X11-BASIC

VERSION 1.24

User Manual

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

Latest revision: February 20, 2016

X11-Basic is a dialect of the BASIC programming language with graphics capability that integrates features like shell scripting, cgi-programming and full graphical visualization into the easy to learn BASIC language on modern 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) and the compiler (the program `xbc` under UNIX or `xbc.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 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 dialect of the BASIC programming language with graphics and sound which integrates features like traditional BASIC language syntax, structured programming, shell scripting, cgi programming, powerful math, and full graphical visualization into the easy to learn BASIC language on modern computers.

The syntax of X11-Basic is most similar to GFA-Basic in its original ancient implementation for the ATARI ST. Old GFA-programs should run with only a few changes. Also DOS/QBASIC programmers will feel comfortable.

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 proven 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.

X11-Basic supports the principle 'small is beautiful'. Its aim is to use the fewest system resources and execute with the highest speed. X11-Basic meets in this, by providing very powerful built-in commands and functions, and a very fast compiler producing even faster applications. X11-Basic lets you write an application with very little effort, giving you full control over your application. X11-Basic doesn't use "black boxes" with an enormous overhead, but instead calls operating system functions whenever possible. In case the X11-Basic commands and functions aren't sufficient, you can easily use the native shell to execute other programs and commands, or you will be able to use any shared library on the system, which can be dynamically linked.

No language is perfect and X11-Basic is no exception. It has its weak and it's strong points. You won't use X11-Basic to write major applications, but it is extremely well suited to develop small to medium sized programs. X11-Basic is nearly as versatile as C, it uses procedures and functions and parameter passing similar to C.

X11-Basic programs are constructed in a straightforward fashion. As C, X11-Basic doesn't use object oriented structures and allows an easy start.

Because it is an interpretive language each new step in your program can be tested quickly providing you with instant feedback. And when you finished your program you can use the X11-Basic compiler to create a very fast stand-alone executable. No

complicated compiler options and linker switches are necessary to create a stand-alone application.

Portability

X11-Basic is designed to run on many platforms with extremely low resources. It has started on UNIX workstations and Linux-systems with the X-Window system (commonly known as X11, based on its current major version being 11). In case where no X11 implementation is available, X11-Basic can be compiled with a framebuffer-device graphics engine. The Android version e.g. uses the framebuffer interface. Also such a version for the TomTom navigation devices has been created.

Porting X11-Basic to more basic and embedded systems with a very low amount of RAM and processing speed is well possible. On UNIX and Linux systems, not only the X11 graphics engine can be used, but also the SDL library (=Simple Direct-Media Library), as well as any raw framebuffer device or no graphics at all. The MS WINDOWS version supports only SDL (or no graphics at all).

X11-Basic supports complex numbers and complex math, as well as arbitrary precision numbers and calculations where needed, as well as very fast 32bit integer and 64bit floating point operations, very powerful string handling functions for character strings of any length and any content.

Sound is not available on every system. Where available, X11-Basic implements a 16 channel sound synthesizer as well as the option to play sound samples from standard sound file formats (line .wav and .ogg). On LINUX systems the ALSA sound engine is used. The Android port of X11-Basic uses the Android sound and speech engine.

The X11-Basic environment contains a library of GEM[1] GUI[2] functions. This makes writing GUI programs in X11-Basic faster, easier and more portable than programming with native GUI tools.

The Android version of X11-Basic contains a full featured coloured VT100/ANSI terminal emulation and support for unicode character sets (UTF-8 coded) for standard output.

[1] GEM=Graphics Environment Manager, an operating environment created by Digital Research, Inc. (DRI), which was used on the ATARI ST and GFA-BASIC.

[2] GUI=Graphical User Interface

Structured programming

X11-Basic is a structured procedural 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.

In contrast to more traditional BASIC implementations, line numbers are not used in X11-Basic. Every line holds only one instruction. Jumps with GOTO are possible but not necessary. All the well-known loops are available including additional commands for discontinuation (⟶ `EXIT IF`, `BREAK`).

Procedures and functions with return values of any type can be defined. This way BASIC programs can be structured in a modular way. A program can contain a main part to call subfunctions and subprocedures, which may or may not be defined in the same source file. Distinct sources can form a library. Whole libraries can be added with the merge command (⟶ `MERGE`).

To help porting ANSI-Basic[1] programs (with line numbers) to X11-Basic, a converter (⟶ `bas2x11basic`) has been written. It comes with the X11-Basic package.

The third-party tool `gfalist`[2] by Peter Backes (not included in the X11-Basic package) even allows to decode GFA-Basic `.gfa` files to ASCII.

Speed of X11-Basic

How fast is X11-Basic? The answer depends on the way an X11-Basic program is run: It depends on if the code is interpreted, run as bytecode in a virtual machine, or being compiled to native machine language. Generally we find:

1. X11-Basic programs run by the interpreter are slow,
2. X11-Basic programs compiled to bytecode and then run in the X11-Basic virtual machine (`xbvm`) is fast, but
3. X11-Basic bytecode compiled natively to real machine language is even faster.
4. arbitrary precision numbers and calculations are slow, but
5. 64bit floating point and complex number calculations as well as 32bit integers are very fast.

[1] So-called ANSI-Basic has been standardized by the American National Standards Institute. ANSI-Basic uses line numbers and the syntax can be quite different from X11-Basic.

[2] You will find a link to gfalist (the project name is ONS) on the X11-Basic homepage.

Bytecoded programs are always interpreted faster than scripted programming languages. The X11-Basic compiler can translate the X11-Basic bytecode to C, which then can be compiled to native machine language using any C-compiler (preferably `gcc` on UNIX systems). Obviously your programs will be slower than optimized C/C++ code but it already comes close.

If you need highest possible speed you can load and link a separate DLL/shared object with the time critical part of your code written in another language (e.g. C or Assembler).

A speed comparison was done with the Whetstone benchmark (⟶ `Whets.bas`). This shows, that bytecode-programs are about 19 times faster than the interpreted code and a natively compiled program can run about 28 times faster.

Optimality of code and code overhead

At a minimum the X11-Basic interpreter and the bytecode interpreter (virtual machine) require about 350 KB of memory and another 400 kB of file size, which includes the X11-Basic runtime-library. So this is the overhead that all your programs will have. Compared to some Windows programs, this isn't that bad. Most likely your bytecode is less than 50 kB anyway (for a moderate/large application), plus any resources and graphics you may want to include of course. In the end the code produced will be reasonably small and light enough to be also used on portable devices (e.g. cell phones, e-book readers, and navigation devices) which have only a small amount of native memory (and a relatively slow processor).

Copyright information

WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

Read the file COPYING for details.

(Basically that means, free, open source, use and modify as you like, don't incorporate it into non-free software, no warranty of any sort, don't blame me if it doesn't work.)

2 USAGE

This chapter describes how to install X11-Basic on the most popular operating systems and how to run the interpreter and how to compile BASIC programs.

The X11-Basic interpreter is called `xbasic` (`xbasic.exe` under Windows). The compiler `xbc` (`xbc.exe` under Windows). Under Unix these executables are usually installed in the `/usr/bin/` (if installed via the package management system) or in `/usr/local/bin` (if installed manually from the source package) path. Under Windows, the files are installed normally under the directory `C:\x11basic`. Under Android you will not have to care about the individual components of X11-Basic, because there the X11-Basic app comes with a little IDE (Integrated Development Environment) which handles the terminal, editor, loading running and the compile process for you.

2.1. Installing X11-Basic

For the most popular operating systems, ready-made packages are available which allow an easy installation of X11-Basic without the need of compiling it from source code.

For other operating systems not mentioned here, X11-Basic may or may not work. Generally no binary package might be available, so in these cases you will have to compile all X11-Basic components (manually) by your own. You may be lucky and you are not the first trying this, so searching the internet for hints is generally a good idea.

But most likely you are reading this manual because you have already got X11-Basic installed on your system, or you at least have a package ready to be installed right away.

SuSE-Linux and RedHat

If you have got a Redhat-Package (RPM) e.g. a file named `X11Basic-1.24-1.i386.rpm`, then you can install this package (being root) with

```
rpm -i X11Basic-1.24-1.i386.rpm      .
```

This is a very convenient way at least for the Linux distributions *Feodora*, *Mandriva*, *SuSE* and *RedHat* (and maybe others, basically derived distributions[1]) to install the interpreter, the compiler, and its documentation, the man-pages and a small collection of example programs.

Following files will be normally installed:

```
/usr/bin/xbasic            -- the X11-Basic interpreter
/usr/bin/xbc               -- the compiler
/usr/bin/xbbc              -- bytecode compiler
/usr/bin/xvbm              -- bytecode interpreter (virtual machine)
/usr/bin/xb2c              -- the bytecode to C translator
/usr/bin/bas2x11basic      -- the ANSI BASIC to X11-Basic translator
/usr/lib/libx11basic.so    -- the runtime library (shared object)
/usr/lib/libx11basic.a     -- the runtime library for static linking
/usr/include/x11basic/x11basic.h -- the header file for library API
/usr/include/x11basic/xb2csol.h  -- the header file for compilation of xb2c output
/usr/share/man/man1/x11basic.1 -- the man-page of X11-Basic
/usr/share/man/man1/xbasic.1   -- the man-page of the X11-Basic interpreter
/usr/share/man/man1/xbc.1   -- the man-page of the compiler
/usr/share/man/man1/xbbc.1  -- the man-page of the bytecode compiler
/usr/share/man/man1/xbvm.1  -- the man-page of the virtual machine
/usr/share/man/man1/xb2c.1  -- the man-page of the X11-Basic to C translator
/usr/share/man/man1/bas2x11basic.1 -- the man-page of the ANSI to X11-Basic  translator
```

After having installed the package, you can execute the interpreter with `xbasic` or read the man pages with `man xbasic` or `man x11basic`.

The documentation should install into the `/usr/share/doc/packages/X11Basic/` directory and you should find the following files:

```
-rw-r--r--    1005  ACKNOWLEGEMENTS      -- acknowledgments
-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--     170  RELEASE_NOTES        -- release notes
-rw-r--r--  164370  X11-Basic-manual.txt -- the manual (txt version)
drwxr-xr-x    1024  editors/             -- files for editors / syntax highlighting
drwxr-xr-x    1024  examples/            -- few example programs
```

Debian based distributions, Ubuntu and Knoppix

If your Linux distributions does not use the RedHat package system it is very likely that it instead uses the Debian package system. The most popular Debian based Linux distributions are *Knoppix* and *Ubuntu*[2].

[1] A list of RPM based Linux distributions can be found here: `http://en.wikipedia.org/wiki/Category:RPM-based_Linux_distributions`

[2] A list of Debian based Linux distributions can be found here: `http://en.wikipedia.org/wiki/Category:Debian-based_distributions`

X11-Basic also comes in packages called (e.g.) `x11basic_1.24-1_i386.deb`. Usually you can very easily install the file from a file browser with simply double clicking on it. Also a

```
dpkg -i x11basic_1.24-1_i386.deb
```

from a terminal will do. The file system structure should be similar to what is described in the previous chapter (explaining the RedHat packages), so you should expect to find the same files at the same places. Please note, that you need a special debian package if you want to install it on 64 bit linux installations, usually called `x11basic_1.24-1_amd64.deb`.

Other Linux and UNIX distributions

The author currently provides only 32bit and 64bit debian binary packages for linux (specifically *Ubuntu linux*). A rpm package can be made out of the debian packet with a tool called `alien`.

For exotic linux based devices usually binary distributions come as a zip file (like the TomTom version). In these cases they are accompanied by a README or other instructions how to install them. The package for Android comes in a file called `X11-Basic-1.24-22.apk` usually provided by *Google Play* (formerly known as *Android Market*), which also installs it for you. If you do not like to use *Google Play* for some reason, you can also install X11-Basic from any file browser taping on its `.apk` file, downloaded from `sourceforge.net`.

For all other systems you will have to get the source-package `X11Basic-1.24.tar.gz` and compile the sources. This should work for all Linux distributions, and probably with little modifications also for *HP-UX* (Hewlett-Packard UniX), for DEC/alpha, for MAC/OSX, for SUN/SOLARIS and FreeBSD and maybe others. Also X11-Basic compiles on Cygwin, and on ARM-Linuxes like the one often used together with the *Raspberry Pi*. Please note that X11-Basic is designed for 32-bit operating systems. X11-Basic will also compile on 64 bit systems. But some of the functions may not work, especially pointer aritmetric (`VARPTR()`, etc.) will probably lead to segmentation faults when using huge amounts of memory.

