : Dateiname zu lang
187	* Resource deadlock would occur
188	* Advertise error
189	* Speicherblockfehler
190	* Kein Binärprogramm
191	* Link has been severed
192	* Object is remote
193	* Math result not representable
194	* Math arg out of domain of func
195	* Cross-device link
196	* Device not a stream
197	* Mount device busy
198	* Block device required
199	* Bad address
200	* No more processes
201	* No children
202	* Exchange full
203	* Interrupted system call
204	* Invalid exchange
205	* Permission denied, you must be super-user
206	* Operation auf diesem Kanal nicht (mehr) möglich
207	* Keine weiteren Dateien
208	* Link number out of range
209	* Level 3 reset
210	* Ungültige Laufwerksbezeichnung
211	* Level 2 not synchronized
212	* Channel number out of range
213	* Identifier removed
214	* No message of desired type
215	* Operation would block

216	* Ungültige Speicherblockadresse
217	* Directory not empty : Das Verzeichnis ist nicht leer
218	* Function not implemented : Unbekannter Befehl
219	* Ungültiges Handle
220	* Zugriff nicht möglich
221	* Zu viele Dateien offen
222	* Pfadname nicht gefunden
223	* Datei nicht gefunden
224	* Broken pipe : Verbindung wurde unterbrochen
225	* Too many links : Zu viele Links
226	* Read-Only File-System : File-System ist Schreibgeschützt
227	* Illegal seek : Seek falsch
228	* No space left on device : File-System ist voll
229	* File too large : File ist zu gross für diese Operation
230	* Text file busy
231	* Not a typewriter
232	* Too many open files
233	* File table overflow : Zur Zeit sind keine weiteren offenen Files möglich
234	* Invalid argument
235	* Is a directory
236	* Not a directory
237	* No such device
238	* Cross-device link
239	* File exists
240	* Bad Sektor (Verify)
241	* Unbekanntes Gerät
242	* Diskette wurde gewechselt
243	* Permission denied : Die Erlaubnis wurde verweigert
244	* Not enough core : Speicher voll
245	* Lesefehler
246	* Schreibfehler
247	* Kein Papier
248	* Sektor nicht gefunden
249	* Arg list too long : Zu viele Parameter
250	* Seek Error Spur nicht gefunden
251	* Bad Request Ungültiger Befehl
252	* CRC Fehler Disk-Prüfsumme falsch
253	* No such process
254	* Timeout
255	* IO-Error : Allgemeiner IO-Fehler

5 X11-Basic command reference

5.1 Syntax templates

This manual describes the syntax of BASIC commands and BASIC functions in a generalized form. Here is an example:

```
PRINT [#<device-number>,] <expression> [<,>|<;> [...]]
```

Those parts of the command that must appear literally in the source code (like PRINT in the example above) are all uppercase. Descriptions in angle brackets ("<>") are not meant to appear literally in the source code but are descriptive references to the element that is supposed to be used in the source code at this place, like a variable, a numeric expression etc. Optional elements are listed inside square brackets ("[]"). They may be omitted from the command line. Mutually exclusive alternatives are separated by the "|" character. Exactly one of these alternatives must appear in the command line. Finally, repetitive syntax is indicated by three dots "...". Here are some BASIC command lines that all match the syntax template above:

```
PRINT x
PRINT #1,2*y
PRINT "result = ";result
```

5.2 A

Function: `ABS()`

Syntax: `<num-result>=ABS(<num-expression>)`

DESCRIPTION:

Returns the absolute value of an expression. The absolute value is the value without regard to the sign (negative, zero or positive). The result of ABS will always be a positive number or zero.

SEE ALSO: `SGN()`

EXAMPLE:

```
PRINT ABS(-34.5),ABS(34)
       Result: 34.5     34
```

Function: ACOS()

Syntax: `<num-result>=ACOS(<num-expression>)`

DESCRIPTION:

The acos() is the arcus cosine-function, i.e. the inverse of the cos()-function. It returns the angle (in radian), which, fed to the cosine-function will produce the argument passed to the acos()-function.

EXAMPLE:

```
PRINT ACOS(0.5),ACOS(COS(PI))
Result: 1.047197551197  3.14159265359
```

SEE ALSO: `COS(), ASIN()`

*

Function: ACOSH()

Syntax: `<num-result>=ACOSH(<num-expression>)`

DESCRIPTION:

The acosh() is the inverse hyperbolic cosine function.

SEE ALSO: `COS(), ASINH()`

Command: ADD

Syntax: `ADD <num-variable>,<num-expression>`

DESCRIPTION:

Increase value of variable by result of <num-expression>.

EXAMPLE:

```
a=0
ADD a,5
```

SEE ALSO: SUB, MUL, DIV

*

Function: ADD()

Syntax: `<num-result>=ADD(<num-expression>,<num-expression>)`

DESCRIPTION:

(Integer) addition.

EXAMPLE:

```
b=ADD(a,5)
```

SEE ALSO: SUB(), MUL(), DIV()

Command: AFTER

Syntax:

```
AFTER <num-variable>, <procedure-name>
AFTER STOP
AFTER CONT
```

DESCRIPTION:

Procedures can be called after the expiry of a set time. Time in seconds. To continue the process, use CONT, and to stop use STOP.

EXAMPLE:

```
PRINT "You have 10 seconds to enter your name: "
AFTER 10,alarm
INPUT name$
END
PROCEDURE alarm
  PRINT "Time out !"
  QUIT
RETURN
```

SEE ALSO: EVERY

Command: ALERT

Syntax: `ALERT <type>,<message-string>,<default-button>, <button-string>,<num-var>[,<string-var>]`

DESCRIPTION:

Creates an alert box and asks for user input.

<type> chooses type of alert symbol, 0=none, 1="!", 2="?", 3="stop" <message-string> Contains main text. Lines are separated by the '|' symbol. Editable fields are started with a chr$(27) followed by the default text to be edited (until "|"). <button-string> Contains text for the buttons (separated by '|'). <default-button> is the button to be highlighted (0=none,1,2,...) to be selected by just pressing return. <num-var> This variable is set to the number of the button selected. <string-var> This is a string variable which holds any text-input the user made. It holds the contents of the editable fields separated by a CHR$(13).

EXAMPLES:

```
ALERT 1,"Pick a|button",1,"Left|Right",a
ALERT 0,"You pressed|Button"+STR$(a),0,"OK",a

' Example of editable fields
i=1
name$="TEST01"
posx$="N54°50'32.3"
t$="Edit waypoint:||Name:   "+chr$(27)+name$+"|"
t$=t$+"Position: "+chr$(27)+posx$+"|"
ALERT 0,t$,1,"OK|UPDATE|DELETE|CANCEL",a,f$
WHILE LEN(f$)
  WORT_SEP f$,CHR$(13),0,a$,f$
  PRINT "Field";i;": ",a$
  INC i
WEND
```

SEE ALSO: `FORM\_ALERT()`, `WORT\_SEP`, `CHR\$()`

Operator: AND

Syntax: `<num-expression1> AND <num-expression2>`

DESCRIPTION:

Used to determine if BOTH conditions are true. If both expression1 AND expression2 are true (non-zero), the result is true. Returns -1 for true, 0 for false.

Also used to compare bits in binary number operations. 1 AND 1 return a 1, all other combinations of 0's and 1's produce 0.

EXAMPLES:

```
Print 3=3 AND 4>2        Result:  -1 (true)
Print 3>3 AND 5>3        Result:   0 (false)

PRINT (30>20 AND 20<30)  Result:  -1 (true)
PRINT (4 AND 255)        Result:   4
```

SEE ALSO: NAND, OR, NOT, XOR

*

Function: AND()

Syntax: `<num-result>=AND(<num-expression>,<num-expression2>)`

DESCRIPTION:

Returns <num-expression> AND <num-expression2>

SEE ALSO: OR(), AND

Command: ARRAYCOPY

Syntax: `ARRAYCOPY d(),s()`

DESCRIPTION:

Copys the contents of array s() to d() (including dimensions). This is the same as the statement: `d()=s()`.

SEE ALSO: DIM

*

Command: ARRAYFILL

Syntax:
```
ARRAYFILL x(),n
ARRAYFILL x$(),t$
```

DESCRIPTION:

Assigns the value to all elements of an array or matrix.

EXAMPLE:

```
DIM a(100)
ARRAYFILL a(),13
PRINT a(22)      Result: 13
```

SEE ALSO: DIM

Function: `ARRPTR()`

Syntax: `<int-result>=ARRPTR(<array>())`

DESCRIPTION:

Finds the address of the descriptor of a string or field array.

SEE ALSO: `VARPTR()`

Function: `ASC()`

Syntax: `<num-result>=ASC(<string-expression>)`

DESCRIPTION:

Returns the ASCII code value (a number between 0 and 255) of the first character in a string. ASCII stands for American Standard Code for Information Interchange. ASC returns 0 if the length of string is zero or the ASCII code of the string is zero.

SEE ALSO: `CHR\$(),CVI(),CVL(),CVS()`

EXAMPLE:

```
PRINT ASC("A"), ASC("T")  Result: 65, 84
PRINT ASC("TEST")         Result: 84
```

Function: `ASIN()`

Syntax: `<num-result>=ASIN(<num-expression>)`

DESCRIPTION:

The asin() is the arcus sine-function, i.e. the inverse of the sin()-function. Or, more elaborate: It Returns the angle (in radian, not degree !), which, fed to the sine-function will produce the argument passed to the asin()-function.

SEE ALSO: `ACOS(), SIN()`

EXAMPLE:

```
PRINT 6*ASIN(0.5)     Result: 3.14159265359
```

*

Function: `ASINH()`

Syntax: `<num-result>=ASINH(<num-expression>)`

DESCRIPTION:

The asinh() function calculates the inverse hyperbolic sine of x; that is the value whose hyperbolic sine is x.

SEE ALSO: `ACOSH(), SIN()`

Keyword: `AT()`

Syntax: `PRINT AT(y,x);[...]`

DESCRIPTION:

The AT-statement takes two numeric arguments (e.g. AT(2,3)) and can be used in combination with the PRINT- or GPRINT-command.

The two numeric arguments of the AT-function may range from 0 to the width of your terminal minus 1, and from 0 to the height of your terminal minus 1; if any argument exceeds these values, it will be truncated accordingly. However, X11-Basic has no influence on the size of your terminal (80x25 is a common, but not mandatory), the size of your terminal and the maximum values acceptable within the AT-statement may vary. To get the size of your terminal you may use the CRSLN and CRSCOL variables.

SEE ALSO: `PRINT`, `GPRINT`, `TAB()`, `SPC()`

Function: ATN(), ATAN()

Syntax:
```
<num-result>=ATN(<num-expression>)
<num-result>=ATAN(<num-expression>)
```

DESCRIPTION:

Returns the angle, in radians, for the inverse tangent of expression.

SEE ALSO: ACOS(), ASIN()

EXAMPLE:

```
PRINT 4*ATAN(1)     Result: 3.14159265359
```

*

Function: ATAN2()

Syntax:
```
<num-result>=ATAN2(<num-expression>,<num-expression>)
```

DESCRIPTION:

The atan()-function has a second form which accepts two arguments: atan2(a,b) which is (mostly) equivilantly to atan(a/b) except for the fact, that the two-argument-form returns an angle in the range -pi to pi, whereas the one-argument-form returns an angle in the range -pi/2 to pi/2.

EXAMPLE:

```
 PRINT DEG(ATAN2(0,-1))     Result: 180
```

SEE ALSO: ATAN()

*

Function: `ATANH()`

Syntax: `<num-result>=ATANH(<num-expression>,<num-expression>)`

DESCRIPTION:

The atanh() function calculates the inverse hyperbolic tangent of x; that is the value whose hyperbolic tangent is x. If the absolute value of x is greater than 1.0, acosh() returns not-a-number (NaN).

SEE ALSO: `ATAN()`

5.3 B

Function: `BCHG()`

Syntax: `<num-result>=BCHG(<num-expression>,<num-expression>))`

DESCRIPTION:

Allow setting and resetting of bits.

SEE ALSO: `BSET(),BCLR()`

*

Function: `BCLR()`

Syntax: `<num-result>=BCLR(<num-expression-x>,<num-expression-y>))`

DESCRIPTION:

BCLR sets the y-th bit of x to zero.

SEE ALSO: `BSET(),BCHG()`

Command: `BEEP, BELL`

Syntax:
```
BEEP
BELL
```

DESCRIPTION:

Sounds the speaker of your terminal. This command is not a sound-interface, so you can neither vary the length or the height of the sound (technically, it just prints chr$(7)). BELL is exactly the same as BEEP.

SEE ALSO: `SOUND`

Command: BGET

Syntax: `BGET #<device-nr>,<adr>,<len>`

DESCRIPTION:

Reads <len> bytes from a data channel into an area of memory starting at address <adr> Unlike BLOAD, several different areas of memory can be read from a file.

SEE ALSO: BLOAD, BPUT

Function: BIN$()

Syntax: `<string-result>=BIN$(<num-expression>[,<num-expression>)])`

DESCRIPTION:

The bin$()-takes a numeric argument an converts it into a string of binary digits (i.e. ’0’ and ’1’). The length of the output, number of digits can be specified by the optional second argument.

EXAMPLE:

```
        PRINT BIN$(64),BIN$(-2000,16)
Result: 01000000        1111100000110000
```

SEE ALSO: HEX\$()

Command: `BLOAD`

Syntax: `BLOAD <filename>,<address>`

DESCRIPTION:

BLOAD reads the specified file into memory. The adress space <address> is pointing to should be allocated before. You should check if the file exists prior to using this function. This command is meant to be used for loading binary data. To load a text file, use OPEN and INPUT # to remain compatible with other BASIC implementations.

SEE ALSO: `MALLOC()`, `BGET`, `INPUT`, `INPUT\$()`, `BSAVE`

Command: `BMOVE`

Syntax: `BMOVE <scr>,<dst>,<len>`

DESCRIPTION:

Fast movement of memory blocks.

<scr> is the address at which the block to be moved begins. <dst> is the address to which the block is to moved. <len> is the length of the block in bytes.

SEE ALSO: `PEEK()`, `POKE`, `BLOAD`, `BSAVE`

Command: `BOUNDARY`

Syntax: `BOUNDARY <num-expression>`

DESCRIPTION:

Switch off (or on) borders on filled shapes (PBOX, PCIRCLE ..). If the argument is zero - no border will be drawn.

SEE ALSO: `PBOX`, `PCIRCLE`

Command: `BOX`

Syntax: `BOX <x>,<y>,<x2>,<y2>`

DESCRIPTION:

Draws a rectangle with corners at (x,y) and (x2,y2).

SEE ALSO: `PBOX`

Command: BPUT

Syntax: `BPUT #<device-nr>,<adr>,<len>`

DESCRIPTION:

Reads <len> bytes from an area of memory starting at <adr> out to a data channel.

SEE ALSO: BGET

Command: `BREAK`

Syntax: `BREAK`

DESCRIPTION:

BREAK transfers control immediately outside the enclosing loop or select statement. This is the preferred way of leaving such a statement (rather than goto).

SEE ALSO: `EXIT IF`

Command: BSAVE

Syntax: `<filename>,<adr>,<len>`

DESCRIPTION:

Save <len> bytes in memory from adress <adr> to file. This command is meant be be used for saving binary data obtained via BLOAD. To save text files, use OPEN and PRINT # to remain compatible with other BASIC implementations.

SEE ALSO: BLOAD, BPUT

Function: `BSET()`

Syntax: `<num-result>=BSET(<num-expression-x>,<num-expression-y>)`

DESCRIPTION:

Allows setting and resetting of bits. BSET sets the y-th bit of x to 1.

SEE ALSO: `BCHG(), BCLR(), BTST()`

Function: `BTST()`

Syntax: `<bool-result>=BTST(<num-expression-x>,<num-expression-y>)`

DESCRIPTION:

BTST results in -1 (TRUE) if bit y of x is set.

SEE ALSO: `BCHG(), BCLR(), BSET()`

Function: `BWTD$()`

Syntax: `b$=BWTD$(a$)`

DESCRIPTION:

BWTD$() performs the inverse Burrows-Wheeler transform on the string a$.

SEE ALSO: `BWTE\$()`

Function: `BWTE$()`

Syntax: `b$=BWTE$(a$)`

DESCRIPTION:

BWTE$() performs a Burrows-Wheeler transform on the string a$.

The Burrows-Wheeler transform (BWT) is an algorithm used in data compression techniques such as bzip2. It was invented by Michael Burrows and David Wheeler.

When a character string is transformed by the BWT, none of its characters change. It just rearranges the order of the characters. If the original string had several substrings that occurred often, then the transformed string will have several places where a single character is repeated multiple times in a row. This is useful for compression, since it tends to be easy to compress a string that has runs of repeated characters by techniques such as run-length encoding.

SEE ALSO: `BWTD\$()`

Function: `BYTE()`

Syntax: `<num>=BYTE(<num-expression>)`

DESCRIPTION:

Returns lower 8 bits of argument. (same as a=b AND 255)

SEE ALSO: `CARD()`, `WORD()`, `SWAP()`

5.4 C

Command: CALL

Syntax: `CALL <adr>[,<parameter-list>]`

DESCRIPTION:

Calls a machine code or C subroutine at address <adr> without return value. Optinal parameters are passed on the stack. (like in C). The default parameter-type is (4-Byte) integer. If you want to specify other types, please use prefixes:

D: – double (8-Bytes) F: – float (4-Bytes) L: – long int

SEE ALSO: EXEC, EXEC()

EXAMPLE:

```
DIM result(100)
LINK #1,"simlib.so"
adr=SYM_ADR(#1,"CalcBeta")
CALL adr,D:1.2,L:0,L:VARPTR(result(0))
UNLINK #1
```

Function: `CARD()`

Syntax: `<num>=CARD(<num-expression>)`

DESCRIPTION:

Returns lower 16 bits of b. (same as a=b AND (2(HOCH)16-1))

SEE ALSO: `BYTE()`, `WORD()`, `SWAP()`

Keyword: `CASE`

Syntax: `SELECT ... CASE <num-expression> ... ENDSELECT`

DESCRIPTION:

See SELECT.