Compiling X11-Basic from its sources under UNIX like systems

If you have a binary package of X11-Basic, you can safely skip this section.

In order to compile X11-Basic, you will need the following:

- A C compiler, preferably GNU C (but other ANSI C compilers will do),
- X11 libraries (for the graphics) or a framebuffer device or the SDL library,
- optionally the `readline` library,
- optionally the `LAPACK` library,
- optionally the `GMP` library,
- optionally the `ALSA` sound library (`libasound`) and/or the SDL framework.

These will suffice to get you started. If one or more of these libraries are not present on your system, the `configure` and `make` scripts will try to compile a version, which does not need them (hence leaving out some of the functionality of X11-Basic.).

1. Install the development environment packages, e.g. done by the command:

   ```
   sudo apt-get install libx11-dev libreadline6-dev liblapack-dev libgmp
   ```

2. Unpack `X11Basic-1.24.tar.gz` with

   ```
   tar xzf X11Basic-1.24.tar.gz
   ```

3. go into the `X11Basic-1.24` directory and do a

   ```
   ./configure
   make
   sudo make install
   ```

That's all you will have to do (for more detailed installation instructions read the file INSTALL, which comes with the package.).

If the 'configure' script fails, please contact me (`kollo@users.sourceforge.net`) and send me the output it generated (`config.log`). I am going to try to help you to fix the problem.

Special comments on the framebuffer version

Very useful on the Raspberry pi and other low memory/low resources computers is the option not to use X or SDL libraries at all. You can have a full featured X11-basic with graphics and mouse input anyway, if you compile the framebuffer version (`make fb`). This will produce the single file `xbasic.framebuffer` which is the interpreter (and virtual machine) ready to be used from a console (and without X). This way you have full comtrol over the screen and mouse and keyboard. Usually everything you need to make the Raspberry pi interact with and display to the user.

Cross-compiling other Versions of X11Basic

The Makefile allows you to also produce the compiler (`make xbc`), the bytecode compiler (`make xbbc`), the virtual machine (`make xbvm`), and the X11-Basic to C translator (`make xb2c`). If you need the separate libraries you can do a `make x11basic.a` and a `make libx11basic.so`. These libraries are for example needed by the compiler `xbc`.

If you want to make a version which uses the framebuffer (instead of the X-Server) do a `make fb`. If you want a version using the SDL library, do a `make sdl`.

The TomTom distribution can be generated with `make TomTom`. (The ARM-Linux cross-compiler is needed).

The MS WINDOWS distribution can be generated with `make windows`. (The mingw cross-compiler is needed).

Support

If you have trouble with X11-Basic, you may send me a mail. Please understand that I need to find time to answer your mails. On `http://sourceforge.net/projects/x11-basic/` there is a forum (bug reports, patches, request for help, feature requests) about X11-Basic. You can as well place your questions there, so that also other users of X11-Basic have a chance to help. It is also worth browsing through the topics. Maybe someone has already found a solution to your problem. It is as well ment for the users to share their experience with other X11-Basic users.

If you have trouble with some X11-Basic command or program, and you think it is a bug in the X11-Basic interpreter or compiler itself, 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 or UNIX, Linux, Android?
- 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!

2.2. Using the X11-Basic Interpreter

There are several ways to start the X11-Basic interpreter depending on the operating system you are using it.

2.2.1. Using the X11-Basic Interpreter under UNIX, Linux

The simplest way is to just start it by the command `xbasic` from a terminal window or a console. Then you can use the interpreter in interactive mode. Just try to enter some X11-Basic commands. The interpreter itself also accepts several options via the command line. Please also read the man-page (`man xbasic`) for more details.

In Ubuntu or Lubuntu you will also find X11-Basic in the start menu. When you select X11-Basic from the start menu, the interpreter should come up in its own terminal window.

X11-Basic as a shell

X11-Basic programs can be executed 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 interpreter `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.bas`. After that your program can simply be executed from your shell and the interpreter works in the background like shells do. You need not even use the extension .bas for your scripts.

Example: draftit

A tool to stamp a postscript file with "draft" on every page.

```
#!/usr/bin/xbasic
i=1
WHILE LEN(PARAM$(i))
  inputfile$=PARAM$(i)
  INC i
WEND
CLR flag,count
IF NOT EXIST(inputfile$)
  QUIT
ENDIF
OPEN "I",#1,inputfile$
WHILE NOT EOF(#1)
  LINEINPUT #1,t$
  IF count=3
    PRINT "%% Created by draftit X11-Basic (c) Markus Hoffmann from "+inputfile$
  ENDIF
  IF GLOB(t$,"%%Page: *") AND NOT GLOB(t$,"%%Page: 1 1*")
    IF flag
      PRINT "grestore"
```

```
    ENDIF
    flag=1
    PRINT t$
    PRINT "gsave"
    PRINT ".80 setgray"
    PRINT "/Helvetica-Bold findfont 140 scalefont setfont"
    PRINT "0 80 800 { 306 exch moveto"
    PRINT "(Draft) dup"
    PRINT "stringwidth pop 4 div neg 0 rmoveto 6 rotate show } for"
    PRINT "grestore"
  ELSE
    PRINT t$
  ENDIF
  INC count
WEND
CLOSE
QUIT
```

2.2.2. Using the WINDOWS Version of X11-Basic

You should have installed the package `X11-Basic-1.24-1-win.zip` by extracting all files and invoking the setup program (`setup.exe`). This installs X11-Basic into a folder `C:\\x11basic`. All files you need for using X11-Basic are located there:

```
lib           -- empty folder for future use
bas.ico       -- the icon for .bas files
demo.bas      -- one of the example programs
readme.txt    -- short description of X11-Basic
SDL.dll       -- the Simple Direct Media Library
setup.exe     -- Installation and uninstall program
x11basic.ico  -- another X11-Basic icon
X11-Basic.pdf -- The X11-Basic User Manual
xb2c.exe      -- bytecode to C translator
xbasic.exe    -- The X11-Basic interpreter
xbc.exe       -- The X11-Basic compiler
xbvm.exe      -- The virtual machine
```

X11-Basic can be invoked in the following three ways:

1. Choose "X11-Basic" from the start-menu: You can choose between

   COMPILER : opens the compiler Application which then asks for a .bas file to compile into .exe,

   DEMO : Opens and rund the demo.bas example program,

DOCU : Opens the X11-Basic User Manual,

X11-Basic : Opens the X11-Basic interpreter. `xbasic.exe` will come up with a console window and the interpreter waits for commands to be typed in right away.

2. Click with the right mouse button 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.

The compiler has a rudimentary graphical user interface, which will ask for the .bas file to be compiled and later for the name of the executable to be written to.

By default, the WINDOWS or DOS console does not support ANSI/VT100 coding. So `PRINT AT()` and line editing will probably not work. To fix this, ANSI.SYS has to be installed and switched on for the console windows. Instructions how to install `ANSI.SYS` can be found on the internet. (Also an alternative extension named ANSICON can be used.)

The Context Menu

Every icon unter WINDOWS offers a contect menu when you click on it with the right mouse button. Clicking on an icon of a X11-Basic program as well opens this context menu with following options:

Execute will invoke the X11-Basic interpreter to execute your program. The same happens, if you doubleclick on the icon.

Edit invokes *notepad*, allowing you to edit your program.

View docu opens a window which shows the embedded documentation of your program if there is any. Embedded documentation within a .bas file are comments, which start with a double comment character (`##`).

2.2.3. The Android Version of X11-Basic

A version of X11-Basic ready to be installed on Android smartphones and tablets is available on the *Android Market* (also called *Google Play*).

Unlike the other versions of X11-Basic, the interpreter and virtual machine is embedded in a little IDE (=Integrated Development Environment) which allows the user to load, run, edit and compile the programs.

The app registers itself as a viewer to .bas and .b files on the system. So from any file browser, basic programs can be started with a single touch.

If you open the X11-Basic app itself, you can directly type in commands with the virtual keyboard. Pressing the MENU button gives you the option to load and run BASIC programs, stop and continue execution, open the keyboard (if its has vanished from the screen) and compile basic programs into bytecode. The virtual machine is integrated, so bytecode compiled code can be run. Depending on the endianess of the processor architecture of the platform, bytecode may or may not be compatible with those produced on a Linux PC or WINDOWS machine. Standard output is rendered directly into the graphics screen with a VT100 compatible terminal emulation. Not all graphics features have the same result than on a X11-Windows installation, the whole screen counts as a single fullscreen window. Finally shortcuts to X11-Basic programs can be placed on the desktop, so they can be started with one click. Also X11-Basic is registered as a method to open files (from a file browser). A small selection of example programs is included in the Android package. If you like to have some fun with a game, try `ballerburg.bas`.

Usage on Android devices

Android devices usually have a BACK button, a HOME button and a MENU button.

- The HOME button suspends X11-Basic and returns to the Android desktop. Selecting the X11-Basic app again will resume it. If a BASIC program was running, it will continue to run in the background.
- With the BACK button, a running BASIC program will be stopped. If you press the BACK button again, the X11-Basic interpreter quits.
- The MENU button opens a menu with following options: About, LOAD program, RUN program, STOP/CONT program, NEW, Keyboard, Paste from clipboard, Info/Settings, Editor, Compile and Quit.

  About shows information about the current version of X11-Basic, news and impressum.

  Load ... opens a fileselector which displays all .bas and all .b programs in the directory `/mnt/sdcard/bas`. The selected program will be loaded into memory. A program eventually stored there before will be overwritten. You can display the sourcecode by entering `LIST`.

Run will simply start the execution of a program which has been loaded before. (You can also enter RUN)

STOP/CONT will interrupt the execution of the program or resume it. (you can also press the BACK button once to stop the program, and you can enter `CONT` to continue it).

New will delete the currently loaded program from memory.

Keyboard will show or hide the on-screen virtual keyboard. If you have a hardware or external USB/Bluetooth keyboard, you can also enter commands with that.

Paste from Clipboard will paste any text you have copied to the clipboard (from any other application) before.

Info/Settings will open a dialog with additional information, links, and preference settings. The preferences can be set as follows:

Show splash screen at X11-Basic start-up. This can be switched off here.

Select screen focus. When the screen will be partially covered by the on-screen virtual keyboard, you can specify which portion of the screen should be visible: The top portion, bottom portion, the whole screen but scaled to fit, the portion with the text cursor in it, or the portion with the mouse pointer in it. The default is: scaled.

Select font size. If the screen is small, but the resolution is high, you may want to change the font size to LARGE. This setting affects the console font (text mode) as well as the graphics/user-interface appearance.

Show title This can be switched off here.

Show status bar This can be switched off here.

Show keyboard at start This can be switched off here.

Editor will execute a 3rd-party text editor (e.g. *Ted* or *Jota* or *920 Text Editor* if installed) to edit the program currently loaded. If no program was loaded, the default file name will be `new.bas`. After having saved and closed the text editor, the modified program will be automatically reloaded into the X11-Basic interpreter.

Compile will compile the basic source code into bytecode which can be executed about 20 times faster (but cannot be edited or merged anymore). The bytecode will be saved with .b extension in the bas/ folder.

Help will open a window in which you can search the command reference.

Quit will terminate the X11-Basic interpreter.

Editing a program

If you want to edit an existing program, do following steps (in this example, the editor used is TED, but it works similar with Jota or many other text editors.):

1. Load an existing program with Menu –> Load,
2. choose Menu → Editor to edit the program,
3. finish editing (and save it in the editor). Leave the editor by choosing EXIT in the menu or by using the BACK button (do not use the HOME button).
4. The program gets automatically reloaded,
5. choose menu → run to run it.

If you want to create a new program, follow these steps (in this example, the editor used is TED):

1. Do a MENU -> New
2. Do a MENU -> Editor. The editor will be excecuted with the default file name (new.bas). If you have more than one editors installed, you will be asked which one to use. Select TED Text Editor.
3. Inside the editor do a "Save As" and give it a different name, e.g. "mything.bas", make sure that it is saved into the folder "bas".
4. Press the back button (not the HOME button), so the editor returns to X11Basic.
5. X11-Basic now reloads new.bas, but this is not what you want, so
6. within X11-Basic load "mything.bas"

The next time you edit it, it has the correct name, and a regular save in the editor should do as well as automatic reload in X11-Basic.