SEE ALSO: `SELECT`, `DEFAULT`, `ENDSELECT`

Function: `CBRT()`

Syntax: `a=CBRT(x)`

DESCRIPTION:

The CBRT() function returns the cube root of x. This function cannot fail; every representable real value has a representable real cube root.

SEE ALSO: `SQRT()`

Function: `CEIL()`

Syntax: `<num-result>=CEIL(<num-expression>)`

DESCRIPTION:

Ceiling function: return smallest integral value not less than argument.

SEE ALSO: `INT()`

Command: `CHAIN`

Syntax: `CHAIN <file-name>`

DESCRIPTION:

CHAIN loads and runs another BASIC program. Global variables will be available with their current value to the new program, all other variables are erased. If you want to append another program to the current program (as opposed to erasing the current program and loading a new program), use the MERGE command instead.

SEE ALSO: `MERGE`, `RUN`

Function: CHR$()

Syntax: `<string-result> = CHR$(<num-expression>)`

DESCRIPTION:

CHR$() returns the character associated with a given ASCII code.
Character table

032		048	0	064	@	080	P	096	‘	112	p
033	!	049	1	065	A	081	Q	097	a	113	q
034	"	050	2	066	B	082	R	098	b	114	r
035	\#	051	3	067	C	083	S	099	c	115	s
036	\$	052	4	068	D	084	T	100	d	116	t
037	\%	053	5	069	E	085	U	101	e	117	u
038	\&	054	6	070	F	086	V	102	f	118	v
039	’	055	7	071	G	087	W	103	g	119	w
040	(	056	8	072	H	088	X	104	h	120	x
041	)	057	9	073	I	089	Y	105	i	121	y
042	*	058	:	074	J	090	Z	106	j	122	z
043	+	059	;	075	K	091	\$[\$	107	k	123	{
044	,	060	<	076	L	092	\	108	l	124	|
045	-	061	=	077	M	093	\$]\$	109	m	125	}
046	.	062	>	078	N	094	(HOCH)	110	n	126	˜
047	/	063	?	079	O	095	_	111	o	127	

Control codes

```
00 NUL       08 BS   -- Backspace   16 DLE
01 SOH       09 HT   -- horizontal TAB   17 DC1  -- XON
02 STX       10 LF   -- Newline   18 DC2
03 ETX       11 VT   19 DC3  -- XOFF
04 EOT       12 FF   -- Form feed   20 DC4
05 ENQ       13 CR   -- Carriage Return  21 NAK
06 ACK       14 SO   22 SYN
07 BEL  -- Bell  15 SI   23 ETB

24 CAN       32 SP  -- Space
25 EM       127 DEL -- Delete
26 SUB
```

```
27 ESC28 FS
29 GS
30 RT
31 US
```

EXAMPLE:

```
PRINT CHR$(34);"Hello World !";CHR$(34)
Result: "Hello World !"
```

SEE ALSO: `ASC()`

Function: `CINT()`

Syntax: `<num-result>=CINT(<num-expression>)`

DESCRIPTION:

CINT() returns the rounded absolute value of its argument prefixed with the sign of its argument.

EXAMPLE:

```
PRINT CINT(1.4), CINT(-1.7)
       Result: 2, -2
```

SEE ALSO: `INT(),FRAC(),TRUNC(),ROUND()`

Command: `CIRCLE`

Syntax: `CIRCLE <x>,<y>,<r>[,<w1>,<w2>]`

DESCRIPTION:

Draw a circle with actual color (and fillpattern). The x- and y-coordinates of the center and the radius of the circle in screen coordinates. Optionally a starting angle <w1> and stop angle <w2> can be passed to draw a circular arc.

EXAMPLE:

```
CIRCLE 100,100,50
```

SEE ALSO: `ELLIPSE`, `COLOR`, `DEFFILL`, `PCIRCLE`

Command: `CLEAR`

Syntax: `CLEAR`

DESCRIPTION:

Clear all variables and arrays as if they were never used before.

SEE ALSO: `NEW`

Command: `CLEARW`

Syntax: `CLEARW [<num>]`

DESCRIPTION:

Clear graphic window. If a number is given, clear window with the number given. The Window is filled with the backgound color given by the X-server.

SEE ALSO: `CLOSEW`

Command: `CLIP`

Syntax: `CLIP x,y,w,h[,ox,oy]`

DESCRIPTION:

This command provide the ’Clipping’ function, ie. the limiting of graphic display within a specified rectangular screen area. The command CLIP x,y,w,h defines the clipping rectangle starting at upper left coordinates x,y and extends w wide and h high. The optional additional command parameters ox,oy make it possible to redefine the origin of the graphic display.

SEE ALSO:

Command: CLOSE

Syntax: `CLOSE [[#]<num-expression>[,[#]<num-expression>,...]]`

DESCRIPTION:

This statement is used to CLOSE one or more OPEN files or other devices. The parameter expression indicates a device number or file number. If no file or device numbers are declared all OPEN devices will be closed.

COMMENT:

All files should be closed before leaving a program to insure that data will not be lost or destroyed. If a program exit is through END or QUIT, all files will be closed. If a program is stopped with the STOP command, alle open files remain open.

EXAMPLE:

```
CLOSE #1,#2
CLOSE
```

SEE ALSO: OPEN, LINK

Command: `CLOSEW`

Syntax: `CLOSEW [<num>]`

DESCRIPTION:

Close graphic window (make it disappear from the screen). If a number is given, closes window with the number given. The Window will again be opened, when the next graphic command is executed.

SEE ALSO: `CLEARW`

Command: `CLR`

Syntax: `CLR <var>[,<var>,...]`

DESCRIPTION:

Sets specified variables to 0 or "".

Command: `CLS`

Syntax: `CLS`

DESCRIPTION:

Clear text screen and move cursor home (upper left corner).

SEE ALSO: `PRINT`

Command: COLOR

Syntax: `COLOR <foreground-color>[,<background-color>]`

DESCRIPTION:

COLOR sets the foreground color (and optionally the background color) for graphic output into the graphic window. The color values are dependant of the color depth of the Screen/X-Server. Usually the COLOR statement is used together with the GET_COLOR() function, so arbitrary colors may be used.

EXAMPLE:

```
yellow=GET_COLOR(65535,65535,0)
blue=GET_COLOR(0,0,65535)
COLOR yellow,blue
```

SEE ALSO: GET_COLOR, LINE

Function: `COMBIN()`

Syntax: `<num-result>=COMBIN(<n>,<k>)`

DESCRIPTION:

Calculates the number of combinations of <n> elements to the <k>th class without repetitions. Defined as z=n!/((n-k)!*k!).

Function: `COMPRESS$()`

Syntax: `c$=COMPRESS$(a$)`

DESCRIPTION:

Performs a losless compression on the String a$. The algorithm uses run length encoding in combination with the Burrows-Wheeler transform. The result is a better compression than p.ex. the algorithm used by gzip. At the moment the COMPRESS$() function is identical to following combination: b$=ARIE$(RLE$(MTFE$(BWTE$(RLE$(a$)))))

SEE ALSO: `UNCOMPRESS\$(), BWTE\$(), RLE\$(), MTFE\$()`

Command: `CONNECT`

Syntax: `CONNECT #n,server$,port%`

DESCRIPTION:

Initiate a connection on a socket.

The file number #n must refer to a socket. If the socket is of type "U" then the server$ address is the address to which packets are sent by default, and the only address from which packets are received. If the socket is of type "S","A","C", this call attempts to make a connection to another socket. The other socket is specified by server$, which is an address in the communications space of the socket.

Generally, connection-based protocol sockets may success fully connect only once; connectionless protocol sockets may use connect multiple times to change their associa tion.

SEE ALSO: `OPEN`, `CLOSE`, `SEND`, `RECEIVE`

Command: `CONT`

Syntax: `CONT`

DESCRIPTION:

Resumes execution of a program. Continue the execution of a program after interruption.

Command: `COPYARCA`

Syntax: `COPYAREA x,y,w,h,xd,yd`

DESCRIPTION:

Copies a rectangular screen sections given by x,y,w,h to a destination at xd,yd.

x,y top left corner of source rectangle w,h width & height " " " xd,yd destination x and y coordinates

This command is very fast compared to the GET and PUT commands because the whole data transfer takes place on the X-client.

SEE ALSO: `GET`, `PUT`, `GRAPHMODE`

Function: `COS()`

Syntax: `<num-result>=COS(<num-expression>)`

DESCRIPTION:

Returns the Cosine of the expression in radians.

SEE ALSO: `SIN(), ASIN()`

EXAMPLE:

```
PRINT COS(0)     Result: 1
```

*

Function: `COSH()`

Syntax: `<num-result>=COSH(<num-expression>)`

DESCRIPTION:

The cosh() function returns the hyperbolic cosine of x, which is defined mathematically as (exp(x)+exp(-x))/2

SEE ALSO: `COS(), ACOSH()`

*

Function: `CRC()`

Syntax: `<num-result>=CRC(t$[,oc])`

DESCRIPTION:

Calculates a 32 bit checksum on the given String. If oc is passed, the checksum will be updated with the given String.

SEE ALSO: `LEN()`

Variable: `CRSCOL, CRSLIN`

Syntax:
```
CRSCOL
CRSLIN
```

DESCRIPTION:

Returns current cursor line and column.

SEE ALSO: `PRINT AT()`

Variable: `CTIMER`

Syntax: `CTIMER`

DESCRIPTION:

Returns CPU-Clock in seconds. This timer returns the amount of time this application was running. It is most usefull for benchmark applications on multi tasking environments.

SEE ALSO: `TIMER`, `STIMER`

Function: `CVA()`

Syntax: `<array-result>=CVA(<string-expression>)`

DESCRIPTION:

Returns array reconstructed from the string. This function is the compliment of MKA$().

EXAMPLE:

```
a()=CVA(t$)
```

SEE ALSO: `ASC(), CVF(), CVL(), MKA\$()`

Function: `CVD()`

Syntax: `<num-result>=CVD(<string-expression>)`

DESCRIPTION:

Returns the binary double value of the first 8 characters of string. This function is the compliment of MKD$().

SEE ALSO: `ASC(),CVF(),CVL(),MKD\$()`

*

Function: `CVF()`

Syntax: `<num-result>=CVF(<string-expression>)`

DESCRIPTION:

Returns the binary float value of the first 4 characters of string. This function is the compliment of MKF$().

SEE ALSO: `ASC(),CVD(),CVL(),MKF\$()`

*

Function: `CVI()`

Syntax: `<num-result>=CVI(<string-expression>)`

DESCRIPTION:

Returns the binary integer value of the first 2 characters of string. This function is the compliment of MKI$(). Null string returns 0, One character strings will return the ASCII value.

SEE ALSO: `ASC(),CVF(),CVL(),MKI\$()`

*

Function: `CVL()`

Syntax: `<num-result>=CVL(<string-expression>)`

DESCRIPTION:

Returns the binary long integer value of the first 4 characters of string. This function is the compliment of MKL$(). Null string returns 0.

SEE ALSO: `ASC(),CVF(),CVI(),MKL\$()`

*

Function: `CVS()`

Syntax: `<num-result>=CVS(<string-expression>)`

DESCRIPTION:

Returns the binary float value of the first 4 characters of string. This function is the compliment of MKS$().

SEE ALSO: `CVF(),MKS\$()`

5.5 D

Command: `DATA`

Syntax: `DATA [<const>[,<const>, ...]]`

DESCRIPTION:

The DATA statement is used to hold information that may be read into variables using the READ statement. DATA items are a list of string or numeric constants separated by commas and may appear anywhere in a program. No comment statement may follow the DATA statement on the same line. Items are read in the order they appear in a program. RESTORE will set the pointer back to the beginning of the first DATA statement.

Alphanumeric string information in a DATA statement need not be enclosed in quotes if the first character is not a number, math sign or decimal point. Leading spaces will be ignored (unless in quotes). DATA statements can be included anywhere within a program and will be read in order.

SEE ALSO: `READ`, `RESTORE`

Variable: DATE$

Syntax: `DATE$`

DESCRIPTION:

Returns the system date. The format is DD.MM.YYYY.

SEE ALSO: `TIME\$`

Command: `DEC`

Syntax: `DEC <num-variable>`

DESCRIPTION:

Decrement Variable a. The result is a=a-1.

SEE ALSO: `INC`

Keyword: `DEFAULT`

Syntax: `SELECT ... DEFAULT ... ENDSELECT`

DESCRIPTION:

See SELECT.

SEE ALSO: `SELECT`

Command: `DEFFILL`

Syntax: `DEFFILL <col>,<style>,<pattern>`

DESCRIPTION:

Sets fill colour and pattern. <col> - not used at the moment <style> - 0=empty, 1=filled, 2=dots, 3=lines, 4=user (not used) <pattern> - 24 dotted patterns and 12 lined can by chosen.

SEE ALSO: `DEFLINE`, `DEFTEXT`

Command: DEFFN

Syntax: `DEFFN <function-name>[$][(<variable list>)]=<expression>`

DESCRIPTION:

This statement allows the user to define a single line inline function that can thereafter be called by FN name or @name. This is a handy way of adding functions not provided in the language. The expression may be a numeric or string expression and must match the type the function name would assume if it was a variable name. The name must adhere to variable name syntax.

EXAMPLES:

```
DEFFN av(x,y)=SQR(x^2+y^2)
a=@av(b,c)   ! call av
DEFFN add$(a$,b$)=a$+b$
```

SEE ALSO: FUNCTION, GOSUB

Command: `DEFLINE`

Syntax: `DEFLINE <style>,<thickness>[,<begin_s>,<end_s>]`

DESCRIPTION:

Sets line style, width and type of line start and end. <style> – determines the style of line: 1 Solid line 2 Long dashed line 3 Dotted 4 Dot-dashed 5 Dashed 6 Dash dot dot .. 7 User defined (not used) <thickness> – sets line width in pixels. <begin_s>,<end_s> – The start and end symbols are defined by the last parameter, and can be: 0 Square 1 Arrow 2 Round

SEE ALSO: `LINE`, `DEFFILL`

Command: `DEMARK`

Syntax: `DEMARK <color>,<style>,<size>`

DESCRIPTION:

Sets colour,type and size of the corner points to be marked using the command POLYMARK. The color value will be ignored. The color of the points can be set with the COLOR command. The following typesare possible : 0=point 1=dot (circle) 2=plus sign 3=asterisk 4=square 5=cross 6=hash 8=filled circle 9=filled square

SEE ALSO: `POLYMARK`, `DEFLINE`, `COLOR`

Command: `DEMOUSE`

Syntax: `DEMOUSE <style>`

DESCRIPTION:

Chooses a pre-defined mouse form. The following mouse forms are available :

0=arrow 1=expanded (rounded) X 2=busy bee 3=hand, pointing finger 4=open hand 5=thin crosswire 6=thick crosswire 7=bordered crosswire and about 100 other X-Window specific symbols.

SEE ALSO: `HIDEM`, `SHOWM`

Command: `DEFTEXT`

Syntax: `DEFTEXT <colour>,<attr>,<angle>,<height>,<width>`

DESCRIPTION:

Defines the colour,style,rotation and size of text to be printed using the LTEXT command.

<colour> not used. <attr> text style - 0=normal 1=bold 2=light 4=italic 8=underlined 16=outlined (can be combined). <angle> rotation in degrees <width> and <height> size of text in % (100% correspondt to 100 Pixel font)

SEE ALSO: `LTEXT`, `TEXT`

Function: `DEG()`

Syntax: `<num-expression>=DEG(<num-expression_x>)`

DESCRIPTION:

Converts x from radians to degrees.

SEE ALSO: `RAD()`

Command: `DELAY`

Syntax: `DELAY <num-of-seconds>`

DESCRIPTION:

Same as PAUSE. Delays program execution by <num-of-seconds> seconds.

SEE ALSO: `PAUSE`

Function: `DET()`

Syntax: `d=DET(a())`

DESCRIPTION:

Calculates the determinant of a (square) matrix a().

SEE ALSO: `SOLVE()`, `INV()`

Command: DIM

Syntax: `DIM <arrayname>(<indices>)[,<arrayname>(<indices>),...]`

DESCRIPTION:

Sets the dimensions of an array or string array.

EXAMPLES:

```
DIM a(10)
DIM b(100,100)
DIM c$(20,30,405,6)
```

SEE ALSO: ERASE, DIM?()

*

Function: DIM?()

Syntax: `<num-result>=DIM?(<array-name>())`

DESCRIPTION:

Determines the number of elements in an array. Note - arrays have an element ’0’.

EXAMPLE:

```
DIM a(10,10)
PRINT DIM?(A())     Result: 121
```

SEE ALSO: DIM

Command: `DIV`

Syntax: `DIV <num-var>,<num-expression>`

DESCRIPTION:

Divides the value of var by n. As var=var/n but faster.

SEE ALSO: `ADD`, `MUL`, `SUB`

*

Function: `DIV()`

Syntax: `<num-result>=DIV(<num-expression>,<num-expression>)`

DESCRIPTION:

Divides the first value by second.

SEE ALSO: `ADD()`, `MUL()`, `SUB()`

Command: DO

Syntax: `DO ... LOOP`

DESCRIPTION:

DO implements an unconditional loop. The lines between the DO line and the LOOP line form the loop body. The unconditional DO...LOOP block simply loops and the only way out is by EXIT IF or BREAK (or GOTO).

SEE ALSO: `LOOP`, `EXIT IF`, `BREAK`, `WHILE`

EXAMPLE:

```
DO
    INPUT a$
    EXIT IF a$=""
LOOP
```

Keyword: `DOWNTO`

Syntax: `FOR ... DOWNTO ...`

DESCRIPTION:

Used within a FOR..NEXT loop as a counter. Instead of using STEP -1, the command DOWNTO is used, however STEP is not possible with DOWNTO. eg: FOR c=100 DOWNTO 1 is the same as FOR c=100 TO 1 STEP -1

SEE ALSO: `FOR`, `TO`, `NEXT`, `STEP`

Function: `DPEEK()`

Syntax: `<num-result>=DPEEK(<adr>)`

DESCRIPTION:

Reads 2 bytes from address <adr> (a word).

SEE ALSO: `PEEK()`, `LPEEK()`, `DPOKE`

*

Command: `DPOKE`

Syntax: `DPOKE <adr>,<num-expression>`

DESCRIPTION:

Writes <num-expression> as a 2 byte word to address <adr>.

SEE ALSO: `PEEK()`, `LPEEK()`, `POKE`, `DPEEK()`

Command: `DRAW`

Syntax: `DRAW [<x1>,<y1>][TO <x2>,<y2>][TO <x3>,<y3>][TO ...]`

DESCRIPTION:

Draws points and connects two or more points with straight lines. DRAW x,y is the same as PLOT x,y. DRAW TO x,y connects the point to the last set point (set by PLOT, LINE or DRAW).

SEE ALSO: `LINE`, `PLOT`

Command: `DUMP`

Syntax: `DUMM [t$]`

DESCRIPTION:

Query Information about stored Variables, names:

DUMP – Lists all used variable names DUMP "@" – list of functions and procedures DUMP ":" – list of all labels DUMP "#" – list of open Files DUMP "K" – list of X11-Basic commands DUMP "F" – list of X11-Basic functions

SEE ALSO: `LIST`, `PLIST`, `HELP`

5.6 E

Command: `ECHO`

Syntax:

```
ECHO ON
ECHO OFF
```

DESCRIPTION:

Switches the trace function on or off. This causes each command to be listed on the stdout.

SEE ALSO: `TRON`, `TROFF`

Command: `EDIT`

Syntax: `EDIT`

DESCRIPTION:

EDIT invokes the standart editor (given by the environment variable $(EDITOR) to edit the BASIC program in memory. The command invokes the following actions:

- SAVE "name.   " writes the BASIC-program into a temporary file, - calls the editor '$EDITOR', waits until editor is closed - NEW clears internal values - LOAD "name.   " reads the BASIC-program from the temporary file.

You may want to SAVE the file before using the EDIT command if the file has not yet been saved in order to choose a name at that occasion. The default name is "name.   ". This command requires that the editor installed on your system does not detach itself from the calling process or EDIT will not recognize any changes (in that case, use LOAD to load the modified source code).

SEE ALSO: `LOAD`, `SAVE`

Command: ELLIPSE

Syntax: `ELLIPSE <x>,<y>,<a>,<b> [,<w0>,<w1>]`

DESCRIPTION:

Draws an ellipse at <x>,<y>, having <a> as horizontal radius and <b> vertical radius The optional angles <w0> and <w1> give start and end angles in degrees, to create an elliptical arc.

SEE ALSO: `PELLIPSE`, `CIRCLE`

Command: ELSE, ELSE IF

Syntax: `ELSE`
`ELSE IF <expression>`

DESCRIPTION:

ELSE IF <expression> introduces another condition block and the unqualified ELSE introduces the default condition block in a multi-line IF statement. **SEE ALSO:** `IF`,

`ENDIF`

EXAMPLE:

```
IF (N=0)
    PRINT "0"
ELSE IF (N=1)
    PRINT "1"
ELSE
    PRINT "Out of range"
ENDIF
```

Command: `END`

Syntax: `END`

DESCRIPTION:

END terminates program execution. The interpreter switches to interactive mode.

SEE ALSO: `STOP`, `QUIT`

Command: `ENDFUNCTION`

Syntax: `ENDFUNCTION`

DESCRIPTION:

Terminates a user defined function. The Program must return with a RETURN command.

SEE ALSO: `FUNCTION`, `RETURN`

Command: `ENDIF`

Syntax: `ENDIF`

DESCRIPTION:

ENDIF terminates a multi-line IF block.

SEE ALSO: `IF`, `ELSE`, `ELSE IF`

Command: `ENDSELECT`

Syntax: `ENDSELECT`

DESCRIPTION:

Terminates a SELECT block.

SEE ALSO: `SELECT`, `DEFAULT`, `CASE`

Command: `ENV$()`

Syntax: `<string-result>=ENV$(<env-variable>)`

DESCRIPTION:

ENV$() returns the current value of the specified "environment variable". Environment variables are string variables maintained by the operating system. These variables typically are used to save configuration information. Use the SETENV command to set the values of environment variables.

SEE ALSO: `SETENV`

EXAMPLE:

```
PRINT ENV$("USER")
Result: hoffmann
```

Function: `EOF()`

Syntax: `<boolean-result>=EOF(#<dev-number>)`

DESCRIPTION:

EOF() checks the end-of-file status of a file previously opened for reading by the OPEN command. It returns -1 (TRUE) if the end of file has been reached, otherwise null (FALSE).

SEE ALSO: `OPEN`

EXAMPLE:

```
OPEN "I",#1,"filename"
WHILE NOT EOF(#1)
    LINEINPUT #1,a$
WEND
CLOSE #1
```

Operator: EQV

Syntax: `<num-result>=<num-expression> EQV <num-expression>`

DESCRIPTION:

The operator EQV (equivalence) produces a TRUE result only if the arguments of both are either TRUE or both FALSE. (same as NOT(x XOR y)) and ((A IMP B) AND (B IMP A)).

table: A | B | A EQV B ——+——+———— -1 | -1 | -1 -1 | 0 | 0 0 | -1 | 0 0 | 0 | -1

SEE ALSO: TRUE, FALSE, NOT, XOR, IMP

*

Function: EQV()

Syntax: `<num-result>=EQV(<num-expression>,<num-expression>)`

DESCRIPTION:

Binary Function of logical operator EQV.

SEE ALSO: EQV

EXAMPLE:

```
        PRINT BIN$(EQV(15,6),4)
Result:  0110
```

Command: `ERASE`

Syntax: `ERASE <array>()[,<array>(),...]`

DESCRIPTION:

Deletes an array and releases the dimensioned area.

SEE ALSO: `DIM`

Variable: `ERR`

Syntax: `ERR`

DESCRIPTION:

Returns the error code of latest occurred error.

SEE ALSO: `ERROR`, `ERR\$()`

Function: ERR$()

Syntax: `<string-result>=ERR$(<error-nr)`

DESCRIPTION:

Returns, as a string containing the X11-Basic error mesage which belongs to the error number.

EXAMPLE:

```
PRINT "X11-Basic Error messages:"
FOR i=0 TO 255
  PRINT i,ERR$(i)
NEXT i
```

SEE ALSO: ERR

Command: ERROR

Syntax: `ERROR <error-number>`

DESCRIPTION:

ERROR simulates an error, i.e., displays the message appropriate for a given error code or calls the error handler if one was installed via the ON ERROR command. This command is helpful in writing ON ERROR GOSUB routines that can identify errors for special treatment and then ERROR ERR (i.e. default handling) for all others.

EXAMPLE:

```
> ERROR 245
Line -1: * Timeout
```

SEE ALSO: ON ERROR GOSUB, ERR

Command: EVAL

Syntax: `EVAL a$`

DESCRIPTION:

Evaluate or execute X11-Basic command, which is in a$.

EXAMPLE:

```
b$="a=5"
a$="print a"
EVAL a$
EVAL b$
EVAL a$
```

SEE ALSO: EVAL(), &

Function: EVAL()

Syntax: `a=EVAL(b$)`

DESCRIPTION:

Evaluate expression, which is in b$.

EXAMPLE:

```
b$="sin(0.5*exp(0.001))"
result=EVAL(b$)
```

SEE ALSO: EVAL, &

Function: `EVEN()`

Syntax: `<bool-result>=EVEN(<num-expression>)`

DESCRIPTION:

Returns true (-1) if the number is even, else false (0).

SEE ALSO: `ODD()`

Command: `EVENT`

Syntax: `EVENT typ,[x,y,xroot,yroot,s,k,ks,t$]`

DESCRIPTION:

Waits until any of the KEYEVENT, MOUSEEVENT, MOTIONEVENT occurs.

typ determines whichof the events have occured:

typ=6 — motionevent typ=14 — MOUSEEVENT typ=2 — keyevent

x,y — Mouse position relative to window xroot,yroot — Mouse position relative to screen s — State of the Alt, Caps, Shift keys t$ — Character of pressed key

SEE ALSO: `KEYEVENT`, `MOUSEEVENT`, `MOTIONEVENT`

Command: `EVERY`

Syntax:
```
EVERY <seconds> GOSUB <procedure>
EVERY CONT
EVERY STOP
```

DESCRIPTION:

The command EVERY causes the procedure to be called every <seconds> seconds. Using EVERY STOP, the calling of a procedure can be prevented. With EVERY CONT this is again allowed.

SEE ALSO: `AFTER`

Command: `EXEC`

Syntax: `EXEC <adr>[,<parameter-list>]`

DESCRIPTION:

Calls a machine code or C subroutine at address <adr> without return value. Optinal parameters are passed on the stack. (like in C). The default parameter-type is (4-Byte) integer. If you want to specify other types, please use prefixes:
D: – double (8-Bytes) F: – float (4-Bytes) L: – long int

SEE ALSO: `CALL`, `EXEC()`

*

Function: `EXEC()`

Syntax: `<int-return>=EXEC(<adr>[,<parameter-list>])`

DESCRIPTION:

Calls a machine code or C subroutine at address <adr> and returns an integer value. Optinal parameters are passed on the stack. (like in C). The default parameter-type is (4-Byte) integer. If you want to specify other types, please use prefixes:
D: – double (8-Bytes) F: – float (4-Bytes) L: – long int

SEE ALSO: `CALL`, `EXEC`

Function: `EXIST()`

Syntax: `<bool-result>=EXIST(<filename>)`

DESCRIPTION:

Returns TRUE (-1) if the file is present on a file system.

Command: `EXIT IF`

Syntax: `EXIT IF <expression>`

DESCRIPTION:

The innermost loop will be exited if the expression is true. WHILE, REPEAT, DO and FOR loops can be aborted prematurely with the EXIT command. EXIT leaves the current (innermost) loop immediately. EXIT IF leaves the current loop only if the expression after EXIT IF is not FALSE (not null).

SEE ALSO: `DO`, `WHILE`, `FOR`, `REPEAT`, `BREAK`, `IF`

Function: `EXP()`

Syntax: `<num-result> = EXP(<num-expression>)`

DESCRIPTION:

EXP() returns the exponential value of its argument (e to the specified power).

SEE ALSO: `Operator ^`

EXAMPLE:

```
PRINT EXP(1)
Result: 2.718281828459
```

Function: EXPM1()

Syntax: `<num-result> = EXPM1(<num-expression>)`

DESCRIPTION:

Returns a value equivalent to ‘exp(x)-1’. It is computed in a way that is accurate even if the value of x is near zero–a case where ‘exp(x)-1’ would be inaccurate due to subtraction of two numbers that are nearly equal.

SEE ALSO: `LOG1P()`, `EXP()`

EXAMPLE:

```
PRINT EXPM1(1)
Result: 1.718281828459
```

5.7 F

Function: `FACT()`

Syntax: `<num-result>=FACT(<num-expression>)`

DESCRIPTION:

Calculates the factorial (n!)

Variable: `FALSE`

Syntax: `FALSE`

DESCRIPTION:

Constant 0. This is simply another way of expressing the value of a condition when it is false and is equal to zero.

SEE ALSO: `TRUE`

Command: FFT

Syntax: `FFT a(),`

DESCRIPTION:

FFT a(),i $[$,...$]$ – Fouriertransformation. i=-1 Rücktransformation Dim?(a()) muss Zweierpotenz sein. **SEE ALSO:**

Command: FILESELECT

Syntax: `FILESELECT <title$>,<path$>,<default$>,<string-variable>`

DESCRIPTION:

Opens a fileselect box. <title$> gives a short title to be placed in the fileselect box. Such as ’Select a .DOC file to open...’.

<path$> path - if none specified then the default path is assumed. The pathname should include a complete path specification including a drive letter (except for UNIX file system), colon, path, and filemask. The filemask may (and usually does include wildcard characters).

<default$> contains the name of the file to apppear in the selection line. ("" for no default).

FILESELECT returns the selected filename (including path) in <string-variable>. If CANCEL is selected an empty string is returned.

SEE ALSO: `XLOAD`, `XRUN`, `FSEL\_INPUT()`

EXAMPLE:

```
FILESELECT "LOAD File",".\*.dat","input.dat",file$
```

Command: `FILL`

Syntax: `FILL x,y[,bc]`

DESCRIPTION:

Fills a bordered area with a color commencing at the co-ordinates ’x,y’. If a border color (bc) is specified, the fill stops at boundaries with this color. If no border color is given, the fill will stop at any oter color than the one of the starting coordinate. The fill color can be chosen with the command COLOR.

SEE ALSO: `COLOR`

FIT a(),a(),???,???$[$,???,???,???,???,???,???$]$ FIT_LINEAR x(),y()$[$,$[$xerr(),$]$yerr()$]$,n,a,b
– Lineare Regression optional mit Fehlerbalken in beide Richtungen. n=Anzahl der Fitpunkte, f(x)=a+b*x

Function: `FIX()`

Syntax: `<num-result>=FIX(<num-expression-x>)`

DESCRIPTION:

Returns the integer of x after it has been rounded. Same as INT(x) for positive numbers but for negative numbers INT(-1.99)=-2 AND FIX(-1.99)=1. FIX is identical to the function TRUNC and complements FRAC.

SEE ALSO: `INT(), TRUNC(), FRAC(), ROUND()`

Function: `FLOOR()`

Syntax: `<num-result>=FLOOR(<num-expression-x>)`

DESCRIPTION:

Round x down to the nearest integer.

SEE ALSO: `INT()`, `FIX()`

Command: `FLUSH`

Syntax: `FLUSH [#<device-name>]`

DESCRIPTION:

Flushes the output to the file or console. Usually a Line is printed when the newline character is encountered. To enforce output of everything which has been printed so far use FLUSH.

SEE ALSO: `PRINT`

Command: FOR

Syntax: `FOR <variable> = <start-expression> TO <target-expression> [STEP <inc`
`FOR <variable> = <start-expression> DOWNTO <target-expression> [STEP <`

DESCRIPTION:

FOR initiates a FOR...NEXT loop with the specified <variable> initially set to <start-expression> and incrementing in <increment> steps (default is 1). The statements between FOR and NEXT are repeated until the variable value reaches or steps over <target-expression>.

SEE ALSO: NEXT

EXAMPLE:

```
FOR n=2 TO 0 STEP -1
          PRINT n,
        NEXT n
        RESULT: 2  1  0
```

Function: FORK()

Syntax: `<int-result>=FORK()`

DESCRIPTION:

FORK() creates a child process of the running task (usually the X11-Basic interpreter with the Basic program) that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0.

On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a -1 will be returned in the parent's context, no child process will be created.

EXAMPLE:

```
a=FORK()
IF a=-1
  PRINT "error"
  QUIT
ELSE IF a=0
  PRINT "I am child"
ELSE
  PRINT "I am parent. My child is PID=";a
ENDIF
```

Function: `FORM_ALERT()`

Syntax: `<num-result>=FORM_ALERT(<default-button>,<string$>)`

DESCRIPTION:

Creates an alert box. button = number of the default button (0= none). string$ = string defining the message in the alert. Note that the square brackets are part of the string: $[$i$]$$[$Message$]$$[$
where i = the required alert symbol - see ALERT. FORM_ALERT returns the number of the selected Button.

SEE ALSO: `ALERT`

EXAMPLE:

```
~FORM_ALERT(1,"[0][This is my message!][OK]")
```

Function: `FORM_CENTER()`

Syntax: `<num-result>=FORM_CENTER(tree,x,y,w,h)`

DESCRIPTION:

Centers the tree, and returns its coordinates. INPUT: tree - address of the object tree. OUTPUTS: x,y coordinates of top left corner w,h form width and height. returns a reserved value.

Function: FORM_DIAL()

Syntax: `<num-result>=FORM_DIAL(flag,x1,y1,w1,h1,x2,y2,w2,h2)`

DESCRIPTION:

Release (or reserve) a rectangular screen area and draw an expanding/shrinking rectangle. Returns 0 if an error occured. flag function 0 reserve a display area. 1 draw expanding box. 2 draw shrinking box. 3 release reserved display area. x1,y1 top left corner of rectangle at min size w1,h1 width & height " " " " " x2,y2 top left corner of rectangle at max size w2,h2 width & height " " " " "

Function: `FORM_DO()`

Syntax: `<num-result>=FORM_DO(tree,obj)`

DESCRIPTION:

Manages an Object tree, interacts with the user until an object with EXIT or TOUCH EXIT status is clicked on. Returns the number of the object whose clicking or double clicking caused the function to end. If it was a double click, bit 15 will be set. tree = address of the object tree. obj = Number of the first editable field (if there is one).

Function: `FRAC()`

Syntax: `<num-result> = FRAC(<num-expression>)`

DESCRIPTION:

FRAC() returns the fractional part of its argument.

SEE ALSO: `INT()`, `CINT()`, `TRUNC()`, `ROUND()`

EXAMPLE:

```
PRINT FRAC(-1.234)
Result: -0.234
```

Command: FREE

Syntax: `FREE <adr>`

DESCRIPTION:

Frees a previously allocated Memory block.

SEE ALSO: `MALLOC()`

a=FREEFILE() – Returns first free filenumber or -1 on error.

Command: `FULLW`

Syntax: `FULLW [[#]n]`

DESCRIPTION:

Enlarges window ’n’ to full screen size. ’n’ is the window number.

SEE ALSO: `OPENW`, `CLOSEW`, `MOVEW`, `SIZEW`, `TOPW`, `BOTTOMW`

Command: FUNCTION

Syntax: `FUNCTION <name>[$][(<expression> [, ...])]`

DESCRIPTION:

FUNCTION starts a user-defined multi-line function that calculates and returns a value from an optional list of parameters. The FUNCTION is called by using the function name preceeded by a @ in an expression. The function return type can either be a numerical value or a string. In the latter case, the function name must end with the "$" precision qualifier. A FUNCTION returns a result with the RETURN command inside the function. In a function, RETURN can be used several times, with IF or the like. A function cannot be terminated without a RETURN command being before the ENDFUNC command. In a function name ending with the $ character the function returns a string result.

All variables declared inside the FUNCTION block are global variables unless you declare them as local with the LOCAL command. The FUNCTION name may be followed by a list of parameter variables representing the values and variables in the calling line. Variables in the calling line reach the FUNCTION "by-value" unless the VAR keyword is used in the calling line. In that case, the variable is passed "by-reference" to the FUNCTION so that the FUNCTION "gets" the variable and not only its value. Variables passed "by-reference" can be changed by the FUNCTION. The FUNCTION block is terminated by an ENDFUNCTION statement which resumes execution of the calling expression. Unlike a PROCEDURE-subroutine, a FUNCTION must return a value.