If you get an error when calling the text editor, you need to install one. There are plenty around, e.g. *920 Text Editor* or *Ted (tiny text editor)*. Install them from the Android market. You can install multiple editors. Then you are asked which one you like to use every time you call the editor.

LOAD file select functions

To load a program, press menu –> load. You can now select a program file (either .bas or .b) to load. If you touch the filename long you get another menu with advanced functions:

LOAD – load the program.
MERGE – merge the program to the one already loaded (works only with .bas files).

LOAD + RUN – load the program and immediately run it.
LOAD + LIST – load the program an list it.
LOAD + edit – load the program and immediately start the editor.
LOAD + compile – load the program and compile it.
compile + RUN – compile the program and immediately run the compiled program.
delete – delete the selected file (you will be asked to confirm).
CANCEL – return to the file menu.

These functions are here for convenience only. You probably want to use LOAD+RUN or compile+RUN more often.

Running in the Background

When a program is running and you press the home button, the program will continue to run in the background. If you select X11-Basic app again, it brings up the screen output.

Also: When you rotate the screen the running program should continue to run. It needs to find out by using `GET_GEOMETRY` if the screen size has changed.

Desktop shortcuts

You can create desktop shortcuts to your BASIC programs. You can place an application shortcut on the home screen by simply pressing anywhere (and hold for 1 second) on the background of the desktop screen (on Android 4.x devices go to Apps → Widgets). You first are asked to place the shortcut somewhere on the desktop. The X11-Basic launcher then asks for a .bas or .b file and places the link on the desktop. Pressing this link will automatically load X11-Basic and the .bas program and run it.

You can select any file from the `/sdcard/bas` folder which then is placed in the desktop.

Updates of example programs

The X11-Basic app comes with a small selection of example programs. They are copied into the `/mnt/sdcard/bas/` directory. The X11-Basic app will never overwrite a file in `bas/` which is already there. If you want a specific example program be updated (replaced with a potentially newer version, which has come with an update of the X11-Basic app), simply delete the file. It will be restored after the next execution of X11-Basic.

Troubleshooting the Android Version

SCREEN REFRESH PROBLEM: (Was reported sometimes on Samsung Tabs, all Android versions) e.g. galaxy note 1, Android 4.1.2: **Symptoms:** Running the X11-Basic app, the screen output is not updating or refreshing while X11-Basic runs a program. **CURE:** you should check the system settings:

```
Developer settings -->  deactivate Hardware overlays: ON
                   -->  force Gpu: OFF
```

Characters typed are not visible If the whole line appears after you pressed ENTER, but you like to see what you are typing, you need to modify the settings of the keyboard (switch off auto-completion and anything like that, which may make the keyboard hold text back until you press enter.) If still nothing appears after ENTER, then you probably have the Screen Refresh Problem (see above).

2.2.4. The TomTom Version of X11-Basic

On `http://www.opentom.org/X11-Basic` you will find a version of X11-Basic which has been specially compiled for TomTom navigation devices. They run Linux based on the ARM processor. A ready made package as well as installation instructions can be found on that web-site. Currently only versions 1.14, 1.15 and 1.18 of X11-basic are available as a ready-made binary package. Since the new versions of TomTom devices do not allow to install any third party apps anymore, the support for TomTom has been given up (in 2011).

2.2.5. Command line parameters

If you are using X11-Basic under Android, you can skip this section.

The X11-Basic interpreter `xbasic` can be evoked with additional but optional command line parameters. It takes the following ones:

xbasic <filename>	run Basic program `[input.bas]`
`-l`	load only, don't execute
`-e <command>`	execute basic command
`-eval <expression>`	evaluate numerical expression
`-daemon`	switch off prompting and echoing
`-h -help`	print a short help

`-help <topic>` print help on a specific topic

Examples:

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

X11-Basic as daemon

The command line 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. This is useful, if you want to run X11-Basic programs as a background service.

2.3. Editing X11-Basic programs

X11-Basic programs (source code, `.bas` files) are regular ASCII files and therefore can be created with any text editor available.

Users of UNIX like operating systems are fine with every text editor. Simple ones like `pico` or `nano` will do. MS-WINDOWS user can use the simple *notepad* text editor.

Users of X11-Basic under Android need to install a good text editor. TED (Text Editor), `920 Text Editor`, or `Jota` will work fine. Other text editors which might have been already preinstalled can be a source of frustration and trouble. So if unsure, please installe one of the mentioned editors from the Android Market. If you have installed more than one editor, this is no problem, you will be asked which one to use every time, the editor is invoked.

Besides from the basic editing features I recommend to use a text editor with syntax highlighting. Currently X11-Basic syntax definitions are available for the *Nirvana Editor* (`nedit`, available for Linux, UNIX and WINDOWS) and for the `920 Text Editor` and `Jota`, available for Android.

X11-Basic can support foreign language characters. Therefor the basic program may be coded in UTF-8 which is compatible to ASCII but has the ability to use

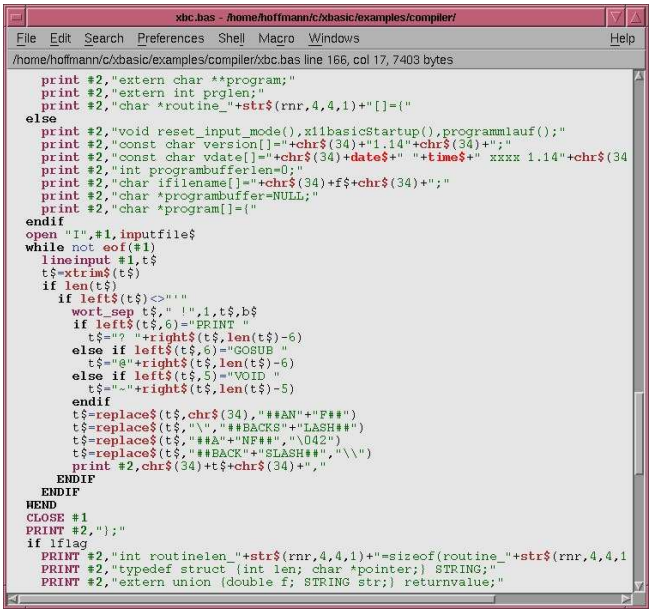

Figure 2.1: The Nirvana Editor with syntax highlighting for a X11-Basic program.

and encode any Unicode character. Such characters can be used in X11-Basic string constants, but may not be used in variable names. Currently only the standard output (console) supports the full UTF-8 character sets.[1]

Using syntax highlighting with nedit

NEdit, the full featured, plain text *Nirvana editor*[2] is a GUI style text editor for workstations with the X Window System. Also a MS Windows port is available[3]. NEdit provides all of the standard menu, dialog, editing, mouse support, macro extension language, syntax highlighting, and a lot other nice features (and extensions for programmers). In short, it has everything you want to develop your X11-Basic programs. Unfortunately nedit does not support UTF-8.

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).

[1] LTEXT will accept some of the special characters (currently only german), TEXT will work with UTF-8 only on Android devices (all latin, greek, cyrillic).

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

[3] `http://nedit.gmxhome.de/winport.html`

2.4. The Bytecode Compiler and the Virtual Machine

If you are using the Android version of X11-Basic, you can skip this chapter. All you need to know is that there is the option to compile X11-Basic programs (to bytecode) which makes them run much faster.

Under UNIX, Linux and Windows a seperate program need to be used to compile .bas files and make bytecode files or standalobe .exe files out of it.

If you are using WINDOWS, the most convinient way to compile X11-basic programs is to execute the compiler `xbc.exe` which has a little use interface. Also under UNIX/Linus it is very convinient to use the compiler manager `xbc` with appropiate command line options (watch out for the `-virtualm` option).

Advanced users probably want to deal with the bytecode filed produced in the compiling process. For each compilation step there are separate programs which do it; namely: `xbbc`, `xb2c` and `xbvm`.

`xbbc` compiles X11-Basic programs (.bas files) to bytecode files (.b). `xb2c` can translate bytecode files to C source code. `xbvm` is a virtual machine (interpreter for bytecode).

The idea is to increase the execution speed of X11-Basic programs a lot by compiling it to a bytecode, this still being portable. The bytecode itself is interpreted by a bytecode interpreter (also called a virtual machine). This virtual machine needs to be present on the target computer, and then all bytecode programs can be used there. This way, the X11-Basic compiler need not deal with different target machine architectures, and also the bytecode can be run much faster than the interpreted BASIC source code.

The conversion to bytecode is a real compilation. The step to assembler or machine code is not far. Also a translation to C or to JAVA or any other language will be straight forward. As with JAVA, the bytecode is platform independent and can be run on any system, which has a virtual machine ported to.

Also one point to mention (whether this is a feature or a disadvantage): X11-Basic bytecode can not be converted back into BASIC source code (.bas), but is rather a very abstract representation of your program.

If you want to get a feeling on what this is about, open a .c source file, which has been produced by the bytecode to C translator `xb2c`. Implemented with an additional macro translation step, the bytecode is in a way readable. Here is an example:

```
...
    PUSH2;              /* 2  */
    ZUWEIS(2);          /* I= */
LBL_38:  PUSHV(2);      /* I */
```

```
    X2I;
    PUSHARRAYELEM(3,1); /* F(.) */
    X2I;
    JUMPIFZERO LBL_91; /* JEQ(0x91); */
    PUSH2;              /* 2 */
    PUSHV(2);           /* I */
    EXCH;
    X2F;
    MULf;
    PUSHV(0);           /* S */
    LESS;
    JUMPIFZERO LBL_81; /* JEQ(0x81); */
    PUSH2;
    PUSHV(2);           /* I */
    EXCH;
    X2F;
    MULf;
    ZUWEIS(5);          /* K */
LBL_61:  PUSHV(5);      /* K */
    X2I;
    PUSHVVI(3,1);       /* F */
    PUSHCOMM(30,1);     /* CLR */
    PUSHV(5);           /* K */
    PUSHV(2);           /* I */
    ADD;
    DUP;
    ZUWEIS(5);          /* K */
    PUSHV(0);           /* S */
    GREATER;
    JUMPIFZERO LBL_61; /* BEQ_s(-29); */
    PUSHCOMM(74,0);     /* FLUSH */
LBL_81:  PUSHX("I");
    PUSHLEER;
    PUSHCOMM(147,2);    /* PRINT */
    PUSHVV(4);          /* C */
    COMM_INC;           /* INC */
LBL_91:  PUSHV(2);      /* I */
    PUSH1;
    ADD;
    DUP;
    ZUWEIS(2);          /* I= */
    PUSHV(0);           /* S */
    GREATER;
    JUMPIFZERO LBL_38;  /* BEQ_s(-104); */
...
```

This is bytecode made out of the (X11-Basic) lines:

```
...
FOR i=2 TO s
  IF f(i)
    IF 2*i<s
      FOR k=2*i TO s STEP i
```

```
        CLR f(k)
      NEXT k
      FLUSH
    ENDIF
    PRINT i,
    INC c
  ENDIF
NEXT i
...
```

You are not suuposed to understand any of these, but it may give you a feeling about what bytecode really is, and that is really hard to reconstruct the original BASIC lines out of it.

Please try the bytecode compiler out and maybe you want to report errors etc. Quite a lot of the example programs are known to work well with the bytecode compiler: e.g. `mandel-simple.bas`. The bytecode will execute about 10 times faster than the interpreted program. Here is how to use it:

```
xbbc myprogram.bas -o b.b
xbvm b.b
```

2.5. Using the X11-Basic to C translator

It is possible to translate the bytecode generated by `xbbc` to C source code and finally compile this intermediate C-source to a native executable (e.g. with the GNU C compiler `gcc`). This way the program will be a real native executable which –again– runs even a bit faster that the bytecode interpreted by the virtual machine.

Such programs can be linked against the dynamic library (.so or .dll) or the static library (.a or .lib). In the end they run independently of any interpreter or virtual machine. However, some restrictions to the code apply. Which means: not every program, which can be interpreted, can also be compiled.

The generated C-sources depend on the header file `xb2csol.h` (normally installed under `/usr/include/x11basic/`) the `x11basic.a` or `libx11basic.so` libraries, which therefore should be present.

`xb2c` processes one input file. The suffix of the input file is usually `.b` (which should be a bytecode file produced by `xbbc`). The default output file name is `11.c` but you can specify alternate names with the -o option.