SEE ALSO: ENDFUNCTION, RETURN, DEFFN, LOCAL, PROCEDURE

EXAMPLE:

```
FUNCTION theta(x,a)
  if x>a
    RETURN 0
  else
    RETURN a
  endif
ENDFUNCTION
```

5.8 G

Function: `GASDEV()`

Syntax: `<num-result>=GASDEV(<num-expression>)`

DESCRIPTION:

Returns a random number which is gauss distributed. The numbers range from minus infinity to infinity but values around 0 are much more likely. The argument is taken as a seed for the random generator.

SEE ALSO: `RND()`

Command: GET

Syntax : GET <x1>,<y1>,<x2>,<y2>,<var$>

DESCRIPTION:

GET puts a section of the graphic window into a string variable (x1,y1 and x2,y2 are coordinates of diagonally opposite corners).

SEE ALSO: PUT

Function: `GET_COLOR()`

Syntax: `<num-result>=GET_COLOR(<red-value>,<green-value>,<blue-value>)`

DESCRIPTION:

GET_COLOR() returns acolor number for the specified color. The rgb-values range from 0 (dark) to 65535 (bright). The returned number depends on the screen depth of the X-Server. For 8 bit a color cell is allocated or if there is no free cell, a color is chosen which is most similar to the specified. The color numbers may be passed to the COLOR command.

EXAMPLE:

```
yellow=GET_COLOR(65535,65535,0)
COLOR yellow
```

SEE ALSO: `COLOR`

Function: GLOB()

Syntax: `<bool>=GLOB(name$,pattern$[,flags])`

DESCRIPTION:

GLOB() checks if name$ matches the wildcard pattern pattern$ and gives -1 (TRUE), else 0 (FALSE). The kind of check can be secified with the flags parameter.

flags 0 – default, no extras 1 – name$ is treated as a filename (Chars '/' are not matched) 2 – Backslashes quote special characters 4 – special treatment of '.' 8 – just check path of file name name$ 16 – case insensitive

EXAMPLES:

```
glob("abcd","abc?")         Result: -1
glob("abcd","*")                    -1
glob("abc","ab??")                   0
glob("*a[0-9]*","sad33333")          0
```

SEE ALSO: INSTR(), WORT_SEP

Command: GOSUB ABBREV. @

Syntax: `GOSUB <preocedure-name>[(<parameterlist>)]`

DESCRIPTION:

GOSUB initiates a jump to the preocedure specified after GOSUB. The code reached that way must end with a RETURN statement which returns control to the calling line.

A procedure name can begin with a digit and contain letters, numbers, dots and the underline dash. <parameterlist> contains expressions which are passed by value to local variables to the procedure. It is possible to call further procedures whilst in a procedure. It is even possible to call the procedure one is in at the time (recursive call).

EXAMPLES:

```
GOSUB testproc
@calcvac(12,s,4,t$)
```

SEE ALSO: PROCEDURE, RETURN, SPAWN, GOTO, EVERY, AFTER

Command: GOTO

Syntax: `GOTO <label-name>`

DESCRIPTION:

Allows an unconditional jump to a label. A label must be defined at the beginning of a line and must end in a colon.

EXAMPLE:

```
GOTO here
PRINT "never"
here:
PRINT "ever"
```

SEE ALSO: GOSUB

Command: `GPRINT`

Syntax: `GPRINT [[AT(),TAB(),SPC()]a$b`|const|USING|...;',]|

DESCRIPTION:

The GPRINT-statement writes all its arguments to the graphic window. It uses the same sytax as PRINT. Unlike PRINT thoe output goes to the graphic window.

SEE ALSO: `PRINT`, `TEXT`

Command: `GRAPHMODE`

Syntax: `GRAPHMODE <n>`

DESCRIPTION:

Sets the graphic mode:
<n>=0 default <n>=1 replace <n>=2 transparent <n>=3 xor <n>=4 reverse transparent

Function: `GRAY()`

Syntax: `<num-result>=GRAY(<num-expression>)`

DESCRIPTION:

This function calculates the Gray-code of a given positive integer number. If the number is negative, the inverse Graycode is calculated.

EXAMPLE:

```
PRINT GRAY(34)
       Result: 51
```

5.9 H

Command: HELP

Syntax: `HELP <string-pattern>`

DESCRIPTION:

Gives information of built in commands and functions.

EXAMPLE:

```
HELP CL*

Result:

CLEAR [,...]
CLEARW [,i%]
CLOSE [,...]
CLOSEW [,i%]
CLR [,...]
CLS
```

Function: `HEX$()`

Syntax: `<string-result>=HEX$(<x>[,<n>])`

DESCRIPTION:

Changes the value of <x> into a string expression which contains the value in hexadecimal form. The optional parameter <n> specifies the number of characters to be used.

SEE ALSO: `STR\$(), BIN\$()`

Command: `HIDEM`

Syntax: `HIDEM`

DESCRIPTION:

Switches off the mouse pointer.

SEE ALSO: `SHOWM`, `DEFMOUSE`

Command: `HOME`

Syntax: `HOME`

DESCRIPTION:

moves text cursor home. (upper left corner)

SEE ALSO: `PRINT AT()`

Command: `HTAB`

Syntax: `HTAB <num-expression>`

DESCRIPTION:

Positions the text cursor to the specified column.

SEE ALSO: `PRINT AT(),VTAB`

Function: `HYPOT()`

Syntax: `<num-result>=HYPOT(<num-expression:x>,<num-expression:y>)`

DESCRIPTION:

The hypot() function returns the sqrt(x*x+y*y). This is the length of the hypotenuse of a right-angle triangle with sides of length x and y, or the distance of the point (x,y) from the origin.

SEE ALSO: `SQRT()`

EXAMPLE:

```
PRINT HYPOT(3,4)    Result: 5
```

5.10 I

Command: `IF`

Syntax: `IF <condition> [... ELSE [IF <expression> ...] ... ENDIF`

DESCRIPTION:

Divides a program up into different blocks depending on how it relates to the ’condition’.

SEE ALSO: `ELSE`, `ENDIF`

Operator: `IMP`

Syntax: `<num-result>=<num-expression> IMP <num-expression>`

DESCRIPTION:

The operator IMP (implication) corresponds to a logical consequence. The result is FALSE if a FALSE expression follows a TRUE one. The sequence of the argument is important.

Table: A | B | A IMP B —–+—–+———– -1 | -1 | -1 -1 | 0 | 0 0 | -1 | -1 0 | 0 | -1

SEE ALSO: `TRUE`, `FALSE`, `NOT`, `XOR`, `EQV`

*

Function: `IMP()`

Syntax: `<num-result>=IMP(<num-expression>,<num-expression>)`

DESCRIPTION:

Binary Function of logical operator IMP.

SEE ALSO: `IMP`

EXAMPLE:

```
        PRINT BIN$(IMP(13,14),4)
Result:  1110
```

Command: `INC`

Syntax: `INC <num-variable>`

DESCRIPTION:

INC increments a (numeric) variable. This command is considerably faster then the equivalent statement "<variable> = <variable> + 1".

SEE ALSO: `ADD`, `DEC`

Command: `INFOW`

Syntax: `INFOW [<window-nr>],<string-expression>`

DESCRIPTION:

Links the (new) information string to the window with the number. On UNIX this Information will be displayed in ICONIFIED state of the window.

SEE ALSO: `TITLEW`

Variable: INKEY$

Syntax: `<string-result>=INKEY$`

DESCRIPTION:

Returns a string containing the ASCII characters of all keys which have been pressed on the keyboard.

EXAMPLE:

```
REPEAT ! Wait until a
UNTIL LEN(INKEY$) ! Key was pressed
```

SEE ALSO: INP(), KEYEVENT

Function: INLINE$()

Syntax: `<string-result>=INLINE$(<string-expression>)`

DESCRIPTION:

7Bit-ASCII to Binary conversion. This command basically does a RADIX conversion (from 64 to 256) on the contents of the string. This is intended to be used to include binary data into the source code of a basic program.

The inverse coding (from binary to 7 bit ASCII) is done by the program inline.bas which comes with X11-Basic.

EXAMPLE:

```
        sym$=INLINE$("$$$$$$$$0$&Tc_>$QL&ZD3cccccK]UD<*%D$$$$$$$$$") ! Train
PUT_BITMAP sym$,92,92,16,16
```

SEE ALSO: PUT_BITMAP

Function: INP\(), INP\\%(), INP\\&()

Syntax:

```
<num-result>=INP(<channel-nr>)
<num-result>=INP\&(<channel-nr>)
<num-result>=INP\%(<channel-nr>)
```

DESCRIPTION:

Reads one byte from a file previously opened with OPEN (nr>0) or from the standart files (-1=stderr, -2=stdin, -4=stdout). INP&() reads a word (2 Bytes) and INP%() reads a long word (4 bytes).

EXAMPLE:

```
~INP(-2)        ! Waits for a key beeing pressed
PRINT INP%(#1)  ! reads a long from a previously opened file
```

SEE ALSO: OUT, INPUT\$()

Function: `INP?()`

Syntax: `<bool-result>=INP?(<channel-nr>)`

DESCRIPTION:

Determine the input status of the device. TRUE(-1) is device is ready (chars can be read) ortherwise FALSE(0).

SEE ALSO: `INP()`

Function: `INV()`

Syntax: `b()=INV(a())`

DESCRIPTION:

Calculate the inverse of a square matrix a(). The calculation is done using the singular value decomposition. If the matrix is singular the algorithm tells you how many singular values are zero or close to zero.

SEE ALSO: `SOLVE(), DET()`

Command: INPUT

Syntax: `INPUT [#<device-number>]`<prompt-expression>], <variable> [, ...]|

DESCRIPTION:

INPUT gets comma-delimited input from the standart input or from a previously opened file as specified by <device-number> (use the LINE INPUT function do read complete lines from a file and BLOAD to load complete files). Any input is assigned to the variable(s) specified. If input is expected from a console window, then <prompt-expression> is printed to the console window to request input from the user.

SEE ALSO: `LINEINPUT`, `FORM INPUT AS`, `PRINT`

EXAMPLE:

```
INPUT #1,a$
INPUT "Enter your name:",a$
```

Function: INPUT$()

Syntax:
```
<string-result>=INPUT$(#<nr>,<len>)
<string-result>=INPUT$(<len>)
```

DESCRIPTION:

Reads <len> characters from the keyboard and assigns them to a string. Optionally, if the device-number is specified, the characters are read in from a previously OPENed channel <nr>.

SEE ALSO: `INPUT`, `INP()`

Function: `INSTR()`

Syntax: `<num-result>=INSTR(<a$>,<b$>[,<n>])`

DESCRIPTION:

Searches to see if b$ is present in a$ and returns its position. <n> is a numeric expression indicating the position in a$ at which the search is to begin (default=1). If <n> is not given the search begins at the first character of a$. If b$ is found in a$ the start position is returned, otherwise 0.

SEE ALSO: `RINSTR()`, `GLOB()`

Function: INT()

Syntax: `<num-result> = INT(<num-expression>)`

DESCRIPTION:

INT() returns the largest integer number smaller than or equal to its argument.

SEE ALSO: CINT(), FRAC(), TRUNC(), ROUND()

EXAMPLE:

```
PRINT INT(1.4), INT(-1.7)
       Result: 1, -2
```

Function: IOCTL()

Syntax: `<num-result> = IOCTL(#n,d%[,adr%])`

DESCRIPTION:

IOCTL() manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with ioctl requests. The argument #n must refer to an open file.

The second argument is a device-dependent request code. The third argument is an untyped pointer to memory.

An ioctl request has encoded in it whether the argument is an in param- eter or out parameter, and the size of the argument adr% refers to in bytes.

Usually, on success zero is returned. A few ioctls use the return value as an output parameter and return a nonnegative value on success. On error, -1 is returned, and errno is set appropriately.

SEE ALSO: OPEN, CLOSE

EXAMPLE:

```
OPEN "U",#1,"/dev/console"
frequency=300
tone=1190000/frequency
KIOCSOUND=19247
PRINT ioctl(#1,KIOCSOUND,tone)  ! Sounds the speaker
CLOSE #1
Result: 0
```

5.11 J

Function: `JULDATE$()`

Syntax: `d$=juldate$(a)`

DESCRIPTION:

Returns the date as string (see DATE$) given by the julian day number a.

SEE ALSO: `JULIAN(), DATE\$`

Function: `JULIAN()`

Syntax: `a=JULIAN(date$)`

DESCRIPTION:

Returns the julian date corresponding to the date given as a string in standart format. The number which is returned is an integer number and has the unit days.

EXAMPLE:

```
PRINT "Number of days since Sept. 11 2001: ";julian(date$)-julian("11.09.2001")
```

SEE ALSO: `JULDATE\$()`, `DATE\$`

5.12 K

Command: `KEYEVENT`

Syntax: `KEYEVENT kc,ks[,t$,k,x,y,xroot,yroot]`

DESCRIPTION:

Waits until Key is pressed. (graphic window) After the key event has occured, the variables have following meaning:

kc – Key-code ks – state of Shift/Control/Alt etc. t$ – correspondig character x – x coordinate of mouse pointer relative to window y – y coordinate xroot – x coordinate of mouse pointer relative to screen yroot – y coordinate k – mouse button state

SEE ALSO: `MOUSEEVENT`

5.13 L

Function: LEFT$()

Syntax: `<string-result> = LEFT$(<string-expression> [,<characters>])`

DESCRIPTION:

LEFT$() returns the specified number of characters from its argument, beginning at its left side. If the number of <characters> is not specified then LEFT$() returns only the leftmost character.

SEE ALSO: `RIGHT\$(), MID\$()`

EXAMPLE:

```
PRINT LEFT$("Hello",1)
Result: H
```

Function: LEN()

Syntax: `l=LEN(t$)`

DESCRIPTION:

Returns the length of a string.

EXAMPLE:

```
PRINT LEN("Hello")
Result: 5
```

Command: `LET`

Syntax: `LET <variable> = <expression>`

DESCRIPTION:

LET assigns the value of <expression> to <variable>. The interpreter also supports implicit assignments, ie. the LET keyword before an assignment may be omitted. This works because the first equal sign is regarded as assignment operator.

EXAMPLE:

```
LET N=1
```

Command: `LINE`

Syntax: `LINE <x1>,<y1>,<x2>,<y2>`

DESCRIPTION:

Draws a straight line from (x1,y1) to (x2,y2).

SEE ALSO: `DRAW`, `PLOT`

Command: `LINEINPUT`

Syntax: `LINEINPUT [[#]<device-number>,] <string-variable>`

DESCRIPTION:

LINE INPUT reads an entire line from atandart input or from a previously opened file as specified by <device-number> (to load a complete file, use BLOAD). Unlike the regular INPUT command, LINEINPUT does not stop at delimiters (commas).

SEE ALSO: `INPUT`

Command: `LINK`

Syntax: `LINK #<device-nr>,<string-expression:name>`

DESCRIPTION:

LINK linkes a shared object file/library (*.so in /var/lib) dynamically. It will from now on be addressed via the device-nr.

The adresses of he symbols of that library can be read with the SYM_ADR() function.

If the Library is not used any more it can be unlinked with the UNLINK command.

SEE ALSO: `UNLINK`, `SYM\_ADR()`, `CALL`

Command: `LIST`

Syntax: `LIST [<line-number>[,<line-number>]`

DESCRIPTION:

LIST displays the source code or a code segment. Note that the line number of the first line in a file is 1, that the second line is line 2 etc.

SEE ALSO: `LLIST`, `PLIST`, `PRG\$()`

EXAMPLE:

```
LIST
LIST 1-10
LIST 5
```

Command: `LOAD`

Syntax: `LOAD <string-expression:name>`

DESCRIPTION:

Loads a program into memory.

SEE ALSO: `XLOAD`, `MERGE`, `CHAIN`

EXAMPLE:

```
LOAD "testme.bas"
```

Function: `LOC()`

Syntax: `<int-result>=LOC(#<device-nre>)`

DESCRIPTION:

Returns the location of the file pointer for the file with the device number. The location is given in number of bytes from the start of the file.

SEE ALSO: `LOF()`

Command: `LOCAL`

Syntax: `LOCAL <var>[,<var>,...]`

DESCRIPTION:

Declares several variables to be a local variable.

EXAMPLE:

```
LOCAL a,b$,s()
```

Command: `LOCATE`

Syntax: `LOCATE <row>,<column>`

DESCRIPTION:

Positions the cursor to the specified location.

SEE ALSO: `PRINT AT()`, `CRSLIN`, `CRSCOL`

Function: `LOF()`

Syntax: `<int-result>=LOF(#<device-nre>)`

DESCRIPTION:

Returns length of file with device number.

SEE ALSO: `LOC()`

Function: `LOG(), LOG10(), LN()`

Syntax:

```
<num-result>=LOG(<num-expression>)
<num-result>=LOG10(<num-expression>)
<num-result>=LN(<num-expression>)
```

DESCRIPTION:

Returns the natural logarithm (log, ln) or the logarithm base 10 (log10).

SEE ALSO: `EXP()`

Function: `LOGB()`

Syntax: `<int-result>=LOGB(<num-expression>)`

DESCRIPTION:

Returns the logarithm base 2 in interger values.

SEE ALSO: `LOG()`

Function: `LOG1P()`

Syntax: `<num-result>=LOG1P(<num-expression>)`

DESCRIPTION:

Returns a value equivalent to log(1+x). It is computed in a way that is accurate even if the value of x is near zero.

SEE ALSO: `LOG(),EXP(),LN()`

Command: `LOOP`

Syntax: `LOOP`

DESCRIPTION:

LOOP terminates a DO loop and can be used as unqualified loop terminator (such a loop can only be aborted with the EXIT command). Execution continues with the DO line.

SEE ALSO: `DO`, `EXIT IF`, `BREAK`

Function: `LPEEK()`

Syntax: `<int-result>=LPEEK(<num-expression>)`

DESCRIPTION:

Reads a 4 byte integer from address.

SEE ALSO: `PEEK()`, `POKE`

Command: `LPOKE`

Syntax: `LPOKE <adr>,<num-expression>`

DESCRIPTION:

Writes a 4 byte integer to address <adr>.