Actually `xb2c` is not a real compiler, but rather a translator. The compilation is already done by the bytecode compiler. `xb2c` itself does a one to one translation of the bytecode (currently only into C). This translation process is not yet highly

optimized, but quite robust and portable. There is no way to recreate the .bas source code from the .c file. But still the C file is platform independent and can be compiled on all platforms, where a C compiler is available (and the x11basic library is ported to).

Here is how to use it (examples are under linux):

```
xbbc myprogram.bas -o b.b
xbvm b.b
xb2c b.b -o 11.c
gcc  11.c -lm -lX11 -lx11basic -lasound -lreadline -lgmp -llapack -o a.ou
```

For convinience, a

```
xbc -virtualm myprogram.bas -o a.out
```

will exactly do the same.

2.6. The X11-Basic compiler manager xbc

The X11-Basic package is shipped with the X11-Basic compiler `xbc`, which makes stand-alone binaries out of X11-Basic source code. It also can produce `.o` object files, shared objects (or DLLs) and bytecode.

There are three methods on how the compilation can be done:

1. **The pseudo method:** The sourcecode is bundled together with the X11-Basic interpreter into one executable file, which can be run. Execution speed is not faster than the interpreted source code, but all programs will run and behave exactly the same as if they were run in the interpreter. Currently this method is not available for WINDOWS since `gcc` is used to do the compression and linking with the X11-Basic runtime library. This is the default on UNIX and Linux operating systems.
2. **The bytecode method:** The sourcecode is compiled into bytecode and this bytecode is bundled together with the X11-Basic virtual machine into one executable file, which can be run. Execution speed is much faster than the interpreted source code. However, some restrictions to the compiled sourcecode apply, e.g. GOTOs across procedures are not possible, as well as ON ERROR and ON BREAK will currently not work. So some obscure code will probably not compile correctly. However, this method is recommended as the preferred method and it is the default on MS WINDOWS.

3. **The independent method:** The sourcecode is compiled to bytecode and then translated to C sourcecode, which finally will be compiled using a C-Compiler (e.g. GNU gcc) or a cross-compiler. This is the preferred method on UNIX systems (although it is not the default) where a development environment (gcc and development packages for libraries) is available. On WINDOWS this is usually not the case, so method 3 can not be used. On Ubuntu Linux you will need to install at least following packages: `gcc, libreadline-dev, libasound-dev, libgmp-dev, liblapack-dev` and maybe others. If done so, the compiler with method 3 will work fine.

To select method 3 on UNIX/Linux systems, use the command line option `-virtualm`. The windows version of the compiler will automatically use method 2 only.

The compiler `xbc` itself is written in X11-Basic and relies on the presence of `xbbc` and `xv2c` (for methods 2 and 3). You can find the compiler in `examples/compiler/xbc.bas`. Yes, the compiler compiles itself. Just 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
```

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

2.7. The ANSI-Basic to X11-Basic converter

X11-Basic packages come with a simple ANSI-Basic to X11-Basic converter `bas2x11basic`. (The sourcecode `bas2x11basic.bas` of the converter can be found in the `examples/compiler` directory.) It helps converting old (real) Basic Programs with line numbers and multiple commands per line to the X11-Basic structure. Because there are so many different BASIC versions around, 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. For details on the compatibility to other dialects of BASIC, please read chapter 9.

Example:

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

For further options try

```
xbasic bas2x11basic.bas --help
```

and read the man-page `man bas2x11basic`. If you like to improve the converter please feel free to do so. You may want to send me the result.

2.8. Using GFA-BASIC programs

GFA-Basic programs have a tokenized binary format and usually the suffix `.gfa`. This binary format has to be decoded to ASCII files before they can be used with X11-Basic. This job is done by the utility `gfalist` (sometimes also called `gfa2lst` or `ons-gfalist`) by Peter Backes[1].

The resulting GFA-Basic programs usually need some manual corrections. Very simple ones may well work fine with X11-Basic without. For details on the compatibility, please read chapter 9.2.

[1] `http://titan.plasma.xg8.de:8080/~rtc/`

3 Programming in X11-Basic

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

3.1. The dialect of X11-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 designed 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 developed for the ATARI ST in 1985. GFA BASIC (as of version 3.5, the most popular one) was, by the standards of its time, a very modern programming language. Like X11-Basic, it does without line numbers and has a reasonable range of structured programming commands.

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 computers since the mid 1980's:

- One command or declaration per line for better readability,
- use of subroutines (procedures) and functions with local variables and parameter passing by value or by reference,
- data statements and arrays,
- powerful loop and program flow constructs,
- file and socket operations,
- complex number mathematics,
- operations for handling arbitrary/infinite precision numbers,
- commands to directly access the operation system shell,
- commands for using graphics in multiple windows,
- a port of the AES (the graphical user interface from the ATARI ST), allowing for easy use of graphics in your program,
- commands for direct memory manipulation, allowing you to access the computer almost as with machine language,

- possibility to merge source code for libraries and reuse,
- inline data compression and encryption (disabled in US-versions),
- Unicode (UTF-8) support,
- powerful mathematics (including matrix/equation solving and fast Fourier transformations), and
- a compiler is available.

Interpreter and 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 bytecode or machine code resulting in a more speedy program execution as the step for command look-up does not appear anymore. The compiled program just can be executed out of the box. On the other hand, 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, once testing has been finished.

3.2. Getting started

To write a first X11-Basic program you will need an editor, where you can type in the source code. The X11-Basic package does not include an editor, but many so-called text editors are readily available nearly everywhere and by chance they are already installed on your system. You can use *Notapad2* on MS WINDOWS systems, *pico, nano, vi, emacs, nedit, gedit* and many more on UNIX and Linux systems, *pico* on the TomTom device, *Ted* or *920 text editor* on Android. This is just a small list of possibilities here.

Open such an editor, and you can start programming.

3.3. Your first X11-Basic program

We assume, that you have opened a console window (a shell) on linux or WINDOWS. The Android version is a bit different.

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 additional 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 prompt when done.

Now lets compile it:

```
xbbc hello.bas -o hello.b
```

will produce a bytecode binary `hello.b`.

You can run this:

```
xbvm hello.b
```

will give you the same output "Hello X11-Basic!".

Real compilation will need two more steps:

```
xb2c hello.b -o hello.c
```

produced a translated C-sourcefile `hello.c`.

If you have the *gnu C compiler* available you can compile it to an independent executable program called hello with:

```
gcc hello.c -o hello -lm -lX11 -lx11basic -lasound -lreadline
```

There you go. Your program can now directly be started with

```
./hello
```

3.4. Program structure

If you want to write more sophisticated programs than the Hello-example, you should understand the general structure of a X11-Basic program.

A 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 program this is all 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.

3.5. General Syntax

The syntax of a typical X11-Basic line is

```
COMMAND parameters
```

parameters usually consists of a list of comma separated expressions. Another type of X11-Basic lines are variable assignments

```
variable=expression
```

variables typically have a name and can have different types. The result of the expression will be stored under that name for further reference. Each line of X11-Basic code can contain exactly one command or one assignment (or a comment).

Here is a typical piece of X11-Basic code:

```
LOCAL l,ll,content$,g$,gg$,comp
CLR comp
IF EXIST(f$)
  OPEN "I",#1,f$
  ll=LOF(#1)
```

```
    content$=INPUT$(#1,ll)
    CLOSE #1
  ENDIF
  ' and so on
```

Appending lines

With many editors a limitation on the maximal line length applies (e.g. 4096 characters/line[1]). 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). Therefor a possibility of splitting 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,"; \
 " that's it"
```

will be treated as:

```
PRINT "Hello,";" that's 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.

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

[1] Note, that in X11-Basic itself there is no limitation on the line lengths.

Example:

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

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

3.6. 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.

Example:

```
PRINT "The result of 1+1 is: ";1+1
```

With the **INPUT** command you let the user input data, p.ex. numbers or text. The data can be entered on the text screen/console window. Together with PRINT this allows already to implement a very simple user interface.

Example:

```
INPUT "Please enter your name: ",name$
PRINT "Hello ";name$
```

The **IF** command let the program do different things depending on the result of a calculation. Therefor the code is grouped into a block which should only be executed if the result of the expression after IF is TRUE (this means, not zero). The block starts with the IF command and ends with an ENDIF command. If the result of the expression after IF is not TRUE, means it is FALSE (or zero), the program will be continued after the ENDIF and the lines of code between the IF and the ENDIF are not run.

Example:

```
INPUT "Please enter a number: ";a
IF a=13
  PRINT "Oh, you obviously like the thirteen!"
ENDIF
PRINT "Thank you for the ";a;"."
```

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. Since no line numbers are used, you must use a label to define lines where the GOTO command can jump to.

Example:

```
again:
INPUT "Please enter a number, but not the 13: ";a
IF a=13
  PRINT "Oh, you obviously like the thirteen!"
  PRINT "But, please enter a different number."
  GOTO again
ENDIF
PRINT "Thank you for the ";a;"."
```

Besides these four very basic commands (which exist in every BASIC dialect) X11-Basic has many more features which make life easier and your programs more user friendly.

3.7. Variables

Variables in BASIC programming are analogous to variables in mathematics. Variable identifiers (names) 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.

Example:

```
a=1     ! Assigns a 1 to a variable named a
b=a+1   ! The variable a can be referred to, to make a calculation
PRINT "The variable b contains now a ";b
```

Variable names can be very long if you like and also can contain digits and an underscore, with the exception, that the first letter of the variable name must not be a digit.

Example:

```
my_very_long_variable_name=1.23456
PRINT my_very_long_variable_name
```

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 line of X11-Basic code will create a variable called age for you and assign it the value of 18.

```
age=18
```

But if you want to store a text, the variable needs to be capable to hold characters instead of numbers. Therefore, in that case you mark the variable with a $ to tell that it should store text, not numbers:

```
name$="Robert"
```

Text constants, by the way, need to be enclosed with "", to tell X11-Basic that the text should not be interpreted as program code, but just be treated as any text.

The assignment is done with the '=' operator. The '=' operator is also used in expressions, e.g. after an IF command. But there it is not used as an assignment operator but instead there it is treated as a comparison operator. In X11-Basic both operators are '='. The interpreter distinguishes between them just by context.

Example:

```
x=(a=3)
```

Here the first = is an assignment and the second is the comparison operator. x will be assigned a -1 (TRUE) if a is 3 and a 0 (FALSE) else. The brackets are not necessary here they just help reading this expression. Confused? Well, you eventually will get used to it.

Last to say, 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.

3.7.1. The scope of a Variable

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. Unless otherwise declared with LOCAL, all X11-Basic variables are global by default and this need not be explicitly declared.

But there is one downside of global variables: 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 scope variable. Also recursive programming is nearly impossible with only global variables, last but not least, the usage of procedures and functions becomes much more clear, if you are able to encapsulate all internal variables of that function and you do not bother outside of the functions scope if you accidentally use one of these internal variables somewhere else in the code, possibly altering the functions behavior.

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

Local variables need to be declared with the command LOCAL inside the function od procedure where they belong to. Outside this specific procedure or function they simply do not exist, or if a global variable of the same name exists, they refer to different contents.

3.7.2. Data types

Now, lets come back to the type of a variable. How can one see what kind of content a variable can store? How does X11-Basic know? 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: 32bit integer variables have the suffix %, arbitrary precision integer variables have a &, complex variables a #, character strings a $, arrays a (). Variables without suffix are treated as real 64bit floating point variables. 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 (even if the rest of the name is identical) without.

Examples:

```
x=10.3         ! this is a number variable (64bit floating point)
x$="Hello"     ! this is a different character string variable
x%=5           ! this is a (32bit) integer variable, still different
x&=79523612688076834923316 ! this is a big integer variable, still diffe
x#=3+4i        ! this is a complex number variable,
@x             ! this refers to a function or procedure called x
x()=[1,2,3,4] ! This beast refers to an array.
```

3.7.3. Variable naming

You can use all letters and numbers for your variable names. Spaces are not allowed but underscores inside the variable name. The variable name can be of any length. 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 cause troubles. 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, z#   .
```

Invalid variable names look like this and X11-Basic will complain:

```
_blank, 1x, ?value%, 5s$, 1i, #u.
```

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$` is a whole different variable(value) than `A` which is different from `A%` or `A$(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$)`
	`b%=MALLOC(100000)`
big integer variables:	`i&=79523612688076834923316`
	`a&=FACT(100)`
float variables:	`a=1.2443e17`
	`b=@f(x)`
complex variables:	`a#=1.2443e17+1.2i`
	`b#=CONJ(a#)`
character strings:	`t$="Hello everybody !"`
fields and arrays:	`i%(),a(),t$(), [1,3,5;7,6,2]`

3.7.4. Numbers

X11-Basic normally uses integer numbers (32 Bit) which range from -2147483648 to 2147483647, and floating point numbers, which are 64Bit IEEE 754 standard values. These 64bit floating point numbers have a mantissa of of 52 bits and an exponent of 11 bits and a sign bit. These numbers can represent 15 to 16 significant digits and

powers of 1e-308 to 1e308. Complex numbers consist of two 64bit floating point values.

X11-Basic currently also support infinite precision integer numbers. These numbers are stored in a variable size portion of memory, so that an arbitrary number of digits can be stored. However, calculation with big interges is slow and only a few built-in functions can be used on them.

Number (constants) can be preceded by a sign, + or -, and are written as a string of numeric digits with or without a decimal point and can also have a positive or negative exponent as a power of 10 multiplier e.g.

```
-253  67.3  0.25  -127.42E-3  -1.3E7  1
```

The imaginary part of complex number constants are market with a trailling "i", e.g.

```
-2i  1i  0.25+3i     -127.42E-3i
```

Note: A single "i" is always treated as a real variable name. If you want the imaginary unit, please always use "1i".

Integer numbers, with no decimal fraction or exponent, can also be in either hexadecimal or binary. Hexadecimal numbers should be preceded by $ (or 0x) and binary numbers preceded by %, e.g.

```
%101010     -$FFE0    0xA0127BD     -%10001001     %00011010
```

3.7.5. Strings

String variables can contain sequences of characters (bytes) of arbitrary length. There is no length limit for a string other than the virtual memory of the machine. Strings generally contain ASCII text, but can hold arbitrary byte sequences, even characters that have the ASCII code zero. In other words a string is a collection of bytes of certain length. You can treat strings as arbitrary length of binary data if you need. 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.

String variables are distinguished by the $ suffix.

String constants are enclosed with pairs of "" (double quote).

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

There is a way to include special characters into string constants. The usual way in BASIC is to split the string into sub strings and concatenate the parts during run time, like in the code fragment:

Example:

```
st$="This is a special string, containing a bell character at the end"+CHR$(7)
```

By the way, the double quote character can be added with `CHR$(34)`.

3.7.6. Arrays

Arrays are memory locations that store many values of the same type at the same time. While normal variables store a single value at a time, an array variable can store many values. The values are accessed via the name of the variable and the appropriate indexes. The index or indexes follow the name of the variable between (and).

There is no limit on the number of indexes (the dimension). You can use as many as you like. Also there is no limit on the index values other than the index values have to be positive integer and that memory may limit the array sizes.

X11-Basic arrays can contain variables of any data type, including strings. All arrays, even multi-dimensional arrays, can be re-dimensioned without altering the contents. A special feature of X11-Basic is the implicit dimensioning of arrays and the existence of array constants. You may define an array by using the DIM command. You might also define the array by an assignment like

```
DIM b(10)
a()=b()
```

if b() already has been DIMed or by

```
a()=[1,2,3,4;6,7,8,9]
```

assigning an array constant. (In this example a 2 dimensional array will be created and the rows are separated by ';'.)

3.7.7. Arbitrary precision numbers

X11-Basic also support infinite or arbitrary precision numbers with a sprecial data type. Arbitrary-precision arithmetic, also called bignum arithmetic, multiple precision arithmetic, or sometimes infinite-precision arithmetic, indicates that calculations are performed on numbers whose digits of precision are limited only by the available memory of the computer. This contrasts with the faster fixed-precision arithmetic, normally used.

Infinite precision math is slow, and not all functions are available for this data type. Arbitrary precision is used in applications where the speed of arithmetic is not a limiting factor, or where precise results with very large numbers are required.

The data type with the suffix & supports (big) integers only. It is up to the user (and straight forward) to write routines for handling rational numbers (using two big interges, numerator and denominator), and corresponding routines for adding, subtracting, multiplication and division of thouse fractions. Multiple precision Irrational numbers using a floiting point representation are (currently) not supported. If somebody needs this, please let me know.

Supported operators (for big integers) are + - * / = <> < > MOD DIV ABS() SQRT() NEXTPRIME() FACT() PRIMORIAL() FIB() LUCNUM() RANDOM() ADD() SUB() MUL() DIV() MOD() POWM() ROOT() GDC() LCM() INVERT() MIN() MAX(). Also STR$(), BIN$(), OCT$() and HEX$() work with big integers.

The advantage is, that you can handle big integer numbers without loosing precision, as it is useful for cryptography and number theory.

Also rounding errors can be avoided by using infinite-precision rational number arithmetic (which is not implemented by X11-Basic itself, but which could be realized using pairs of big integers.)

Variables of either type may be used and mixed in expressions. They will be converted to big integer or from big integer to float or 32bit integers when needed. One should j ust be aware of the eventual loss of precision.

Here is an example how to use big number arithmetrics in X11-Basic to factorize a big number into its prime factors:

Example:

```
' Factorize (big) Integer numers into prime factors.
' with X11-Basic  >= V.1.23
'
DIM smallprimes&(1000000)
```

```
CLR anzprimes
smallprimes&(0)=2
INC anzprimes

INPUT "Enter a (big) number: ",a&
PRINT "Calculating primes up to ";lim&;". Please wait..."
lim&=SQRT(a&)   ! Limit up to which the primes are searchd
FOR i=1 TO DIM?(smallprimes&())-1
  b&=NEXTPRIME(smallprimes&(i-1))
  EXIT IF b&>lim&
  smallprimes&(i)=b&
NEXT i
anzprimes=i
PRINT "calculated ";anzprimes;" primes up to: ";b&

PRINT "Factorization:"
PRINT a&;"=";
FOR i=0 TO anzprimes-1
  WHILE (a& MOD smallprimes&(i))=0
    PRINT smallprimes&(i);"*";
    FLUSH
    a&=(a& DIV smallprimes&(i))
    lim&=SQRT(a&)
  WEND
  EXIT IF smallprimes&(i)>lim&
NEXT i
IF nextprime(a&-1)=a& or a&=1
  PRINT a&
ELSE
  ' The number is too big and we cannot be sure that this is a prime
  PRINT "----incomplete test -----";a&
ENDIF
END
```

Note that the list of small primes could also be generated by a sieve. The method used is based on prime number tests (using the function `NEXTPRIME()`) and may be not optimal.

3.8. 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.

Most of these functions can work on different input data types. E.g. you can use the `SQRT()` function also on complex numbers, thus returning a complex result.

3.8.1. Expressions and Conditions

No difference makes the difference.

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 expressions are treated the same in X11-Basic. Because X11-BASIC doesn't have separate Boolean operators for conditions and expressions, the operators (AND, OR, XOR, NOT) actually operate on binary values. Therefore a TRUE is -1, which means, that every bit is one. So the operators will operate on each of these bits. Such: a condition is considered TRUE if the expression is not FALSE (means the result must be a value other than zero).

3.8.2. 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/power
3	−	sign (negation)
3	+	sign
4	NOT	bitwise not
5	/	divide
5	*	multiply
5	DIV	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	NAND	bitwise not and
10	OR	bitwise or
10	NOR	bitwise not or
10	XOR	bitwise exclusive or
10	IMP	implies
10	EQV	equivalence
11	=	assignment

(*) = not implemented

Addition and subtraction operators are both binary and unary operators. In their unary form they are used out of the precedence orders. Unary operators are always applied first, unless parentheses drive different calculation order.

The power operator `a^b` calculates the b-th power of a. The actual implementation of the power operator always uses the `pow()` function, which always treats all operants as real numbers. Under some circumstances it might be more optimal to use `a*a` instead of `a^2`.

The multiplication operator multiplies the operands. If any of the operands is an array then the result will be an array.

The division operator divides the first operand with the second. If the second operand is zero then an error will occur.

The integer division operator divides the first operand with the second. The calculation is performed using integer numbers and the result is truncated towards zero.

Bit-wise and logical NOT This unary operator calculates the logical negate (the complement) of the operand. The calculation is done on integer numbers, thus the operand is converted to an integer value. The operator inverts each bit of the operand.

Logical operators (AND, OR, XOR) These operators can be used for both logical and bit-wise operations. X11-Basic does not have a separate type for logical values. The logical TRUE is represented as integer value -1 (all bits set to 1) and the logical FALSE is 0. The operators AND, OR and XOR perform the calculation on integer values. If any of the operands is not integer it is converted to integer value before the operation takes place. The operations are performed on each bit of the operands.

Operators for Character Strings

There are a few operations which can directly be done to character strings or string variables, using operators.

plus operator, conjunction The plus '+' operator used on strings, links two strings together.

Example:

```
a$="X11"
b$="-"
c$="BASIC"
d$=a$+b$+c$
```

results in a string `"X11-BASIC"`.

comparison operators, <, <=, =, =>, >, <> comparison functions belong to numerical (Boolean) functions because the result is a number, although they can be used with strings.

Example:

```
IF a$="X11"
  ...
ENDIF
result=(a$<>"Hello")
```

code evaluation operator, & the eval operator evaluates command or expression which is given by the String. Example see below.

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 strings with the greater and smaller operator works character by character until one of them is smaller or one of 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 Evaluation 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.

3.8.3. 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 command `SPLIT` or the function `WORD$()` .

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.

3.8.4. Arrays

Arrays are special variables which consist of many values (of the same type). There can be floating point arrays, integer arrays, string arrays, and arrays of arrays. The memory for an array need to be declared before it can be used. This can be done with the `DIM` statement or by direcly assigning a value to the array.

Array constants

The common way to assign data to a whole array is to put the input figures into list into square brackets (which forms an array constant) and assign this to an array variable like:

```
a()=[1,2,3;4,5,6]
```

A comma is used to separate columns elements, and semicolon is used to separate rows. So [1, 2, 3] is a row vector, and [1; 2; 3] is a column vector.

Now that you know how to define a simple array, you should know how to access its elements. Accessing the content of an array is done through the operator (), with the index inside the parenthesis; the indexing of the first element is 0:

```
b=a(0)
a(1)=5
```

Accessing an element outside the bounds will result in an error: "Field index too large."

To access a single matrix element, you can use the (i,j) subscript, where i is the index in the row, and j in the column:

```
b=a(1,2)
a(3,4)=3
```

It is also possible to access blocks of matrices using the colon (:) operator. This operator is like a wildcard; it tells X11-Basic that you want all elements of a given dimension or with indexes between two given values. For example, say you want to access the entire first row of matrix a above, but not the second row. Then you can write:

```
b()=a(1,:)
```

Now say you only want the first two elements in the first row. To do this, use the following syntax:

```
b()=a(1,1:2)
```

It is also possible to use arrays of any higher dimension.

```
DIM a(10,10,10,10,10)
b=a(2,5,4,2,7)
```

Array operators

Arrays are not only good for storing information in tables, but one can apply operations on arrays. You can for example use the classic arithmetic operations + and - on any array in X11-Basic: this results in the vector addition and subtraction as defined in classic vector vectors spaces, which is simply the addition and subtraction elements wise.

Array Operator	Description
+	Vector/Matrix addition element by element
-	Vector/Matrix subtraction element by element
*	Array/Matrix multiplication
:	Subarray (a block)
=,<>	comparison element by element
<,>,<=,>=	comparison using a norm

Array functions and operators act on entire arrays. Some return a list, which can then either be used as a value for another array function, or assigned into an array variable.

Array comparisons compare the array contents element-by-element, using the default comparison function for the element data type (=,>,<). In multidimensional arrays the elements are visited in row-major order (last subscript varies most rapidly). If the contents of two arrays are equal but the dimensionality is different, the first difference in the dimensionality information determines the sort order.

3.9. Procedures and Functions

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.