SEE ALSO: `DPOKE`, `POKE`, `PEEK()`

```
LSET t$=a$
```

Command: `LTEXT`

Syntax: `LTEXT x,y,t$`

DESCRIPTION:

Draws a Text at position x,y. The LTEXT command uses a Linegraphic-Text, which allows the user to draw very large Fonts and be independant of the system fonts. The Font Style can be influenced with the DEFLINE and the DEFTEXT command.

SEE ALSO: `DEFTEXT`, `TEXT`, `DEFLINE`

5.14 M

Function: `MALLOC()`

Syntax: `<int-result:adr>=MALLOC(<num-expression:size>)`

DESCRIPTION:

Allocates size bytes and returns a pointer to the allocated memory. The memory is not cleared.

SEE ALSO: `FREE()`, `MFREE()`, `REALLOC()`

Function: `MAX()`

Syntax:

```
m=MAX(a,b[,c,...])
m=MAX(f())
```

DESCRIPTION:

Returns the largest value out of the list of arguments or the largest value of an array.

SEE ALSO: `MIN()`

Command: MENU

Syntax: `MENU`

DESCRIPTION:

Performs menu check and action. This command handles EVENTs. Prior to use, the required action should be specified with a MENUDEF command. For constant supervision of events, MENU is usually found in a loop.

EXAMPLE:

```
MENUDEF field$(),menuaction
DO
  pause 0.05
  MENU
LOOP
PROCEDURE menuaction(k)
   ...
RETURN
```

SEE ALSO: MENUDEF

Command: MENUDEF

Syntax: `MENUDEF array$(),<procname>`

DESCRIPTION:

This command reads text for menu-header from `array\$()` the string-array contains the text for menu-titles and menu-entrys

- end of row: empty string "" - end of menu-text: empty string ""

`<procname>` The procedure to which control will be passed on selection of a menu entry is determined.

`<procname>` is a procedure with one parameter which is the number of the selected item to call when item was selected.

EXAMPLE:

```
field$()=["INFO","  Menutest  ","","FILE","  new","  open ...","  save","\
save as ...","--------------","  print","--------------","  Quit","",""]
MENUDEF field$(),menuaction
DO
  pause 0.05
  MENU
LOOP
PROCEDURE menuaction(k)
  PRINT "MENU selected ";k;" contents: ";field$(k)
  IF field$(k)="  Quit"
    QUIT
  ENDIF
RETURN
```

SEE ALSO: `MENU`, `MENUSET`, `MENUKILL`

Command: `MENUSET`

Syntax: `MENUSET n,x`

DESCRIPTION:

Change apperance of menu-entry n with value x.

x=0 ' ' normal, reset marker '(HOCH)' x=1 '(HOCH)' set marker x=2 '=' set menu-point non selectable x=3 ' ' set menue-point selectable x=4 check the menu entry '-' permanent non selectable

SEE ALSO: `MENU`

Command: MENUKILL

Syntax: `MENUKILL`

DESCRIPTION:

Erases the menu, which prior has been defined with MENUDEF.

SEE ALSO: `MENUDEF`

Command: `MERGE`

Syntax: `MERGE <filename>`

DESCRIPTION:

MERGE appends a BASIC program to the program currently in memory. Program execution is not interrupted. This command typically is used to append often-used subroutines at run-time.

SEE ALSO: `CHAIN`, `LOAD`

EXAMPLE:

```
MERGE "examples/hello.basic"
```

Function: `MID$()`

Syntax: `m$=MID$(t$,s[,l])`

DESCRIPTION:

Returns l characters in a string from the positon s of the string t $.Ifsislargerthanthelengthoft$, then an empty string is returned. If l is omitted, then the function returns only one character of the string from position x.

SEE ALSO: `LEFT\$(),RIGHT\$()`

Function: MIN()

Syntax:
```
m=MIN(a,b[,c,...])
m=MIN(f())
```

DESCRIPTION:

Returns the smallest value out of the list of arguments or the smallest value of an array.

SEE ALSO: `MAX()`

Function: MKI$(), MKL$(), MKS$(), MKF$(), M

Syntax:

```
<string-result>=MKI$(<num-expression>)
<string-result>=MKL$(<num-expression>)
<string-result>=MKS$(<num-expression>)
<string-result>=MKF$(<num-expression>)
<string-result>=MKD$(<num-expression>)
<string-result>=MKA$(<array-expression>)
```

DESCRIPTION:

Transforms a number into a character string.
MKI$ 16-bit number into a 2-byte string.
MKL$ 32-bit number into a 4-byte string.
MKS$ a number into a 4-byte float format.
MKF$ same as MKS$().
MKD$ a number into a 8-byte double float format. MKA$() transforms a whole Array into a string. It can be beacktransformed with CVA().

SEE ALSO: `CVI(), CVF(), CVL(), CVA(), CVS(), CVD()`

Operator: `MOD`

Syntax: `<num-result>=<num-expression:x> MOD <num-expression:y>`

DESCRIPTION:

Produces the remainder of the division of x by y.

SEE ALSO: `DIV`, `MOD()`

*

Function: `MOD()`

Syntax: `<num-result>=MOD(<num-expression:x>,<num-expression:y>)`

DESCRIPTION:

Produces the remainder of the division of x by y.

SEE ALSO: `DIV`, `MOD`

Command: `MOUSE`

Syntax: `MOUSE x,y,k`

DESCRIPTION:

Determines the mouse position (x,y) relative to the origin of the graphics window and the status of the mouse buttons (k). k=0 no buttons pressed
k=1 left button
k=2 middle button
k=4 right buttons
or any combinations.

SEE ALSO: `MOUSEX`, `MOUSEY`, `MOUSEK`

Command: MOUSEEVENT

Syntax: `MOUSEEVENT x,[y,k,xroot,yroot,s]`

DESCRIPTION:

Waits until a mouse button is pressed (graphic window). Returns Mouse coorinate (x,y) relative to window, mouse coorinate (xroot,yroot) relative to screen, mouse button and state of the Alt/Shift/Caps keys.

SEE ALSO: `MOUSE`, `MOUSEX`, `MOUSEY`, `MOUSEK`, `KEYEVENT`

Variable: MOUSEX, MOUSEY, MOUSEK, MOUSES

Syntax:
```
<int-result:x>=MOUSEX
<int-result:y>=MOUSEY
<int-result:k>=MOUSEK
<int-result:s>=MOUSES
```

DESCRIPTION:

Determines the mouse position (x,y), the status of the mouse buttons (k) and the status of the Shift and Control keys (s): k=0 no buttons pressed
k=1 left button
k=2 middle button
k=4 right buttons
or any combinations.

s=0 no Keys
s=1 Shift
s=2 CapsLock
s=4 Control
s=8 Alt
s=16 NumLock
s=64 Windows-Key
s=128 ScrollLock
or any combination.

SEE ALSO: MOUSE, SETMOUSE, MOUSEEVENT

Command: `MOTIONEVENT`

Syntax: `MOTIONEVENT x,y,xrrot,yroot,s`

DESCRIPTION:

Waits until the mouse has been moved. (graphic window). Returns new mouse coorinate (x,y) relative to window, mouse coorinate (xroot,yroot) relative to screen and state of the Alt/Shift/Caps keys.

SEE ALSO: `MOUSE`, `MOUSEX`, `MOUSEY`, `MOUSEK`, `MOUSEEVENT`

Command: `MOVEW`

Syntax: `MOVEW n,x,y`

DESCRIPTION:

Moves Wondow n to absolute screen position x,y

SEE ALSO: `OPENW`, `SIZEW`, `TITLEW`

Function: `MTFD$()`

Syntax: `b$=MTFD$(a$)`

DESCRIPTION:

This function performs a Move To Front decoding function on an input string. The MTF decoder keeps an array of 256 characters in the order that they have appeared. Each time the encoder sends a number, the decoder uses it to look up a character in the corresponding position of the array, and outputs it. That character is then moved up to position 0 in the array, and all the in-between characters are moved down a spot.

SEE ALSO: `MTFE\$()`

Function: `MTFE$()`

Syntax: `b$=MTFE$(a$)`

DESCRIPTION:

This function performs a Move To Front encoding function on an input string. An MTF encoder encodes each character using the count of distinct previous characters seen since the characters last appearance. This is implemented by keeping an array of characters. Each new input character is encoded with its current position in the array. The character is then moved to position 0 in the array, and all the higher order characters are moved down by one position to make roon.

SEE ALSO: `MTFD\$()`

Command: `MUL`

Syntax: `MUL <num-var>,<num-expression>`

DESCRIPTION:

Same as var=var*n but faster.

SEE ALSO: `ADD`, `SUB`, `MUL()`, `DIV`

*

Function: `MUL()`

Syntax: `<num-result>=MUL(<num-expression>,<num-expression>)`

DESCRIPTION:

Returns product of two numbers.

SEE ALSO: `ADD()`, `SUB()`, `MUL`, `DIV()`

5.15 N

Command: `NEW`

Syntax: `NEW`

DESCRIPTION:

NEW erases the program and all variables in memory (and stops execution of program.)

SEE ALSO: `CLEAR`

Command: `NEXT`

Syntax: `NEXT [<variable>]`

DESCRIPTION:

NEXT terminates a FOR loop. FOR loops must be nested correctly: The variable name after NEXT is for looks only and can not be used to select a FOR statement. Each NEXT jumps to the matching FOR statement regardless if and what <variable> is specified after NEXT.

SEE ALSO: `FOR`

EXAMPLE:

```
FOR n=1 TO 2
    FOR m=10 to 11
        PRINT "n=";n,"m=";m
    NEXT m
NEXT n
```

Command: `NOP, NOOP`

Syntax:

```
NOP
NOOP
```

DESCRIPTION:

No Operation: do nothing.

5.16 O

Command: `OBJC_ADD`

Syntax: `OBJC_ADD tree,parent,child`

DESCRIPTION:

Adds an object to a given tree and pointers between the existing objects and the new object are created. tree address of the object tree parent object number of the parent object child object number of the child to be added.

SEE ALSO: `OBJC\_DELETE`

Command: OBJC_DELETE

Syntax: `OBJC_DELETE tree,object`

DESCRIPTION:

An object is deleted from an object tree by removing the pointers. The object is still there and can be restored by repairing the pointers.

tree address of the object tree object Object number of the object to delete.

SEE ALSO: `OBJC\_ADD`

Function: OBJC_DRAW()

Syntax: `ret=objc_draw(tree,startob,depth,cx,cy,cw,ch)`

DESCRIPTION:

Draws any object or objects in an object tree.

Each OBJC_DRAW call defines a new clip rectangle, to which the drawing is limited for that call.

Returns 0 on error. tree address of the object tree startob number of the first object to be drawn depth Number of object levels to be drawn cx,cy coordinates of top left corner of clipping rectangle cw,ch width & height of clipping rectangle

SEE ALSO: `OBJC\_FIND()`

Function: `OBJC_FIND()`

Syntax: `idx=objc_find(tree,startob,depth,x,y)`

DESCRIPTION:

Finds an object under a specific screen coordinate. (These may be the mouse coordinates.)

The application supplies a pointer to the object tree, the index to the start object to search from, the x- and y-coordinates of the mouse's position, as well as a parameter that tells OBJC_FIND how far downthe tree to search (depth).

This Fucntion returns the index of the found Object or -1 in case no object could be found.

SEE ALSO: `OBJC\_DRAW()`

Function: `OBJC_OFFSET()`

Syntax: `ret=objc_offset(tree,obj,x,y)`

DESCRIPTION:

Calculates the absolute screen coordinates of the specified object in a specified tree. Returns 0 on error. tree address of the object tree obj object number x,y returns the x,y coordinates to these variables.

SEE ALSO: `OBJC\_FIND()`

o$=OCT$(d%,n%) ON * GOSUB proc1$[$,proc2,...$]$

Command: `ON BREAK`

Syntax:
```
ON BREAK CONT
ON BREAK GOSUB <procedure>
ON BREAK GOTO <label>
```

DESCRIPTION:

ON BREAK installs a subroutine that gets called when the BREAK condition (normally CTRL-c) occurs. ON BREAK CONT causes the program to continue in any case. ON BREAK GOTO jumps to a specified label.

SEE ALSO: `GOTO`, `ON ERROR`

Command: `ON ERROR`

Syntax:

```
ON ERROR CONT
ON ERROR GOSUB <procedure>
ON ERROR GOTO <label>
```

DESCRIPTION:

ON ERROR installs an error handling subroutine that gets called when the next error occurs. ON ERROR CONT will ignore any error.

SEE ALSO: `GOSUB`, `ERROR`

Command: OPEN

Syntax: `OPEN <mode>,<device-number>,<filename>[,<port-value>]`

DESCRIPTION:

OPEN opens the specified file for reading or writing or both. The <device-number> is the number you want to assign to the file (functions that read from files or write to files expect to be given this number). The device number must be between 0 and 99 in the current implementation of X11-Basic. When you close a file, the device number is released and can be used again in subsequent OPEN statements.

<mode> is a character string which indicates the way the File should be opened. The first character of that String may be "O", "I", "U" or "A". These characters correspont to the mode for which the file is opened: "I" – INPUT, "O" –OUTPUT, "A" – APPEND and "U" – UNSPECIFIED. Open a file for INPUT if you want to read data from the file. If you open a file for OUTPUT, you can write to the file. However, all data that was stored in the file (if the file already exists) is lost. If you want to write new data to a file while keeping the existing content, open the file for appending to it, using the APPEND mode. When you open a file using the RANDOM ("U") keyword, you can both read from the file and write to the file at arbitrary positions. You can, for example, seek a position in the middle of the file and start appending new lines of text. All file modes but INPUT create the file if it does not exist. OPEN "I" fails if the file does not exist (use the EXIST() function before OPEN to be sure that the file exists).

The second character specifies the type of File which should be opened or created: "" default opens regular file "U" opens a datagramm socket connection "C" opens a Strem socket as client with connection "S" opens a stream socket as server "A" Socket accept connection "X" extra settings for a special device following: (e.g. speed and parity of transmission via serial ports) "UX:baud,parity,bits,stopbits,flow"

<port-nr> The portnr is used only by the OPEN "UC" and OPEN "UU" statement. It determines the TCP/IP Port of connection (FTP, WWW, TELNET, MAIL etc.).

SEE ALSO: `CLOSE`, `EXIST()`, `INPUT`, `LINEINPUT`, `PRINT`, `SEEK`, `LOF()`, `EOF()`, LOC(), BLOAD, LINK, FREEFILE(), CONNECT

EXAMPLES:

```
OPEN "I",#1,"data.dat"      ---- opens file "data.dat" for input
OPEN "UC",#1,"localhost",80 ---- opens port 80 of localhost for read and#
```

```
 write
OPEN "UX:9600,N,8,1,XON,CTS,DTR",#1,"/dev/ttyS1"
                 ---- open COM2 for input and output with 9600:8:N:1 with
      software flow control and hardware flow control and also
      drop DTR line and raise it again.
```

Command: `OPENW`

Syntax: `OPENW n`

DESCRIPTION:

Opens a graphic window. There can be up to 16 graphic windows opened. All graphic output goes to the window which was opened latest. OPENW can be used to switch between multiple windows. Window 1 os opened automatically on default when the first graphic command is executed and no other window is already opened.

SEE ALSO: `CLOSEW`, `MOVEW`, `SIZEW`, `TITLEW`, `ROOTWINDOW`

Command: `OUT`

Syntax: `OUT #n,a`

DESCRIPTION:

Writes a byte a to an open channel n.

SEE ALSO: `PRINT`, `INP()`

5.17 P

Command: `PAUSE`

Syntax: `PAUSE <sec>`

DESCRIPTION:

pauses <sec> seconds. The resolution of this command is microseconds (in theory).

Command: `PBOX`

Syntax: `PBOX x1,y1,x2,y2`

DESCRIPTION:

Draws a filled box.

SEE ALSO: `BOX`, `RBOX`, `DEFFILL`, `COLOR`

Command: `PCIRCLE`

Syntax: `PCIRCLE x,y,r[,a1,a2]`

DESCRIPTION:

Draws a filled circle (sector).

SEE ALSO: `CIRCLE`, `DEFFILL`, `COLOR`

Function: PEEK()

Syntax: `<int-result>=PEEK(<address>)`

DESCRIPTION:

PEEK() reads a byte from an address in memory. The following example dumps a section of the internal memory near a string t$.

EXAMPLE:

```
t$="Hallo, this is a string..."
i=varptr(t$)-2000
DO
  PRINT "$";HEX$(i,8,8)'
  FOR iu=0 TO 15
    PRINT HEX$(PEEK(i+iu) and 255,2,2)'
  NEXT iu
  PRINT '
  FOR iu=0 TO 15
    a=PEEK(i+iu)
    IF a>31
      PRINT CHR$(a);
    ELSE
      PRINT ".";
    ENDIF
  NEXT iu
  PRINT
  ADD i,16
LOOP
```

SEE ALSO: POKE

Command: PELLIPSE

Syntax: `PELLIPSE x,y,a,b[,a1,a2]`

DESCRIPTION:

Draws a filled ellipse

SEE ALSO: `PCIRCLE`, `ELLIPSE`, `DEFFILL`, `COLOR`

Command: PIPE

Syntax: `PIPE #n1,#n2`

DESCRIPTION:

PIPE links two File channels n1 and n2 together to form a pipe. n1 is for reading, n2 is for writing. Whatever you write to the pipe can be read from it at a different time. The content is buffered in the kernel. The mechanism is FIFI (0 first in first out). The biggest advantage is, that you can read and write to it from different processes (created by FORK()). This allows inter-process communication.

SEE ALSO: `CLOSE`, `OPEN`, `FORK()`

Command: PLIST

Syntax: `PLIST`

DESCRIPTION:

Outputs a formatted listing of the actual program in memory. Alos the internal tokens are printed ans some internal Information. This is intended for internal use only.