3.9.1. 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 shows a 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
```

3.9.2. 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 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 to define

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

instead.

An alternative for FUNCTION is the DEFFN statement, 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. There is another difference between DEFFN and FUNCTION: The compiler will use the DEFFN expression as an inline expression and will not produce a function with a symbol name. This is a bit faster, but produces longer code.

3.9.3. 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.

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 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.

All though a (copy of an) Array can be passed to a subroutine by value, the functions cannot return arrays[1].

If a function needs to return information in form of an array, the return array sould be passed as a `VAR` parameter in the parameter list. The return values can then be assigned to it inside the function.

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
  CLR idx
  WHILE idx<DIM?(n$()) AND n$(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.

You could as well declare the FUNCTION like

```
FUNCTION SearchName(n$,n$())
```

Then a local copy of the whole array Name$() would be used inside the function, any changes to n$() would have no effect on the original array Name$(). But win case you wanted to make changes to the array, like in following example:

```
idx%=@EliminateName("Jack",Name$())

FUNCTION EliminateName(n$,VAR n$())
  LOCAL i
  FOR i=0 TO DIM?(n$())
    IF n$=n$(i)
      n$(i)="deleted."
```

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

```
    ENDIF
  NEXT i
  RETURN i
ENDFUNC
```

you need to use VAR.

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 were global variables.

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

3.10. Simple Input/Output

There are many ways in X11-Basic to take data by the user and display other data. This can be done by taking data from the keyboard, the mouse, a microphone, etc. and by displaying data to the text console, the graphics window, the speaker, etc. Also reading and writing to files and internet connections can be done.

The most basic input and output from and to the user is by using the text console, the so-called standard input and standard output. This is done in X11-Basic like in all BASIC dialects with the basic commands PRINT and INPUT.

3.10.1. 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.

The syntax of PRINT 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 add special commands and functions to your PRINT statement for screen control such as cursor positioning and formatting of the data. 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 10.123 USING "+##.###"
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. They can of course be more complicated and all features can be combined.

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.

3.10.2. 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. You can use the `AT()` statement. 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";
```

The 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 you have depends on the current text 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 columns 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.

3.10.3. Formatting output with PRINT USING

X11-BASIC normally prints numbers in a form convenient for most purposes. But on occasion you may prefer a more elaborate form. For example, you may want to print

financial quantities with two decimal places (for cents) and, possibly, with commas inserted every three digits to the left of the decimal point. Or, you want to print the numbers in scientific notation. `PRINT USING` provides a way to print numbers in this and almost any other form[1].

The generic syntax is

```
PRINT <expression> USING "<format string>"
```

. The result of the expression should be a number. The format string defines how you want your data to be formatted on screen. The format string may be a string variable, a quoted string, or a more general string expression.

`PRINT USING` also allows one to print strings centered or right-justified, as well as left justified.

The function `USING$()` duplicates the PRINT USING statement almost exactly[2] but returns the result as a string rather than printing it on the screen.

Unlike `STR$()`, where you can specify the length of the string, the number of significant digits of the number and a flag, where there leading zeroes should be used, `USING$()` and `PRINT USING` use a classic and BASIC-style formatter string for formatting numbers. The difference between `USING$()` and `PRINT USING` is just, that `PRINT USING` immediately prints out the formatted number and `USING$()` converters it into a string containing the formatted number, suitable for further processing.

Formatting Numbers

The format string can contain any letters, but some have a special meaning. All other characters are just taken as they are. The length of the format string defines the length of the output field. Whatever is formatted, it will exactly take as many characters as the length of the format string.

The most important special character in the format string is the symbol #, which stands for a digit position to be filled with one digit from the number to be formatted. For example, compare the output resulting from two similar PRINT statements, the first a normal PRINT statement and the second employing USING.

```
x=   |PRINT x| PRINT x USING "###"
-----+-------+--------------------
1    | 1     |   1
```

[1] There are also other built-in commands for formatting data on output. X11-Basic offers e.g. STR$(). Please refer to the sections on functions for details on the syntax of STR$().

[2] only numbers can be formatted, no strings.

```
12   | 12    |  12
123  | 123   | 123
1234 | 1234  | ***
-12  | -12   | -12
```

Without USING, the number is printed left justified and occupying only as much space as needed. With USING, the format string `"###"` specifies a field length of exactly three characters. The number is printed right-justified in this field. If the field is not long enough to print the number properly, asterisks are printed instead. If all you need to do is to print integer numbers in a column but with right-justification, then the preceding example will suffice. Note that a negative number will be printed with the sign occupying one of the digit fields.

With printing financial quantities it is conventional that the decimal points are aligned. Also, you may want to print two decimal places (for the cents) even when they are zero. The following example shows how to do this. (In order to print negative numbers and have the sign at a fixed position, the format string should start with a minus sign.)

```
x=    |PRINT x USING "-##.##"
------+----------------------
1     |  1.00
1.9   |  1.90
-3.14 |- 3.14
1.238 |  1.24
123   |******
0     |  0.00
-123  |******
```

Notice that in this example two decimal digits are always printed, even when they consist of zeroes. Also, the result is first rounded to two decimals. If the number is negative, the minus sign occupies the leading digit position or the position given by a - or + in the format string. If the number is too long to be printed properly (possibly because of a minus sign), asterisks are printed instead.

Financial quantities are often printed with a leading dollar sign ($), and with commas forming three-digit groups to the left of the decimal point. The following example shows how to do this with `PRINT USING`.

```
x=          |PRINT x USING "$#,###,###.##"
```

```
----------+-----------------------------
0         |$        0.00
1         |$        1.00
1234      |$    1,234.00
1234567.89|$1,234,567.89
1e6       |$1,000,000.00
1e7       |10,000,000.00
1e8       |************
```

The dollar sign is only printed if the space is not needed for a digit. It is always in the same position (first) in the field. The separating commas are printed only when needed.

In case you want the dollar sign ($) to float to the right, so that it appears next to the number, avoiding all those blank spaces between the dollar sign and the first digit in the preceding example. The following example shows how to do this.

```
x=        |PRINT x USING "$$,$$$,$$#.##"
----------+---------------------------
0         |        $0.00
1         |        $1.00
1234      |    $1,234.00
1234567.89|$1,234,567.89
```

The format string can also allow leading zeroes to be printed, or to be replaced by asterisks (*). You might find the latter useful if you are preparing a check-writing program.

```
x=        |PRINT x USING "$0,000,000.##"
----------+-----------------------------
0         |$0,000,000.00
1         |$0,000,001.00
1234      |$0,001,234.00
1234567.89|$1,234,567.89

x=        |PRINT x USING "$*,***,***.##"
----------+-----------------------------
0         |$********0.00
1         |$********1.00
```

```
1234       |$****1,234.00
1234567.89 |$1,234,567.89

x=         |PRINT x USING "*$$,$$$,$$#.##"
-----------+-----------------------------
0          |*********$0.00
1          |*********$1.00
1234       |*****$1,234.00
1234567.89 |*$1,234,567.89
```

For compatibility reasons, a % can be used instead of the 0's in the format string, with one exception: The first character in the format string must not be a %[1].

You can also format numbers using scientific notation. Because scientific notation has two parts, the decimal-part and the exponent-part, the format string must also have two parts. The decimal-part follows the rules already illustrated. The exponent-part consists of from three to five carets (^) that should immediately follow the decimal-part. The following example shows how.

```
x=          |PRINT x USING "+#.#####^^^^"
------------+---------------------------
0           |+0.00000e+00
123.456     |+1.23456e+02
-.001324379 |-1.32438e-03
7e30        |+7.00000e+30
0.5e100     |+5.00000e+99
5e100       |************
```

The leading plus sign (+) in the format string guarantees that the sign of the number will be printed, even when the number is positive. Notice that the last number cannot be formatted because the exponent part would have been 100, which requires an exponent field of five carets. Notice also that if there are more carets than needed for the exponent, leading zeroes are inserted. Finally, notice that trailing zeroes in the decimal part are printed.

In addition to the format rules explained above, X11-Basic offers another but different set of format strings. If the first character of the format string is a % the format string is treated as a C style so-called printf-formatter.

Here are some examples:

[1] If the first character is a % the format string is interpreted as a C style printf format string (see below).

```
x=          |format$=     |PRINT x USING format$
------------+-------------------------------
0           | "%012g"     |000000000000
123.456     | "%.1g"      |1e+02
-.001624    | "%.1g"      |-0.002
```

These formatting strings follow some standard which is normally not used in BASIC. The standard is well explained in Wikipedia: `http://en.wikipedia.org/wiki/Printf_format_string#Format_placeholders`.

Formatting Strings

Strings can also be formatted through `PRINT USING` but not with the function `USING$()`, although there are fewer options for strings than for numbers. Strings can be printed in the formatted field either left justified, centered, or right-justified. As with numbers, if the string is too long to fit, then asterisks are printed.

These examples should make it clear:

```
PRINT "|";"OK" USING "#####";"|"     ! result: | OK  |
PRINT "|";"OK" USING ">####";"|"     ! result: |   OK|
PRINT "|";"Hello" USING ">####";"|"  ! result: |Hello|
PRINT "|";"Goodby" USING ">####";"|" ! result: |*****|
```

If centering cannot be exact, the extra space is placed to the right. Actually any string can be used as a format string. Only the length of the string defines the length of the output field. Only the first character of the format string matters. If it is a < the string will be left justified, if it is a > it will be right-justified and centered in any other case. This is especially valuable for printing headers for a numeric table. The following example shows how you can format headers using the same format string we used earlier for numbers.

```
s$=                   |PRINT s$ USING "$#,###,###.##"
----------------------+------------------------------
"Cash"                |                   Cash
"Liabilities"         |              Liabilities
"Accounts Receivable" |             *************
```

3.10.4. Gathering User Input

You can make your program interactive and ask the user to enter data on runtime of your program.

The command `INPUT` allows the user to enter one line of data with the keyboard on the text console. The data is interpreted and stored in one or more variables specified by the INPUT statement. If you specify 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. Also numbers can be entered in scientific notation. Hexadecimal values are possible, too.

```
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 (floating point) variable x. You can then use this variable in your program as normal, doing calculations with it. Please note that 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 commas.

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

The user has then to enter commas at the appropriate places to tell which input goes to which variable. To the example above the user would respond with 5,6,7.

```
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.

3.11. 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.

3.11.1. 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.

3.12. Address 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. Address spaces belonging to other programs which are not shared memory blocks can not be accessed. You will get a segmentation fault on trying this.

3.13. 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 displayed after a SHOWPAGE command only. This allows fast animations.

To allow for animated bitmap graphics and icons, X11-Basic offers the commands GET and PUT, which retrieve rectangular regions from the graphics-window into a string or put back bitmap graphics data from the string to the graphics screen or window. The file format used with PUT is a standard .BMP bitmap, so also externally created icons can be used. Transparency and alpha channels are supported.

3.14. 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.

3.15. 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 standard internet protocols or pipes.

Basically there are two methods of connections to other computers on a network: The TCP/IP Based connections via streams and the implementation 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.

3.15.1. Local inter process communication: Pipes

A pipe is a unidirectional data channel that can be used for interprocess communication. The UNIX kernel usually supports this mechanism. The pipe can be used to send information or data from one process to another. Here is a little example progam you can use to play with it:

```
PIPE #1,#2
a=FORK()
IF a=0    ! Child instance
  GPRINT "Hi, I am Child !",b
  DO
    SHOWPAGE
    LINEINPUT #1,t$
    GPRINT t$
  LOOP
  ' This instance never ends ...
ELSE IF a=-1
  PRINT "ERROR, fork() failed !"
  QUIT
ELSE      ! parent instance
  DO
    DUMP
    ALERT 1,"Hi, I am Parent. Child PID="+str$(a),1," OK | Kill Child ! ",b
    DUMP
    PRINt #2,SYSTEM$("date")
    FLUSH #2
    IF b=2
      SYSTEM "kill "+str$(a)
      ALERT 1,"Child PID="+str$(a)+" killed !",1," OK ",b
      QUIT
    ENDIF
  LOOP
ENDIF
QUIT
```

Instad of with pipes, the interprocess communication can also be done using a shared memory segment.

3.15.2. World-Wide communication: Sockets

Most inter-process 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 communications 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.

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.

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 inter-process 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 and
3. Accept a connection with this other OPEN command, which opens a connection to the connected client:

```
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 (folow step 3) 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 a network.

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 immediately using SEND with a valid destination address and port as argument:

```
SEND #1,"This is my message",CVL(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.

3.16. Accessing USB devices

X11-Basic has a builtin USB interface, which allows X11-Basic programs to accedd USB-Devices, which are connected to the computer. The interface is on a near hardware level, so the driver for the specific hardware connected must be written in X11-Basic. Hence, it is well possible to use dataloggers and USB-to-RS232 adapters with this methods. In pronciple every USB-Device can be accessed, if the protocol for data transfers and data interpretation is known.