EXAMPLE:

```
> PLIST
   0: $00001a |   0,0 |CLS
   1: $000279 |   0,0 |PRINT
   2: $000279 |   0,1 |PRINT " example how to use the ansi color spec."
   3: $000279 |   0,0 |PRINT
   4: $000279 |   0,1 |PRINT "X 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0"
   5: $310240 |  12,1 |FOR U=0 TO 3
   6: $310240 |  11,1 |  FOR J=0 TO 7
   7: $310240 |  10,1 |    FOR I=0 TO 7
   8: $000279 |   0,1 |      PRINT AT(J+6,2*I+2+16*U);CHR$(27)+"["+STR$(U)+";"+STR$(30+I)+";"+STR$(40+J)
   9: $320266 |   7,1 |    NEXT I
  10: $320266 |   6,1 |  NEXT J
  11: $320266 |   5,1 |NEXT U
  12: $000279 |   0,0 |PRINT
  13: $00047f |   0,0 |QUIT
  14: $0008ff |   0,0 |=?=> 2303
```

SEE ALSO: LIST

Command: `PLOT`

Syntax: `PLOT x,y`

DESCRIPTION:

draws point at screen coordinate x,y

SEE ALSO: `LINE`, `POINT()`, `COLOR`

Function: `POINT()`

Syntax: `c=POINT(x,y)`

DESCRIPTION:

returns colour of graphic point x,y in current window

SEE ALSO: `PLOT`

Command: `POKE`

Syntax: `POKE adr%,byte%`

DESCRIPTION:

writes Byte to address adr%

SEE ALSO: `PEEK()`, `DPOKE`, `LPOKE`

Command: `POLYLINE`

Syntax: `POLYLINE n,x(),y()[,x_off,y_off]`

DESCRIPTION:

POLYLINE draws a polygon with n corners. The x,y coordinates for the corner points are given in arrays x() and y(). The optional parameters x_off,y_off can be added to these coordinates.

To draw a closed polygon, the first point hast to be equal to the last point.

SEE ALSO: `LINE`, `DEFLINE`, `COLOR`, `POLYFILL`, `POLYMARK`

Command: `POLYFILL`

Syntax: `POLYFILL n,x(),y()[,x_off,y_off]`

DESCRIPTION:

POLYFILL draws a filled polygon with n corners. The x,y coordinates for the corner points are given in arrays x() and y(). The optional parameters x_off,y_off can be added to these coordinates. POLYFILL fills the polygon with the color and pattern previously defined by COLOR and DEFFILL.

SEE ALSO: `COLOR`, `DEFFILL`, `POLYLINE`, `POLYMARK`

Command: POLYMARK

Syntax: `POLYMARK n,x(),y()[,x_off,y_off]`

DESCRIPTION:

POLYMARK marks the corner points of an invisible polygon with n corners. The x,y coordinates for the corner points are given in arrays x() and y(). The optional parameters x_off,y_off can be added to these coordinates.

POLYMARK marks the points with the shape defined by DEFMARK.

SEE ALSO: `COLOR`, `DEFLINE`, `POLYLINE`, `POLYFILL`

POS PRBOX x1,y1,x2,y2 – filled rounded box t$=PRG$(i) – Programmzeile i (Quelltext)

Command: PRINT ABBREV. ?

Syntax: `PRINT [[AT(),TAB(),SPC(),COLOR()]a$b|const|USING|...;',]|`

DESCRIPTION:

The print-statement writes all its arguments to the screen (standard output); after writing its last argument, print goes to the next line (as in print "Hello ",a$," !"); to avoid this automatic newline, place a semicolon (;) after the last argument (as in print "Please enter your Name:";). To insert a tabulator instead of the automatic newline append a colon (,), e.g. print "Please enter your Name:", . Note that print can be abbreviated with a single question mark (?).

Advanced printing: PRINT AT(), PRINT colour and PRINT USING

PRINT AT

For interactive programs you might want to print output at specific locations. Try the next example: PRINT AT(4,7);"Test"

TAB and SPC PRINT "Hallo";TAB(30);"Test" PRINT "Hallo";SPC(30);"Test"

PRINT COLOR PRINT COLOR(32,2);"Hallo"

The COLOR statement takes up to three arguments. Their meaning is:

Text Mode: Text color: Backgroud color:

0 default setting 30 black 40 black 1 intensive 31 red 41 red 2 dark 32 green 42 green 33 yellow 43 yellow 4 underline 34 blue 44 blue 5 blink 35 magenta 45 magenta 36 cyan 46 cyan 7 reverse 37 white 47 white

PRINT USING

To control the way numbers are printed, use the print using statement: print 12.34 using "###.####" produces 12.3400. The format string ("###.####") consists of hashes (#) with one optional dot and it pictures the appearance of the number to print.

SEE ALSO: `GPRINT`, `STR\$()`, `INPUT`

Command: PROCEDURE

Syntax: `PROCEDURE procname [(p1 [,p2] ... )] * RETURN`

DESCRIPTION:

PROCEDURE starts a user-defined multi-line subroutine which can be executed by the GOSUB command. any number of parameters may be passed to the PROCEDURE via the parameter list. The Variables in that list act like local variables inside the subroutine.

All variables declared inside the PROCEDURE block are global variables unless you declare them as local with the LOCAL command. The PROCEDURE name may be followed by a list of parameter variables representing the values and variables in the calling line. Variables in the calling line reach the PROCEDURE "by-value" unless the VAR keyword is used in the calling line. In that case, the variable is passed "by-reference" to the PROCEDURE so that the PROCEDURE "gets" the variable and not only its value. Variables passed "by-reference" can be changed by the PROCEDURE. The PROCEDURE block is terminated by the RETURN statement which resumes execution of the calling expression. Unlike a FUNCTION-subroutine, a PROCEDURE can not return a value.

SEE ALSO: `GOSUB`, `RETURN`, `LOCAL`, `FUNCTION`

Function: `PTST()`

Syntax: `c=PTST(x,y)`

DESCRIPTION:

returns colour of graphic point x,y in current window. Same as POINT().

SEE ALSO: `POINT()`, `PLOT`

PSAVE a$ – writes the reformatted BASIC-program into file with the name a$ PUT x,y,g$ – displays graphic stored in g$ at postion x,y PUTBACK $[$#n,$]$a – puts character a back to input channel n PUT_BITMAP t$,i,i,i,i

5.18 Q

Command: `QUIT`

Syntax: `QUIT [<return-code>]`

DESCRIPTION:

QUIT exits the interpreter. You may set a <return-code> which will be passed to the program running the interpreter.

SEE ALSO: `END`

5.19 R

Function: RANDOM()

Syntax: `<num-result> = RND(<maximum>)`

DESCRIPTION:

RANDOM() returns a pseudo-random integer number between 0 (inclusive) and <maximum> (exclusive). The sequence of pseudo-random numbers is identical each time you start the interpreter unless the RANDOMIZE statement is used prior to using RANDOM(): RANDOMIZE seeds the pseudo-random number generator to get a new sequence of numbers from RANDOM().

SEE ALSO: RND(), RANDOMIZE, GASDEV

EXAMPLE:

```
PRINT RANDOM(10)
Result: 8
```

Command: `RANDOMIZE`

Syntax: `RANDOMIZE [<seed-expression>]`

DESCRIPTION:

RANDOMIZE seeds the pseudo-random number generator to get a new sequence of numbers from RND(). Recommended argument to RANDOMIZE is a "random" number to randomly select a sequence of pseudo-random numbers. If RANDOMIZE is not used then the sequence of numbers returned by RND() will be identical each time the interpreter is started. If no argument is given, the TIMER value will be used as a seed.

SEE ALSO: `RND(), TIMER`

Command: RBOX

Syntax: `RBOX x1,y1,x2,y2`

DESCRIPTION:

Draws a rectangle with rounded corners from the two diagonally opposite corner points ’x1,y1’ and ’x2,y2’

SEE ALSO: BOX, PBOX, PRBOX

Command: `READ`

Syntax: `READ var[,var2, ...]`

DESCRIPTION:

Reads constant values from a DATA command and assigns them to a variable ’var’. Reading is taken from the last point a RESTORE was done (if any).

SEE ALSO: `DATA`, `RESTORE`

Command: `RELSEEK`

Syntax: `RELSEEK [#]n,d`

DESCRIPTION:

Place file pointer on new relative position d which means it moves the file pointer forward (d>0) or backwards (d<0) d bytes.

SEE ALSO: `SEEK`, `LOC()`, `LOF()`, `EOF()`

Command: `REM ABBREV. ’`

Syntax:
```
REM This is a comment
’ This also is a comment
```

DESCRIPTION:

This command reserves the entire line for a comment.

COMMENT:

Note, that rem is an abbreviation for remark.

Do use comments in your programs, the more the better. Yes, the program will become longer, but it’s nice to be able to understand a well- documented program that you’ve never seen before. Or one of your own masterpieces that you haven’t looked at for a couple of years. Don’t worry about the speed of your program, except in loops or often called Procedures/Functions. There a comment-line (beginning with REM or ’) will slow the interpreter down. A comment after ’!’ has no influence on the speed of a program, so you can use these everywhere.

SEE ALSO: `!`

Command: REPEAT

Syntax: `REPEAT ... UNTIL <expression>`

DESCRIPTION:

REPEAT initiates a REPEAT...UNTIL loop. The loop ends with UNTIL and execution reiterates until the UNTIL <expression> is not FALSE (not null). The loop body is executed at least once.

SEE ALSO: DO, LOOP, UNTIL, EXIT IF, BREAK, WHILE

EXAMPLE:

```
REPEAT
    INC n
UNTIL n=10
```

Function: REPLACE$()

Syntax: `<string-result> = REPLACE$(<string-expression>, <search string>, <replace str`

DESCRIPTION:

REPLACE$() returns string-expression where all <search string> have been replaced by <replace string>.

SEE ALSO: INSTR(), WORT_SEP

EXAMPLE:

```
PRINT REPLACE$("Hello","l","w")
Result: Hewwo
```

Command: RESTORE

Syntax: `RESTORE [<label>]`

DESCRIPTION:

RESTORE sets the position DATA is read from to the first DATA line of the program (or to the first DATA line after <label> if RESTORE is used with an argument).

SEE ALSO: DATA, READ

EXAMPLE:

```
READ  a, b, c
RESTORE
READ  a, b, c
DATA  1, 2, 3
```

Command: RESUME

Syntax:
```
RESUME
RESUME NEXT
RESUME <label>
```

DESCRIPTION:

The RESUME command is especially meaningful with error capture (ON ERROR GOSUB) where it allows a reaction to an error. Anyway, X11-Basic allows the us of RESUME <label> everywhere in the program (instead of GOTO <label>), and can be used to jump out of a subroutine. If you jump into another Subroutine, you must not reach its RETURN statement. RESUME is a bad command and I dislike it very much.

RESUME repeats the erroneous command. RESUME NEXT resumes program execution after an incorrect command. RESUME <label> branches to the <label>. If a fatal error occurs only RESUME <label> is possible

**** RESUME is still not working. If you use ON ERROR GOSUB to a subroutine then RESUME NEXT is the default if the subroutine reaches a RETURN. If you want to resume somwhere else you can just GOTO out of the subroutine. This is possible, but leaves the internal stack pointer incremented, so you should not do this too often during runtime. otherwise there will be a stack overflow after 200 events. **** looks like this also happens with ON ERROR GOTO. *** In future versions of X11-Basic there might be a RESUME <label> command which properly resets the stack. If you want this to be fixed, please send me an email with your test program.

SEE ALSO: `ON ERROR`, `GOTO`, `ERROR`

Command: RETURN

Syntax: `RETURN`
`RETURN <expression>`

DESCRIPTION:

RETURN terminates a PROCEDURE reached via GOSUB and resumes execution after the calling line. Note that code reached via ON ERROR GOSUB should be terminated with a RESUME NEXT, not with RETURN.

RETURN <expression> states the result of the expression as a result of a user defined function. This can not be used in PROCEDURES but in FUNCTIONS. The expression must be of the type the function was.

SEE ALSO: PROCEDURE, FUNCTION, ENDFUNCTION, RESUME, GOSUB, @, ON ERROR

EXAMPLE:

```
PROCEDURE testroutine
    PRINT "Hello World !"
RETURN
FUNCTION givemefive
    RETURN 5
ENDFUNCTION
```

Function: `REVERSE$()`

Syntax: `a$=REVERSE$(t$)`

DESCRIPTION:

Reverses a String.

SEE ALSO: `UPPER\$()`, `TRIM\$()`

Function: `RINSTR()`

Syntax: `<int-result>=RINSTR(s1$,s2$[,n])`

DESCRIPTION:

Operates in same way as INSTR except that search begins at the right end of s1$. tests if s2$ is contained in s1$, then returns start-position of s2$, else 0. start comparison at pos. n (default=1) start comparison at right

SEE ALSO: `INSTR()`

Function: `RLD$()`

Syntax: `a$=RLD$(a$)`

DESCRIPTION:

Does a run length decoding of string a$. This function reverses the Run Length Encoding function on a string.

In the input string, any two consecutive characters with the same value flag a run. A byte following those two characters gives the count of additional(!) repeat characters, which can be anything from 0 to 255.

SEE ALSO: `RLE\$()`

Function: RLE$()

Syntax: `a$=RLE$(a$)`

DESCRIPTION:

Does a run length encoding of string a$. This function performs a Run Length Encoding function on a string. In the output string, any two consecutive characters with the same value flag a run. A byte following those two characters gives the count of additional(!) repeat characters, which can be anything from 0 to 255. The resulting String might be shorter than the input string if there are many equal characters following each other. In the worst case the resulting string will be 50% longer.

SEE ALSO: `RLD\$()`

Function: RND()

Syntax: `<num-result> = RND(<dummy>)`

DESCRIPTION:

RND() returns a pseudo-random number between 0 (inclusive) and 1 (exclusive). The sequence of pseudo-random numbers is identical each time you start the interpreter unless the RANDOMIZE statement is used prior to using RND(): RANDOMIZE seeds the pseudo-random number generator to get a new sequence of numbers from RND().

SEE ALSO: `RANDOMIZE`, `GASDEV()`, `RANDOM()`

EXAMPLE:

```
PRINT RND(1)
Result: 0.3352227557149
```

Function: `ROUND()`

Syntax: `<num-result>=ROUND(<num-expression:b>[,n])`

DESCRIPTION:

Rounds off a value to n fractional digits. n<0: round to digits in front of the decimal point.

SEE ALSO: `INT()`, `FIX()`, `FLOOR()`, `TRUNC()`

RSRC_FREE – frees GEM rsc-File (ATARI ST) RSRC_GADDR(type,index,addr) RSRC_LOAD filename$ – loads GEM rsc-File (ATARI ST)

Command: `RUN`

Syntax: `RUN`

DESCRIPTION:

starts program execution (RUN)

SEE ALSO: `STOP`, `CONT`, `LOAD`

5.20 S

Command: `SAVE`

Syntax: `SAVE [a$]`

DESCRIPTION:

writes the BASIC-program into a file with the name a$

SEE ALSO: `LOAD`

Command: `SAVESCREEN`

Syntax: `SAVESCREEN t$`

DESCRIPTION:

Saves the whole Graphic-Screen (Desktop) into a file with name t$. The Graphics format is XWD (X-Window dump). Maybe later this will be changed to png (portable network graphic).

SEE ALSO: `SAVEWINDOW`

Command: `SAVEWINDOW`

Syntax: `SAVEWINDOW t$`

DESCRIPTION:

Saves the actual X11-Basic Graphic-Window into a file with name t$. The Graphics format is XWD (X-Window dump).

SEE ALSO: `SAVESCREEN`, `SGET`

Command: SCOPE

Syntax:

```
SCOPE a(),typ,yscale,yoffset -- Schnelles Plotten des Feldes a()
SCOPE y(),x(),typ,yscale,yoffset,xscale,xoffset -- 2D Plot
```

DESCRIPTION:

SEE ALSO: LINE, POLYLINE

Command: SCREEN

Syntax: `SCREEN n`

DESCRIPTION:

This commands select the Screen-Resolution in SVGA-Mode. It is only available in the SVGA-Version of X11-Basic and has no effect on the X11-Version or WINDOWS-Version.

Following Screen modes are supported: n Mode === 0 TEXT-Mode, no graphics 1 320x 200 16 colors 2 640x 200 16 colors 3 640x 350 16 colors 4 640x 480 16 colors 5 320x 200 256 colors 6 320x 240 256 colors 7 320x 400 256 colors 8 360x 480 256 colors 9 640x 480 monochrome 10 640x 480 256 colors 11 800x 600 256 colors 12 1024x 768 256 colors

13 1280x1024 256 colors

14 320x200 15Bit colors 15 320x200 16Bit colors 16 320x200 24Bit colors

17 640x480 15Bit colors 18 640x480 16Bit colors 19 640x480 24Bit colors

20 800x600 15Bit colors 21 800x600 16Bit colors 22 800x600 24Bit colors

23 1024x768 15Bit colors 24 1024x768 16Bit colors 25 1024x768 24Bit colors

26 1280x1024 15Bit colors 27 1280x1024 16Bit colors 28 1280x1024 24Bit colors

29 800x 600 16 colors 30 1024x 768 16 colors 31 1280x1024 16 colors

32 720x 348 monochrome Hercules emulation mode 33-37 32-bit per pixel modes. 38-74 additional resolutions

SEE ALSO: `VGA-Version of X11-Basic`

Command: SEEK

Syntax: `SEEK #n[,d]`

DESCRIPTION:

Place file pointer of channel n on new absolute position d (Default on Position 0 which is the beginning of File.)

SEE ALSO: RELSEEK, LOC(), EOF(), LOF()

Command: `SEND`

Syntax: `SEND #n,msg$[,adr%,port%]`

DESCRIPTION:

SEND is used to transmit a message via fast UDP datagramms to another socket which may be on another host. Send with only two parameters may be used only when the socket is in a connected state (see CONNECT), otherwise the destination adress and the port has to be specified.

The address of the target is given by adr%, which usually contains a IP4 adress (e.g. cvl(chr$(127)+chr$(0)+chr$(0)+chr$(1)) which corresponds to 127.0.0.1). msg$ can be an arbitrary message with any data in it. The length of the message must not exceed 1500 Bytes. If the message is too long to pass atomically through the underlying protocol, an error occurs, and the message is not transmitted.

No indication of failure to deliver is implicit in a send.

When the message does not fit into the send buffer of the socket, send blocks. The OUT?() function may be used to determine when it is possible to send more data.

SEE ALSO: `OPEN`, `CLOSE`, `CONNECT`, `RECEIVE`, `OUT?()`

SELECT ??? SETENV t$=a$ – Sets environmentvar t$ using value a$

Command: `SETFONT`

Syntax: `SETFONT t$`

DESCRIPTION:

Loads and sets a X11 Font. t$ may be the fontname or any valid Font pattern.

SEE ALSO: `TEXT`

Command: `SETMOUSE`

Syntax: `SETMOUSE x,y[,k]`

DESCRIPTION:

The SETMOUSE command permits the positioning of the mouse cursor under program control. Tje optional parameter k can simulate the mouse button being pressed or released.

SEE ALSO: `MOUSE`

Command: `SGET`

Syntax: `SGET screen$`

DESCRIPTION:

stores content of actual X11-Basic Graphics window in screen$.

SEE ALSO: `SPUT`, `SAVEWINDOW`

Function: `SGN()`

Syntax: `a=SGN(b)`

DESCRIPTION:

SGN returns the sign of a number b. It may be -1 if b is negative 0 if b equals 0 1 if b is positive.

SEE ALSO: `ABS()`

Function: `SHM_ATTACH()`

Syntax: `adr=SHM_ATTACH(id)`

DESCRIPTION:

Attach a shared memory segment to programs address space.

SEE ALSO: `SHM\_MALLOC()`, `SHM\_DETACH`, `SHM\_FREE`

Command: `SHM_DETACH`

Syntax: `SHM_DETACH id`

DESCRIPTION:

Detach a shared memory segment.

SEE ALSO: `SHM\_MALLOC()`, `SHM\_ATTACH()`

Command: SHM_FREE

Syntax: `SHM_FREE adr`

DESCRIPTION:

Free a shared memory segment.

SEE ALSO: `SHM\_MALLOC()`

Function: SHM_MALLOC()

Syntax: `adr=SHM_MALLOC(a,b)`

DESCRIPTION:

Allocate a shared memory segment.

SEE ALSO: SHM_FREE

Command: `SHOWPAGE`

Syntax: `SHOWPAGE`

DESCRIPTION:

refreshes graphic output

SEE ALSO: `VSYNC`

Function: `SIN()`

Syntax: `<num-result>=SIN(<num-expression>)`

DESCRIPTION:

Returns the sinus of the expression in radians.

EXAMPLE:

```
PRINT SIN(PI/2)     Result: 1
```

SEE ALSO: `COS(),TAN(),ACOS()`

Function: `SINH()`

Syntax: `<num-result>=SINH(<num-expression>)`

DESCRIPTION:

Returns the sinus hyperbolicus of the expression in radians.

SEE ALSO: `SIN()`, `ASINH()`

Command: `SIZEW`

Syntax: `SIZEW nr,w,h`

DESCRIPTION:

Resizes the graphic window nr with width w and heigth h.

SEE ALSO: `OPENW`, `MOVEW`

Command: SORT

Syntax: `SORT <array-name>()[,<n>[,<2nd array>()]]`

DESCRIPTION:

SORT sorts the one-dimensional array specified as argument. Numeric arrays and string arrays can be sorted. If <n> is given, only the first <n> values are sorted. If <2nd array> is given, this (numerical) array will also be sorted corresponding to the first. This is useful for creating an index table.

Command: `SOUND`

Syntax: `SOUND <frequency>`

DESCRIPTION:

SOUND sounds the internal speaker with frequency $[$Hz$]$. If frequency=0 shut up speaker.

COMMENT: The internal speaker is accessed via a console device. The sound does not work unter xterm.

EXAMPLE:

```
SOUND 500
PAUSE 0.1
SOUND 0
```

SEE ALSO: `PAUSE`

Variable: `SP`

Syntax <int-value>=SP

DESCRIPTION:

Variable represents the internal X11-Basic Stack Pointer.

SEE ALSO: `PC`

Function: `SPACE$()`

Syntax: `t$=SPACE$(n)`

DESCRIPTION:

Returns a string containing n spaces.

SEE ALSO: `STRING\$()`

Function: `SOLVE()`

Syntax: `x()=SOLVE(m(),d())`

DESCRIPTION:

Soves a set of linear equations d()=M()*x(). M() has to be a 2 dim Array not necessarily a square matrix. d() must be a 1 dim Array with exactly as many elements as lines of M(). x() will be a 1 dim Array with exactly as many elements as rows of M(). Internally a singular value decomposition is used to solve the equation. If singular values found to be smaller than 1e-10 of the largest value they are set to zero.

SEE ALSO: `INV(),DET()`

Command: SPLIT

Syntax: `SPLIT t$,t$,i%,var$[,var$]`

DESCRIPTION:

Same as WORT_SEP.

SEE ALSO: WORT_SEP

Command: `SPUT`

Syntax: `SPUT screen$`

DESCRIPTION:

(xwd) Grafik in screen$ auf Window **SEE ALSO:** `SGET`, `PUT\_BITMAP`

Function: SQR(), SQRT()

Syntax:
```
<num-result> = SQR(<num-expression>)
<num-result> = SQRT(<num-expression>)
```

DESCRIPTION:

SQR() and SQRT() return the square root of its argument.

EXAMPLES:

```
PRINT SQR(25)
Result: 5

PRINT "Calculate square root of a number."
INPUT "Number=",z
r124=1
105:
r123=r124
r124=(r123^2+z)/(2*r123)
IF ABS(r124-r123)-0.00001>0
  PRINT r124
  GOTO 105
ENDIF
PRINT "Result of this algorithm:"'r124
PRINT "Compare with: sqrt(";z;")=";SQRT(z)
PRINT "Deviation:"'ABS(SQRT(z)-r124)
```

SEE ALSO: `Operator ^`

Function: `SRAND()`

Syntax: `VOID SRAND(b)`

DESCRIPTION:

The SRAND() function sets its argument as the seed for a new sequence of pseudo-random integers to be returned by RAND(), RANDOM() or RND(). These sequences are repeatable by calling SRAND() with the same seed value.

SEE ALSO: `RANDOMIZE`, `RANDOM()`, `RND()`, `RAND()`

Variable: `STIMER`

Syntax: `<int-result>=STIMER`

DESCRIPTION:

Integer part of TIMER

SEE ALSO: `TIMER`, `CTIMER`

Command: `STOP`

Syntax: `STOP`

DESCRIPTION:

STOP halts program execution and sets the interpreter to interactive mode. The execution can be continued with the CONT command.

SEE ALSO: `CONT`, `END`, `QUIT`

STR$(1-4) t$=STR$(a$[$,b,c$]$) – convert number to String of length b with c signifikant digits STRING$(1-2) SUB var,num SUCC(1) SWAP ???,???

Function: `SWAP()`

Syntax: `<num-result>=SWAP(<num-expression:b>)`

DESCRIPTION:

Swaps High and Low words of b and returns the result.

SEE ALSO: `BYTE()`, `CARD()`, `WORD()`

Function: SYM_ADR()

Syntax: `adr=SYM_ADR(#n,sym_name$)`

DESCRIPTION:

SYM_ADR() resolves the adress of a symbol name of a given shared object library which has been linked before.

EXAMPLE:

```
t$="/usr/lib/libreadline.so"    !  If the readline shared object file
IF EXIST(t$)                    !  exist,
  LINK #1,t$                    !  link it, resolve the symbol "readline"
  DUMP "#"                      !  and execute that subroutine with
  promt$=">>>"                  !  one string parameter.
  adr=EXEC(SYM_ADR(#1,"readline"),L:VARPTR(promt$))
  r=adr
  WHILE PEEK(r)>0               ! Print the result
    PRINT CHR$(PEEK(r));
    INC r
  wend
  PRINT
  UNLINK #1 ! Unlink the dynamic lib
  FREE adr
ENDIF
```

SEE ALSO: LINK, UNLINK

Command: SYSTEM

Syntax: `SYSTEM <string-expression>`

DESCRIPTION:

Passes a command to the shell. Executes shell command. This command provides a way to use alle commands like rm, rmdir, mkdir etc. which are not implemented in X11-Basic.

EXAMPLE:

```
SYSTEM "mkdir folder"
```

SEE ALSO: `SYSTEM\$()`

Function: SYSTEM$()

Syntax: `<string-result>=SYSTEM$(<string-expression>)`

DESCRIPTION:

Passes a command to the shell. Executes shell command. The function returns a string containing the stdout of the command executed.

EXAMPLE:

```
d$=SYSTEM$("ls")
PRINT d$
```

SEE ALSO: SYSTEM

5.21 T

Function: `TERMINALNAME$()`

Syntax: `t$=TERMINALNAME$(#n)`

DESCRIPTION:

returns device name of terminal connected to #n if it is a terminal device.

Command: TEXT

Syntax: `TEXT x,y,t$`

DESCRIPTION:

Draws Text t$ in graphics window at position x,y.

EXAMPLE:

```
' Show the complete ASCII Font
SETFONT "*writer*18*"
COLOR GET_COLOR(65535,10000,10000)
FOR x=0 to 15
  FOR y=0 to 15
    TEXT 320+16*y,20+24*x,CHR$(y+16*x)
  NEXT y
NEXT x
SHOWPAGE
```

SEE ALSO: SETFONT

Variable: `TIMER`

Syntax: `TIMER`

DESCRIPTION:

Returns actual time in number of seconds since 01.01.1970.

SEE ALSO: `STIMER`, `CTIMER`, `TIME\$`, `DATE\$`, `UNIXTIME\$()`, `UNIXDATE\$()`

Command: `TITLEW`

Syntax: `TITLEW n,title$`

DESCRIPTION:

Gives the window number ’n’, the new title title$

SEE ALSO: `OPENW`, `INFOW`

Command: `TOPW`

Syntax: `TOPW [n]`

DESCRIPTION:

Activates the windows number `n` and moves it to the top of the window stack.

SEE ALSO: `BOTTOMW`, `MOVEW`

Command: `TROFF`

Syntax: `TROFF`

DESCRIPTION:

TROFF disables tracing output. This command is meant to be used during program development.

SEE ALSO: `TRON`, `ECHO`

Command: `TRON`

Syntax: `TRON`

DESCRIPTION:

TRON enables tracing output: each program line is displayed on the console before it is executed. This command is meant to be used during program development.

SEE ALSO: `TROFF`, `ECHO`

5.22 U

Function: `UCASE$()`

Syntax: `<string-result>=UCASE$(<string-expression>)`

DESCRIPTION:

Transforms all lower case letters of a string to upper case. Any non letter characters are left unchanged.

SEE ALSO: `UPPER\$(), LOWER\$()`

Variable: `UNCOMPRESS$()`

Syntax: `t$=UNCOMPRESS$(c$)`

DESCRIPTION:

Uncompresses the content of a string which has been compressed with the COMPRESS$() function before.

SEE ALSO: `COMPRESS\$()`

Variable: `UNIX?`

Syntax: `<bool-result>=UNIX?`

DESCRIPTION:

Returns TRUE (-1) If the program is running under a UNIX evironment.

SEE ALSO: `WIN32?`, `TT?`

Function: `UNIXTIME$(), UNIXDATE$()`

Syntax:
```
t$=UNIXTIME$(i)
d$=UNIXDATE$(i)
```

DESCRIPTION:

These functions return the date and time as a string which has the same format as DATE$ and TIME$ given by a TIMER value.

EXAMPLE:

```
PRINT UNIXDATE$(1045390004.431), UNIXTIME$(1045390004.431)
Result:   16.02.2003   11:06:44
```

SEE ALSO: DATE\$, TIME\$, TIMER

Command: `UNLINK`

Syntax: `UNLINK #n`

DESCRIPTION:

unlinks shared object file which has been linked before and occupies channel nr #n.

SEE ALSO: `LINK`, `CLOSE`

Command: UNTIL

Syntax: `UNTIL <expression>`

DESCRIPTION:

UNTIL terminates a REPEAT...UNTIL loop.

SEE ALSO: REPEAT, DO

EXAMPLE:

```
REPEAT
    N=N+1
UNTIL (N=10)
```

Function: `UPPER$()`

Syntax: `<string-result>=UPPER$(<string-expression>)`

DESCRIPTION:

Transforms all lower case letters of a string to upper case. Any non letter characters are left unchanged.

SEE ALSO: `UCASE\$(), LOWER\$()`

5.23 V

Function: `VAL()`

Syntax: `<num-result> = VAL(<string-expression>)`

DESCRIPTION:

VAL() converts String/ASCII to a numeric value.

SEE ALSO: `VAL?(), STR\$()`

EXAMPLE:

```
a=VAL("3.1415926")
```

Function: `VAL?()`

Syntax: `a=val?(t$)`

DESCRIPTION:

Returns number of characters forim string which can be converted to a number.

EXAMPLE:

```
print val?("12345.67e12Hallo")  Result: 11
```

SEE ALSO: `VAL()`

Function: `VARPTR()`

Syntax: `<adr>=VARPTR(<variable>)`

DESCRIPTION:

Determines the address of a variable and returns a pointer. VARPTR() can also be used to determin the adress of an array index.

EXAMPLE:

```
PRINT VARPTR(t$), VARPTR(a(2,4))
```

SEE ALSO: `ARRPTR()`

Command: VERSION

Syntax: `VERSION`

DESCRIPTION:

Shows X11-Basic version number and date.

EXAMPLE:

```
VERSION
Result: X11-BASIC Version: 1.08 vom Sat Feb 15 12:00:38 CET 2003
```

Command: VOID ABBREV. ~

Syntax: `VOID <expression>`

DESCRIPTION:

This command performs a calculation and forgets the result. Sounds silly but there are occasions when this command is required, eg. when you want to execute a function but you are not really interested in the return value. e.g. waiting for a keystroke (inp(-2)).

SEE ALSO: GOSUB, @

EXAMPLE:

```
~INP(-2)
VOID FORM_ALERT(1,"[1][Hello][OK]")
```

Command: `VSYNC`

Syntax: `VSYNC`

DESCRIPTION:

Enables synchronization with the screen. Actually this is a synonyme for SHOWPAGE. Graphic output will not be shown in the window until SHOWPAGE (or VSYNC).

SEE ALSO: `SHOWPAGE`

5.24 W

Command: WHILE

Syntax: `WHILE <num-expression>`

DESCRIPTION:

WHILE initiates a WHILE...WEND loop. The loop ends with WEND and execution reiterates while the WHILE <num-expression> is not FALSE (not null). Unlike a REPEAT...UNTIL loop, the loop body is not necessarily executed at least once.

SEE ALSO: WEND, DO

EXAMPLE:

```
WHILE NOT EOF(#1)
    LINEINPUT #1,a$
WEND
```

Command: WEND

Syntax: `WEND`

DESCRIPTION:

WEND terminates a WHILE...WEND loop. **SEE ALSO:** `WHILE`, `DO`

EXAMPLE:

```
WHILE NOT EOF(#1)
    LINEINPUT #1,a$
WEND
```

Function: `WORD()`

Syntax: `a=WORD(b)`

DESCRIPTION:

Returns lower 16 bits of b and expands sign.

SEE ALSO: `BYTE()`, `CARD()`, `SWAP()`

Command: `WORT_SEP`

Syntax: `WORT_SEP t$,d$,mode,a$,b$`

DESCRIPTION:

Splits up string t$ into two parts a$ and b$ concerning a delimiter string d$. So that t$=a$+d$+b$. mode can be: 0 – default 1 – do not serch parts of t$ which are in brackets.

Quoted parts of the string are not spit up.

EXAMPLE:

```
WORT_SEP "Hallo, this is a string."," ",0,a$,b$
```

SEE ALSO: `SPLIT`, `WORT\_SEP()`

Function: `WORT_SEP()`

Syntax: `<num-result>=WORT_SEP(t$,d$,mode,a$,b$)`

DESCRIPTION:

Splits up string t$ into two parts a$ and b$ concerning a delimiter string d$. So that t$=a$+d$+b$. mode can be: 0 – default 1 – do not serch parts of t$ which are in brackets.

Quoted parts of the string are not spit up.

The return value can be: 2 – The string has been split up. 1 – The string did not contain d$, a$=t$, b$="" 0 – The string was empty. a$="",b$=""

SEE ALSO: `SPLIT`

5.25 X

Command: `XLOAD`

Syntax: `XLOAD`

DESCRIPTION:

Opens a fileselector where the user can select a basic source file which then will be load into memory.

SEE ALSO: `XRUN`, `LOAD`

Operator: XOR

Syntax: `<num-expression1> XOR <num-expression2>`

DESCRIPTION:

Logical exclusive OR operator. XOR returns fale (0) if both arguments have the same logical value.

Table:

A & B & A XOR B
-1 & -1 & 0
-1 & 0 & -1
0 & -1 & -1
0 & 0 & 0

SEE ALSO: NAND, OR, NOT, AND

EXAMPLE:

```
Print 3=3 XOR 4>2        Result:   0 (flase)
       Print 3>3 XOR 5>3        Result:  -1 (true)

PRINT (4 XOR 255)        Result:   251
```

Function: `XOR()`

Syntax: `<num-result>=XOR(<num-expression>,<num-expression2>)`

DESCRIPTION:

Returns <num-expression> XOR <num-expression2>

Table:

a & b & XOR(a,b)
-1 & -1 & 0
-1 & 0 & -1
0 & -1 & -1
0 & 0 & 0

SEE ALSO: `OR()`, `AND`

Command: `XRUN`

Syntax: `XRUN`

DESCRIPTION:

Opens a fileselector where the user can select a basic source file which then will be load into memory and executed.

SEE ALSO: `XLOAD`, `RUN`

6 Compatibility

6.1 General remarks

X11-Basic deviates in numerous aspects from ANSI BASIC. It in event is also different from Gfa-Basic (Atari ST) all though it really looks similar:

ELSE IF vs. ELSEIF

This interpreter uses the ELSE IF form of the "else if" statement with a space between ELSE and IF. In contrast, ANSI BASIC uses ELSEIF and END IF. Other interpreters may also use the combination ELSEIF and END IF.

Local variables

Local variables must be declared local in the procedure/function. Any other variables are treated as global.