Please see the example program `usb.bas` for an example, how to readout data from a `VOLTCRAFT VDTL101-T` datalogger.

USB support is work in progress and may not yet work on Android and WINDOWS.

USB-Devices are opened with the OPEN command. Instead of a filename, a combination if PID/VID is used. Once opened, the commands CLOSE, IOCTL(), SEND and RECEIVE can be used on that device. (PRINT and INPUT currently will not work).

3.17. 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 how to store arbitrary binary data, which can be used e.g. to store the bitmapdata for a bitmap (). Or also for other resources like pictograms and any other bitmap or icon.

```
' 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
```

For convenience, a program called `inline.bas` shippes with X11-Basic. It does the conversion from and compression of any binary file to ready-to-use X11-Basic sourcecode.

3.18. 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.

3.18.1. 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 shared object files simultaneously, although 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.

3.19. 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.

3.19.1. 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 address of the memory occupied by a string can be obtained by the `VARPTR()` function. Its length by the `LEN()` function.

To allocate memory from the global and system-wide program 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%
```

3.20. 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 functions provided from within the X11-Basic program

4 GRAPHICAL USER INTERFACE

This chapter describes how to use the graphical user interface (GUI) built into X11-Basic.

4.1. ALERT and FILESELECT

Two most often used graphic functions are implemented as a full functional graphical user interface dialog: Message boxes and a file selector. Arbitrary dialogs can be created with the object and resource functions. Also a pull down menu function is implemented.

Fig. 4.1 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. 4.2. Here is a little example 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$,1,"OK|UPDATE|LÖSCHEN|CANCEL",a,f$
```

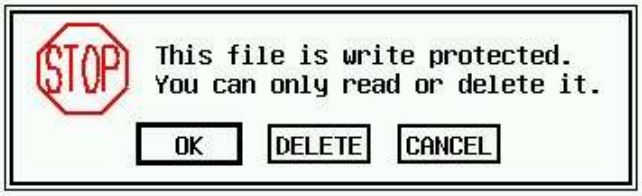

Figure 4.1: A message box.

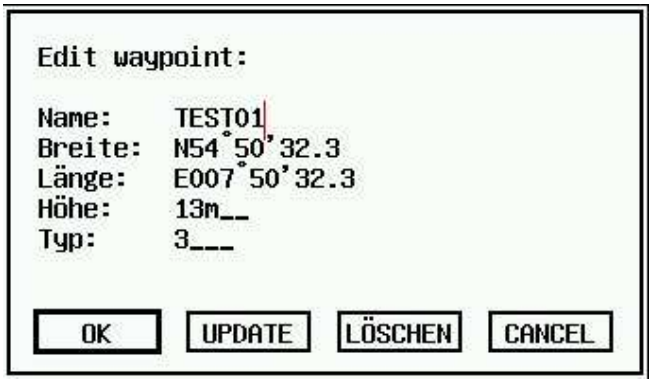

Figure 4.2: A simple input box.

```
WHILE LEN(f$)
  WORT_SEP f$,CHR$(13),0,a$,f$
  PRINT "Feld";i;": ",a$
  INC i
WEND
QUIT
```

Fig. 4.4 shows the fileselector box. The command which produces it is:

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

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

4.2. Resources

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 data format of X11Basic resource 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 pointers 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

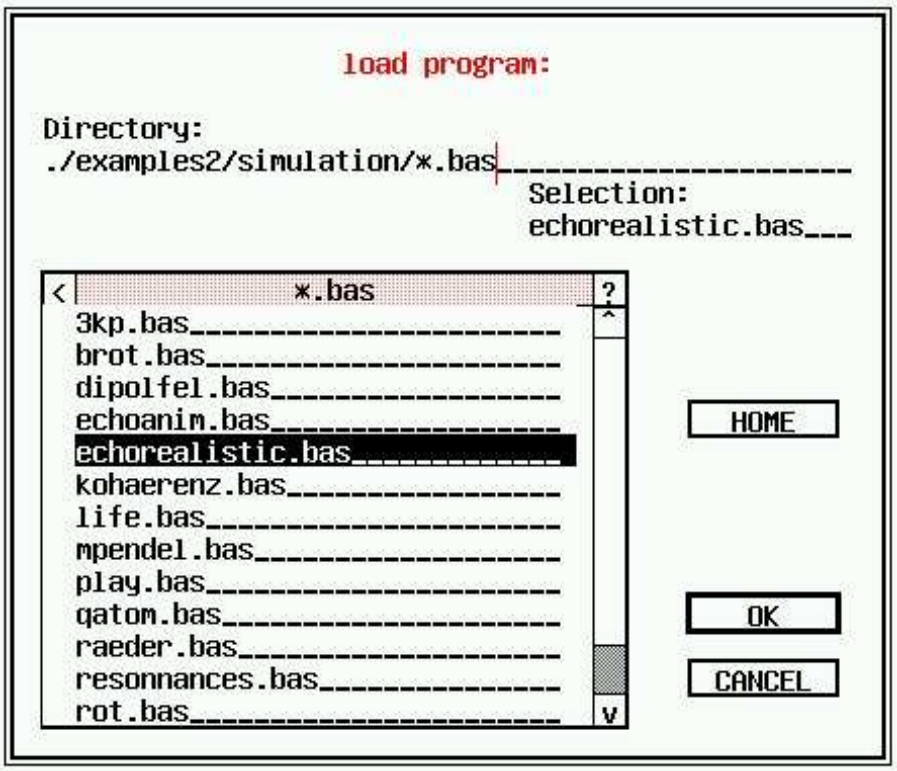

Figure 4.3: The fileselecetor

Figure 4.4: A pull down menu

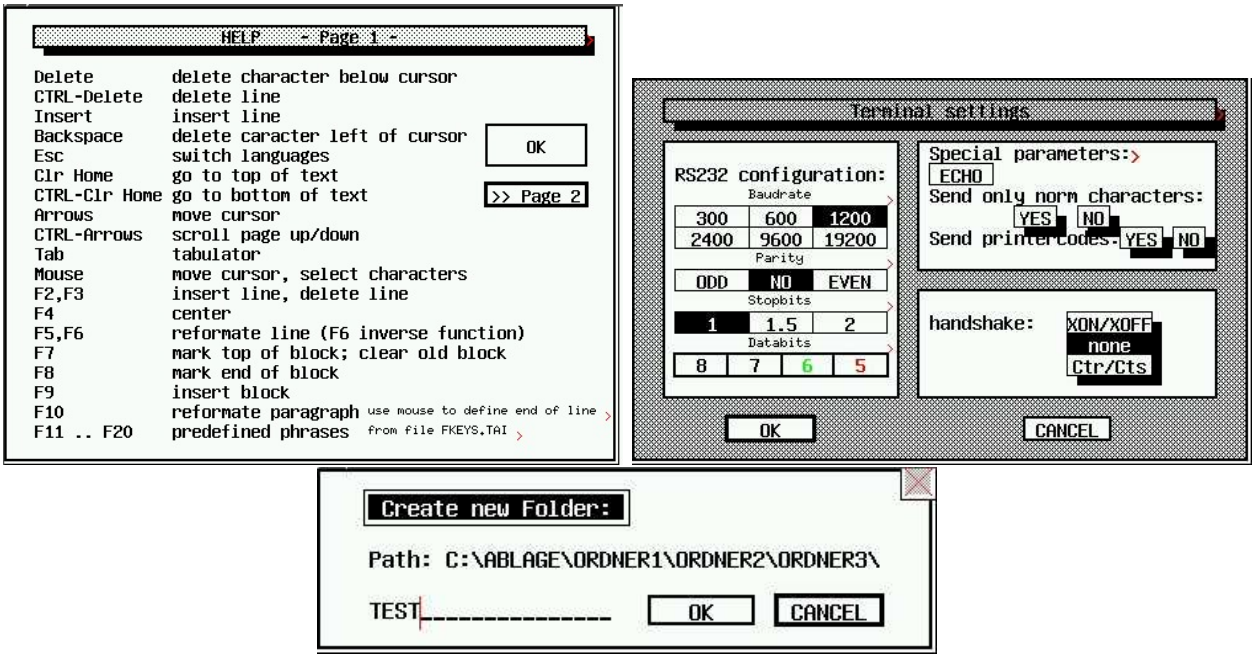

Figure 4.5.: Examples of forms in X11-Basic

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.

4.2.1. 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	Un-formatted 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 flag 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 check mark 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:

G_BOX	The low 16 bits contain a WORD containing color 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.
G_IBOX	The low 16 bits contain a WORD containing color 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.
G_BOXCHAR	The low 16 bits contain a WORD containing color information for the OBJECT. Bits 23-16 contain a signed 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:

Character	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, as
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.

4.2.2. The gui file format

The *.gui file format, which is basically an ASCII representation of the ATARI ST resource 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 resource files to *.gui Files see rsc2gui(1).

The *.gui file consists of Lines and Blocks which specify objects and their hierarchical 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 which opened the block. The block will be closed by a '}' in a single line.

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

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=SHAD
  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=1
      BUTTON(X=8,Y=1,W=4,H=1, TEXT="OFF",FLAGS=RADIOBUTTON+SELECTABLE,FRAME=2, FRAMECOL=1, TEXTCOL=1, BG
    }
  }
  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+)
}
```

4.3. 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 pop-up menus and drop-down list boxes (a special form of a pop-up 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 no|functio
    if b=2
      MENUSET k,8
    endif
  ENDIF
RETURN
```

5 WEB PROGRAMMING

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

5.1. 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.

5.1.1. 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 additional `chr$(13)`) in order for this to work. The flush statement ensures, that this statement is 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 along 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 easily 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.

5.2. 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.

5.2.1. 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. The standard (and platform independent) environment variables are shown in the following table:

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.

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 basic MIME type and subtype as follows:

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.

Type/Subtype	**Description**
`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/x-compress`	Data that has been compressed using UNIX compress
`application/x-gzip`	Data that has been compressed using UNIX gzip
`application/x-tar`	Data that has been archived using UNIX tar
`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
`multipart/mixed`	Multipart message with data in multiple formats
`text/html`	HTML-formatted text
`text/plain`	Plain text with no HTML formatting included
`video/mpeg`	Video in the MPEG format

Note, that there are more than the above listed types.
Some MIME content types can be used with additional parameters. These content types include text/plain, text/html, and all multi-part 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 multi-part 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 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.

5.2.2. 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.

5.2.3. Which CGI Input Method to use?

You can identify a submission method when you create your fill-out forms. 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.

5.2.4. 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.

5.2.5. 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.
```

5.2.6. 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
```

6 QUICK REFERENCE

6.1. Reserved variable names

There are some reserved variables. Some Keywords may not work as variable names as well. Although there is no checking done, parsing errors could occur. 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:

int	ANDROID?	-1 on Android systems, else 0	p.145
int	COLS	number of columns of the text terminal	p.201
int	CRSCOL	text cursor position: current column	p.211
int	CRSLIN	text cursor position: current line	p.211
flt	CTIMER	CPU system timer (seconds)	p.212
int	ERR	number of the last error	p.261
int	FALSE	constant: 0	p.280
int	GPS?	TRUE if GPS is available, else 0	p.322
flt	GPS_ALT	Altitude in m received from GPS	p.323
flt	GPS_LAT	latitude in degrees received from GPS	p.324
flt	GPS_LON	longitude in degrees received from GPS	p.324
int	MOUSEK	mouse button state	p.405
int	MOUSES	state of the shift, alt, ctrl, caps keys	p.405
int	MOUSEX	x coordinate of mouse position	p.405
int	MOUSEY	y coordinate of mouse position	p.405
int	PC	program counter	p.444
flt	PI	constant: 3.14159265359...	p.448
int	ROWS	number of rows of the text terminal	p.513
int	SENSOR?	TRUE if sensor phalanx is available	p.530
int	SP	internal stack pointer	p.556
int	STIMER	integer system timer	p.564
flt	TIMER	Unix system timer, float	p.584
int	TRUE	constant: -1	p.592
int	UNIX?	TRUE if OS is UNIX like (Linux, BSD)	p.599
int	WIN32?	TRUE if OS is MS WINDOWS 32 bit	p.623
	DATE$	current date	p.220

FILEEVENT$	get events about files	p.284
INKEY$	content of the keyboard-buffer	p.340
TERMINALNAME$	device name of the standard terminal	p.580
TIME$	current time	p.583
TRACE$	current program code line	p.588

6.2. Conditions

Conditions and expression are the same, FALSE is defined as 0 and TRUE as -1. As a consequence, Boolean operators like AND, OR, XOR etc. are applied as a bitwise operation. This way they can be used in expressions as well as in conditions.