Call By-Value vs. By-Reference

Variables in a GOSUB statement as in `GOSUB test(a)` are passed "by-value" to the PROCEDURE: the subroutine gets the value but can not change the variable. To pass the variable "by-reference", use the VAR keyword as in "GOSUB test(VAR a)": the subroutine then not only gets the value but the variable itself and can change it (for more information, see the documentation of the GO SUB statement). The same rules apply to FUNCTION: VAR in the pa rameter list of a function call allows a FUNCTION to get a vari able parameter "by-reference". In contrast, traditional BASIC in terpreters always pass variables in parameter lists "by-refer ence". The problem with "by-reference" parameters is that you must be fully aware of what happens inside the subroutine: as signments to parameter variables inside the subroutine might change the values of variables in the calling line.

Assignment operator

X11-BASIC does not have an assignment operator but overloads the equal sign to act as the assignment operator or as comparison operator depending on context: In a regular expression, all equal signs are considered to be the comparison operator, as in `IF (a=2).` However, in an "assignment-style" expression (as in `LET a=1`), the first equal sign is considered to be the assignment operator. Here is an example which assigns the result of a comparison (TRUE or FALSE) to the variable <a> and thus shows both forms of usage of the equal sign:

```
a=(b=c)
```

Assignments to modifiable lvalue

Some implementations of BASIC allow the use of functions on the left side of assignments as in `MID$(a$,5,1)="a"`. X11-Basic does not support this syntax but requires a variable (a "modifiable lvalue") on the left side of such expressions.

TOS/GEM implementation

Because Gfa-Basic on ATARI-ST makes much use of the built in GUI functions of the ATARI ST, the X11-Basic interpreter can only have limited compatibility. GEM style (and compatible) ALERT boxes, menus and object trees are included in the interpreter and can be used in a similar way. Even ATARI ST *.rsc files can be loaded. But Other functions like LINEA functions, the VDISYS, GEMSYS, BIOS, XBIOS and GEMDOS calls are (of course) not possible. Also many other commands are not implemented because the author thinks that they have no useful effect on UNIX platforms. Some might be included in a later Version of X11-Basic (see the list below). Since many Gfa-basic programs make use of more or less of these functions, they will have to be modified before they can be run with X11-Basic.

The INLINE statement

The INLINE statement is not supported, because the source code of X11-Basic programs is pure ASCII text. But an alternative has been implemented. (see `INLINE$()`).

6.2 GFA-Basic compatibility

Following GFA-Basic Commands and functions are not supported and probably never will be. Most of them are obsolete on unix systems. When porting from GFA-Basic to X11-Basic they have to be removed or replaced by an alternative routine:

obsolete, because there is an alternative function in X11-Basic:

`==`	Comparison operator for approximately equal –> =
`ARECT, ATEXT, HLINE`	LINE-A functions–> LINE, BOX, TEXT
`ACHAR, ACLIP, ALINE, APOLY`	TEXT,CLIP,LINE, POLY

`COSQ(),SINQ()`	quick cosine/sine using an internal table –> COS(),SIN()
`DIR`	Lists the files on a disc. –> system "ls"
`DIR$()`	Names the active directory –> env$("PWD")
`FILES`	Lists the files on a disk. –> system "ls -l"
`FRE()`	Returns the amount of memory free (in bytes). –> see below
`KILL`	Deletes a file –> system "rm -f ..."
`MKDIR`	Creates a new directory. –> system "mkdir ..."
`MSHRINK()`	Reduces the size of a storage area –> `REALLOC()`
`NAME AS`	Renames an existing file. –> system "mv ..."
`RC_COPY`	Copies rectangular screen sections (–> `COPYAREA`)
`RENAME AS`	Rename a file. ⟶ `SYSTEM "mv ..."`
`RESERVE`	Increases or decreases the memory used by basic (obsolete)
`RMDIR`	Deletes empty directories ⟶ `SYSTEM "rmdir "`
`SHEL_FIND()`	⟶ `SYSTEM "find ..."`
`SYSTEM`	obsolete ⟶ `QUIT`
`SHEL_ENVRN()`	⟶ `ENV$()`
`SHEL_READ`	obsolete ⟶ `PARAM$()`
`THEN`	keyword in If statements (obsolete)

For some GFA-Basic commands you can construckt replacement functions in X11-Basic like:

```
' Get the free memory available (in Bytes)
' n=0 physical memory
' n=1 Swap space
FUNCTION fre(n)
  LOCAL i,a,t$,a$
  a=FREEFILE()
  OPEN "I",#a,"/proc/meminfo"
  LINEINPUT #a,t$
  IF n=1
    LINEINPUT #a,t$
  ENDIF
  LINEINPUT #a,t$
  t$=TRIM$(t$)
  FOR i=0 to 3
    WORT_SEP t$," ",0,a$,t$
  NEXT i
  CLOSE #a
  RETURN val(a$)
ENDFUNCTION
```

obsolete, because TOS-Specific, and no similar function on other OSes exist:

`ADDRIN, ADDROUT`	address of the AES Address Input/Output blocks
`APPL_EXIT()`	Declare program has finished
`APPL_INIT()`	Announce the program as an application.

APPL_TPLAY()	Plays back a record of user activities
APPL_TRECORD()	makes a record of user activities
BIOS()	call BIOS routies
CONTRL	Address of the VDI control table.
FGETDTA()	Returns the DTA (Disk Transfer Address).
FSETDTA	Sets the address of the DTA
GB, GCONTRL	Address of the AES Parameter/control Block
GDOS?	Returns TRUE (-1) if GDOS is resident
GEMDOS()	call the GEMDOS routines.
GEMSYS	call the AES routinen
GINTIN, GINTOUT	Address of the AES Integer input/output block.
HIMEM	address of the area of memory which is not allocated by interpreter
INTIN, INTOUT	Address of the VDI integer Input/output block.
L~A	Returns base address of the LINE-A variables.
MENU_REGISTER()	Give a desk accessory a name
MONITOR	Calls a monitor resident in memory.
SHEL_GET, SHEL_PUT	obsolete
SHEL_WRITE	obsolete
VDIBASE, VDISYS	VDI functions
VQT_EXTENT	coordinates of a rectangle which surround the text
VQT_NAME()	VDI function
VSETCOLOR	TOS specific
VST_LOAD_FONTS(), VST_UNLOAD_FONTS()	
V_CLRWK(), V_CLSVWK(), V_CLSWK(), V_OPNVWK()	
V_OPNWK(), V_UPDWK()	VDI-GDOS functions
V~H	Returns the internal VDI handle
W_HAND(#n)	Returns the GEM handle of the window
W_INDEX()	Returns the window number of the specified GEM handle.
WORK_OUT()	Determines the values from OPEN_WORKSTATION.
XBIOS()	call XBIOS system routines.

obsolete, because ATARI-ST-Hardware-Specific, and no similar function exists on UNIX platforms:

CHDRIVE	Sets the default disk drive
DMACONTROL, DMASOUND	Controls the DMA sound
INPMID$	read data from the MIDI port
LPENX, LPENY	Returns the coordinates of a light pen.
PADT(), PADX(), PADY()	Reads the paddle on the STE
STICK	control the joystick

`SDPOKE, SLPOKE, SPOKE`	Supervisor mode memory access

not supported because of other reasons:

`APPL_READ()`	read from the event buffer.
`APPL_WRITE()`	write to the event buffer.
`BASEPAGE`	address of the basepage
`BITBLT`	Raster copying command
`CFLOAT()`	Changes integer into a floating point number.
`DEFBIT, DEFBYT, DEFWRD, DEFFLT, DEFSTR`	sets the varaible type
`DFREE()`	free space on a disc
`GETSIZE()`	return the number of Bytes required by a screen area
`HARDCOPY`	Prints the screen –> savescreen
`INPAUX$`	read data from the serial port
`KEYDEF`	Assign a string to a Function Key.
`LLIST`	Prints out the listing of the current program.
`LPOS()`	column in which the printer head is located
`LPRINT`	prints data on the printer.
`PSAVE`	save with protection
`RCALL`	Calls an assembler routine
`SCRP_READ(), SCRP_WRITE()`	Communication between GEM programs.
`SETCOLOR i,r,g,b`	set rgb value of color cell (–>GET_COLOR())
`SETTIME`	Sets the time and the date.
`WAVE`	Produces noises from the sound channels.
`WINDTAB`	Gives the address of the Window Parameter Table.

`WIND_CALC(), WIND_CLOSE(), WIND_CREATE(), WIND_DELETE(), WIND_FIND(),`
`WIND_GET(), WIND_OPEN(),`
`WIND_SET(), WIND_UPDATE()`
	GEM-Wondow-Function

`_DATA`	Specifies the position of the DATA pointer.

These GFA-Basic commands may be supported in a later versions of X11-Basic:

`ABSOLUTE x,y`	Assigns the address y to the variable x.
`APPL_FIND(fname$)`	Returns the ID of the sought after application.
`BYTE{x}`	read the contents of the address x
`C:`	Calls a C or assembler program with parameters as in C
`CARD{x}`	Reads/writes a 2-byte unsigned integer

`CHAR{x}`	Reads a string of bytes until a null byte is encountered
`CHDIR`	Changes the current directory.
`CRSCOL  CRSLIN`	Returns current cursor line and column.
`CURVE x0,y0,x1,y1,x2,y2,x3,y3`	The BEZIER-Curve
`DEFLIST x`	Defines the program listing format.
`DEFMARK [C],[A],[G]`	Sets colour, type and size of the corner points
`DEFNUM n`	Affects output of numbers by the PRINT command
`DELETE x(i)`	Removes the ith element of array x.
`DO UNTIL`	extention
`DO WHILE`	extention
`DOUBLE{x}`	reads/writes an 8-byte floating point variable
`DRAW`	Draws points and lines
`EVNT_BUTTON()`	Waits for one or more mouse clicks
`EVNT_DCLICK()`	Sets the speed for double-clicks of a mouse button.
`EVNT_KEYBD()`	Waits for a key to be pressed and returns a word
`EVNT_MESAG()`	Waits for the arrival of a message in the event buffer.
`EVNT_MOUSE()`	Waits for the mouse pointer to be located inside
`EVNT_MULTI()`	Waits for the occurence of selected events.
`EVNT_TIMER()`	waits for a period of time
`FATAL`	Returns the value 0 or -1 according to the type of error
`FIELD`	Divides records into fields.
`FILL`	Fills a bordered area with a pattern
`FORM INPUT`	Enables the insertion of a character string
`FORM INPUT AS`	the current value of a$ is displayed, and can be edited.
`FORM_BUTTON()`	Make inputs in a form possible using the mouse.
`FORM_ERROR()`	Displays the ALERT associated with the error numbered
`FORM_KEYBD()`	Allows a form to be edited via the keyboard.
`FSEL_INPUT()`	Calls the AES fileselect library
`FSFIRST()`	Searches for the first file to fulfill the criteria
`FSNEXT()`	Search for the next file which fulfills the conditions
`GET #`	Reads a record from a random access file.
`GRAF_DRAGBOX()`	a rectangle to be moved about the screen
`GRAF_GROWBOX()`	Draws an expanding rectangle.
`GRAF_HANDLE()`	supplies the size of a character from the system set
`GRAF_MKSTATE()`	supplies the current mouse coordinates
`GRAF_MOUSE)`	allows the mouse shape to be changed.
`GRAF_MOVEBOX()`	a moving rectangle with constant size
`GRAF_RUBBERBOX()`	draws an outline of a rectangle
`GRAF_SHRINKBOX()`	Draws an shrinking rectangle.
`GRAF_SLIDEBOX()`	moves one rectangular object within another
`GRAF_WATCHBOX()`	monitors an object tree while a mouse button is pressed
`HIDEM`	Switches off the mouse pointer.

`HTAB`	Positions the cursor to the specified column.
`OUT?()`	output to device
`INSERT`	Inserts an element into an array.
`INT{x}`	Reads/writes a 2 byte signed integer from/to address x.
`KEYGET n`	similar to INKEY$ but wait
`KEYLOOK n`	similar to INKEY$ but putback chars
`KEYTEST n`	similar to INKEY$
`KEYPAD n`	Sets the usage of the numerical keypad.
`KEYPRESS n`	This simulates the pressing of a key.
`LOCATE`	Positions the cursor to the specified location.
`LONG{x}`	Reads/writes a 4 byte integer from/to address x. Abbrev
`LOOP UNTIL condition`	extention
`LOOP WHILE condition`	extention
`LSET var=string`	Puts the 'string' in the string variable 'var' lefft justified
`MAT ADD a(),b()`	
`MAT ADD a(),x`	
`MAT CLR a()`	
`MAT CPY a([i,j])=b([k,l])[,h,w]`	
`MAT DET x=a([i,j])[,n]`	
`MAT INPUT #i,a()`	
`MAT INV a()=b()`	
`MAT MUL`	
`MAT MUL a(),x`	
`a()=b()*c()`	
`MAT MUL x=a()*b()`	
`MAT MUL x=a()*b()*c()`	
`MAT NORM a(),{0/1}`	
`MAT ONE a()`	
`MAT PRINT [#i]a[,g,n]`	
`MAT QDET x=a([i,j])[,n]`	
`MAT RANG x=a([i,j])[,n]`	
`MAT READ a()`	
`MAT SET a()=x`	
`MAT SUB a(),b()`	
`MAT SUB a(),x`	
`MAT TRANS a()[=b()]`	
`MAT XCPY a([i,j])=b([k,l])[,h,w]`	
`MAT BASE {0/1}`	
`MAX(x [,y,z,...])  or  MAX(a$[,b$,c$...])`	Returns the greatest value (or largest string)
`MENU(x)`	Returns the information about an event in the variable
`MENU OFF`	Returns a menu title to 'normal' display.
`MENU_BAR()`	Displays/erases a menu bar (from a resource file)

MENU_ICHECK()	Deletes/displays a tick against a menu item.
MENU_IENABLE()	Enables/disables a menu entry.
MENU_TEXT()	Changes the text of a menu item.
MENU_TNORMAL()	Switches the menu title to normal/inverse video.
MID$(a$,x[,y])	(as a command)
MIN(expression [,expression ...])	Returns the smallest value (or smallest string)
MODE	representation of decimal point, date and files
OBJC_CHANGE()	Changes the status of an object.
OBJC_EDIT()	Allows input and editing
OBJC_ORDER()	re-positions an object within a tree.
OB_ADR()	Gets the address of an individual object.
OB_FLAGS()	Gets the status of the flags for an object.
OB_H()	Returns the height of an object
OB_HEAD()	Points to the object's first child
OB_NEXT()	Points to the following object on the same level
OB_SPEC()	Returns the address of the the data structure
OB_STATE()	returns the status of an object
OB_TAIL()	Points to the objects last child
OB_TYPE()	Returns the type of object specified.
OB_W()	Returns the width of an object
OB_X(), OB_Y()	relative coordinates of the object
OCT$()	value in octal form
ON BREAK	influence behavior of CTRL-C
OPTION BASE	determine whether an array is to contain a zero element
QSORT	Sorts the elements of an array.
RCALL	Calls an assembler routine
RC_INTERSECT()	Detects whether two rectangles overlap.
RECALL	Inputs n lines from a text file
RECORD	Sets the number of the next record (GET#, PUT#)
RND as a sysvar	see RND()
ROL(), ROL&(), ROL%()	Rotates a bit pattern left.
ROR(), ROR&(), ROR%()	Rotates a bit pattern right.
RSET a$=b$	Moves a string expression, right justified to a string.
RSRC_OBFIX()	converts the coordinates of an object
RSRC_SADDR()	sets the address of an object.
RUN <filname>	see RUN
SETDRAW	see DRAW
SHL(), SHL&(), SHL%()	Shifts a bit pattern left
SHOWM	Makes the mouse pointer appear.
SHR(), SHR&(), SHR%()	Shifts a bit pattern left
SINGLE{x}	Reads/writes a 4 byte floating point
SPRITE	Puts a sprite

`SSORT`	Sorts using the Shell-Metzner method.
`STORE`	Fast save of a string array as a text file.
`TOUCH`	Updates the date and time stamps of a file.
`TRON#`	Tron to file
`TRON proc`	procedure is called before the execution of each command
`VARIAT()`	Returns the number of permutations of n elements to the kth order
`VTAB`	positions the cursor to the specified line number
`WRITE`	Stores data in a sequential file

Following commands have a different meaning and/or syntax in X11-Basic:

GFA-BASIC	X11-Basic
`SYSTEM`	`QUIT`
`LINE INPUT`	`LINEINPUT`
`SOUND`	-
`VSYNC`	-
`ON MENU`	`MENU`
`ON MENU GOSUB ...`	`MENUDEF`
`MENU a$()`	`MENUDEF`
`MENU OFF`	-
`MENU KILL`	`MENUKILL`
`MENU()`	-
`MONITOR`	`SYSTEM`

6.3 Ideas for future releases of X11-Basic

These are some ideas for new commands, which are not GFA-commands and which might be implemented in X11-Basic in future:

```
ENCLOSE$(str[, pair])

Encloses a string.
The default pair is ""

Example:
? enclose$("abc","()")

Result:
(abc)
========================
 LEFTOF$(s1,s2),
 RIGHTOF$(s1,s2)

  Returns the left/right part
  of s1 at the position of the
  first occurrence of the
  string s2 into string s1
========================
SPRINT var$;[USING...;]...
 PRINT anweisung, aber in var$ (Statt Print #1,...)

========================



       CRSCOL n.n.
       CRSLIN n.n.



========================
implemention of mmap in x11basic:

open "I",#1,"myfile"
adr%=map("I|O|U",#1,len,offset)


msync adr%,len

unmap adr%,len
close #1

       offset  should  be a multiple of the page size as returned
       by getpagesize(2).

"I"   entspricht PROT_READ MAP_PRIVATE
"O"   entspricht PROT_WRITE MAP_SHARED
"U"   entspricht PROT_READ PROT_WRITE MAP_SHARED

"*L" entspricht MAP_LOCKED
```

Acknowlegements

Thanks to all people, who helped me to realize this package.

Many thanks to the developers of GFA-Basic. This basic made me start programming in the 1980s. Many ideas and most of the command syntax has been taken from the ATARI ST implementation.

Thanks to sourceforge.net for hosting this project on the web.

Unfortuantely derivates of the routines of the famous book "numerical recepies in C" are not allowed to be distributed as part of the X11-Basic program. So I have not included all of the mathematics features of X11-Basic in this distribution. (see `mathemat_dummy.c`). If you own the book and have a licence you may ask me for the sources of the routines.