6.3. Numbers and Constants

Number constants may precede 0x to represent hex values. String constants are marked with pairs of "". Array constants have following format: [, , ; , , ; , ,].

6.4. Operators

Precedence is defined as follows (highest first):

0. ()	(brackets)	
1. ^	(power)	
2. * /	(multiplication, division)	
3. \	(modulo)	
4. - +	()	
5. MOD DIV	(modulus, ...)	p.400,238
6. < > = <> <= >=	(comparison operators)	
7. AND OR XOR NOT EQV IMP	(logical operators)	p.143,437, 421,259,337

6.5. 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	p.493
?	PRINT	p.466
@	GOSUB	p.318
~	VOID	p.615
!	comment at the end of a line	
&	EVAL / indirect command	p.264

6.6. Interpreter Commands

`CLEAR`	clear and remove all variables	p.192
`CONT`	continue (after STOP)	p.206
`DUMP`	lists all used variable names	p.244
`DUMP "@"`	list of functions and procedures	p.244
`DUMP ":"`	list of all labels	p.244
`DUMP "#"`	list of open files	p.244
`DUMP "K"`	list of implemented commands	p.244
`DUMP "F"`	list of internal functions	p.244
`ECHO ON/OFF`	same as TRON * TROFF	p.246
`EDIT`	call default editor to edit program	p.247
`HELP <expr>`	prints short help on expr	p.328
`LIST [s,e]`	List program code (from line s to e)	p.370
`LOAD file$`	load program	p.372
`NEW`	clear all variables, erase program and stop	p.414
`PLIST`	formatted listing	p.452
`PROGRAM` options	set title and compiler options	p.473
`QUIT`	quits the X11-BASIC-Interpreter	p.480
`REM` comment	remark in program	p.493
`RUN`	start program	p.516
`STOP`	stop program	p.565

SAVE [file$]	writes the BASIC-program into file	p.518
TROFF	Trace mode off	p.590
TRON	Trace mode on (for debugging)	p.591
VERSION	shows X11-Basic version number and date	p.614
XLOAD	select and load a program	p.629
XRUN	select, load and run a program	p.632

6.7. Flow Control Commands

AFTER n,procedure	execute procedure after n seconds	p.139
BREAK	same as EXIT IF TRUE	p.170
CASE const	SELECT * CASE * DEFAULT * ENDSELECT	p.181
CHAIN bas$	executes another basic program	p.184
CONTINUE	SELECT * CASE * CONTINUE * ENDSELECT	p.206
DEFAULT	SELECT * CASE * DEFAULT * ENDSELECT	p.224
DEFFN	define function macro.	p.226
DO * LOOP	(endless) loop without condition	p.239
DOWNTO	FOR ... DOWNTO	p.240
ELSE	see IF * ELSE * ENDIF	p.249
ELSE IF	see IF * ELSE * ENDIF	p.249
END	program end, enter interactive mode	p.252
ENDFUNCTION	FUNCTION * ENDFUNCTION	p.253
ENDIF	IF * ELSE * ENDIF	p.254
ENDSELECT	SELECT * CASE * DEFAULT * ENDSELECT	p.256
EVERY n,procedure	invokes procedure every n seconds	p.269
EXIT IF a	exit loop if condition a is TRUE	p.275
FOR * NEXT	For Next loop	p.294
FUNCTION * ENDFUNC	define function	p.306
GOSUB proc(...)	call subroutine	p.318
GOTO label	goto label	p.319
IF * ELSE * ENDIF	conditional blocks	p.335
LOOP	DO * LOOP	p.380
NEXT	FOR * NEXT	p.415
ON BREAK GOSUB proc	define procedure on break	p.432
ON ERROR GOSUB proc	define procedure on error	p.433

`ON * GOSUB proc1,...`	excecute subroutine depending on value	p.430
`ON * GOTO label1,...`	branch to different labels depending on value	p.431
`REPEAT`	REPEAT * UNTIL	p.495
`RESUME`	resume program after error	p.498
`RETURN`	define the end of a PROCEDURE	p.499
`SELECT expr`	SELECT * CASE * DEFAULT * ENDSELECT	p.525
`UNTIL exp`	REPEAT * UNTIL	p.603
`SPAWN procedure`	Spawn new thread	p.558

6.8. Console Input/Output Commands

`BEEP`	Beep (on TTY/console)	p.162
`BELL`	same as BEEP	p.162
`CLS`	clear (text)screen	p.198
`FLUSH`	flush output	p.293
`HOME`	textcursor home	p.332
`INPUT "text";varlist`	read values for variables	p.345
`LINEINPUT t$`	read entire line from channel/file/console	p.367
`LOCATE row,column`	Place cursor on column and row	p.375
`PRINT a;b$`	console output	p.466
`PRINT AT(x,y);`	locate textcursor at row y and column x	p.467
`PRINT COLOR(x,y);`	change text color	p.468
`PRINT TAB(x);`	locate textcursor at column x	p.**??**
`PRINT SPC(x);`	move textcursor x columns	p.**??**
`PRINT a USING f$`	print number with formatter	p.470
`PUTBACK a`	put back a char to console	p.477

6.9. File Input/Output Commands

`BGET #f,a,n`	read n bytes from file #f to address a	p.163
`BLOAD f$,a[,l]`	reads entire file (given by name) to address a	p.165
`BPUT #f,a,n`	writes n bytes from address a to file/channel f	p.169

`BSAVE f$,a,l`	saves l bytes in memory at address a to file f$	p.171
`CHDIR path$`	change current working directory	p.185
`CHMOD file$,m`	change file permissions	p.186
`CLOSE [[#]n]`	close file, I/O channel or link	p.195
`FLUSH [#n]`	flush output	p.293
`KILL file$`	delete a file	p.358
`MAP`	maps a file into memory	p.**??**
`UNMAP`	unmaps memory	p.602
`MKDIR path$`	create a directory	p.398
`OPEN m$,#n,file$`	open a file or socket for input and/or output	p.434
`OUT #n,a`	out byte a to channel n	p.439
`PRINT #n;`	output to channel/file	p.466
`PUTBACK [#n,]a`	put back a char to channel/file/console	p.477
`RELSEEK #n,d`	Place filepointer on new relative position	p.492
`RENAME file$,dst$`	rename and move a file	p.494
`RMDIR path$`	remove an empty directory	p.506
`SEEK #n,d`	place filepointer to absolute position	p.524
`TOUCH #n`	update timestamps of file	p.587
`WATCH file$`	monitor file changes	p.618

6.10. Variable Manipulation Commands

`ABSOLUTE x,adr%`	Assigns the address to the variable x.	p.134
`ARRAYCOPY dst(),src()`	copies array including dimensioning	p.149
`ARRAYFILL a(),b`	fills array with value	p.149
`CLR a,b,c(),f$`	clear variables; same as `a=0;b=0;c()=[];f$=""`	p.197
`DEC var`	decrement variable; same as var=var-1	p.221
`DIM`	declare and create array	p.235
`ERASE a()[,...]`	erase arrays	p.260
`INC a`	increments variable a	p.338
`LET a=b`	enforces assignment	p.364
`LOCAL var[,...]`	declare local variables in a procedure or function	p.374
`SORT a(),n[,b()]`	Sort array	p.552
`SWAP a,b`	Swap variables	p.571
`VAR vars`	declare arguments to be passed "by reference"	p.611

6.11. Memory Manipulation Commands

`ABSOLUTE x,adr%`	Assigns the address to the variable x.	p.134
`BMOVE q,z,n`	copies a block of n bytes from address q to z	p.166
`DPOKE adr,word`	write short int word to adr	p.241
`FREE adr%`	Frees a previously allocated memory block.	p.301
`LPOKE adr,long`	writes long int value to pointer adr	p.383
`MFREE adr%`	Frees a previously allocated memory block.	p.395
`MSYNC adr%,l`	flushes changes map memory back to disk	p.408
`POKE adr,byte`	write byte to pointer adr	p.457
`SHM_DETACH adr%`	detaches the shared memory segment	p.540
`SHM_FREE adr%`	frees the shared memory segment	p.541

6.12. Math commands

`ADD a,b`	same as a=a+b but faster	p.137
`DEC var`	same as var=var-1 but faster	p.221
`DIV a,b`	same as a=a/b but faster	p.238
`FFT a(),i`	fast fourier transformation on 1D array.	p.283
`FIT x(),y()[,yerr()],n,func(x,a,b,c,...)`	fits function to data	p.288
`FIT_LINEAR x(),y()[,[xerr(),]yerr()],n,a,b[,siga,sigb,chi2,q]`	linear regression with errors	p.289
`FIT_POLY x(),y(),dy(),n%,a(),m%`	fit a polynom to datapoints	p.290
`INC var`	same as var=var+1 but faster	p.338
`MUL a,b`	same as a=a*b but faster	p.411
`SORT a(),n[,b()]`	sorts n values of a() to incrementing order	p.552
`SUB a,b`	same as a=a-b but faster	p.568

6.13. Other Commands

`CALL adr[,par,...]`	see EXEC	p.178
`CONNECT #n,t$[,i%]`	connect a channel	p.205
`DATA 1,"Hallo",...`	define constants	p.219
`DELAY sec`	same as PAUSE	p.232
`ERROR n`	execute error number n	p.263
`EVAL t$`	execute X11-Basic command contained in t$	p.264
`EXEC adr[,var,...]`	call a C subroutine at pointer adr.	p.271
`GET_LOCATION ,,,,,,,`	returns the position of the device	p.315
`GPS ON/OFF`	turns GPS device on/off	p.321
`LINK #n,t$`	load shared object file t$	p.369
`UNLINK #n`	unload shared object file	p.601
`MERGE f$`	Merges bas-file to actual program code	p.394
`NOP`	no operation do nothing	p.418
`NOOP`	no operation do nothing	p.418
`PAUSE sec`	pauses sec seconds	p.442
`PIPE #l,#k`	links two file channels	p.449
`PLAYSOUND c,s$`	plays a WAV sample	p.450
`PLAYSOUNDFILE file$`	plays a sound file	p.451
`PROCEDURE proc(p1,...)`	PROCEDURE * RETURN	p.471
`RANDOMIZE [seed]`	Sets seed for random generator	p.486
`READ var`	reads constant from DATA statement	p.488
`RECEIVE #n,t$`	receive a message from a socket	p.491
`RESTORE [label]`	(re)sets pointer for READ-statement to label	p.497
`RETURN expr`	return value from FUNCTION	p.499
`RSRC_LOAD file$`	loads GEM rsc-File (ATARI ST)	p.515
`RSRC_FREE`	frees GEM rsc-File	p.514
`SEND #n,t$`	send a message to a socket	p.527
`SENSOR ON/OFF`	turns SENSORs on/off	p.531
`SETENV t$=a$`	Sets environmentvar t$ using value a$	p.532
`SOUND freq`	Sound the internal speaker	p.554
`SPLIT t$,d$,mode,a$,b$`	splits t$ by deliminator d$ into a$ and b$	p.560
`SHELL t$`	execute file as shell	p.537
`SPEAK t$`	Text to speach	p.559
`SYSTEM t$`	execute shell with command t$	p.574

UNLINK #n	un-links shared object #n	p.601
VOID a	calculates expression a and discard result	p.615
WAVE c,f,	control the sound synthesizer	p.619
WORT_SEP	same as SPLIT	p.627

6.14. Graphic commands

6.14.1. Drawing and painting

BOUNDARY f	switch borders on or off	p.167
BOX x1,y1,x2,y2	draw a frame	p.168
CIRCLE x,y,r,,	draw a circle	p.191
CLIP ,,,,,	clipping function	p.194
COLOR f[,b]	Set foreground color (and background color)	p.199
COPYAREA ,,,,,	copy a rectangular screen section	p.207
CURVE ,,,,,,,	cubic Bezier-curve	p.213
DEFFILL c,a,b	set fill style and pattern	p.225
DEFLINE a,b	set line width and type	p.227
DEFMARK c,a,g	set color, size, type (POLYMARK)	p.228
DEFMOUSE i	set mouse cursor type	p.229
DEFTEXT c,s,r,g	set text properties for ltext	p.230
DRAW [[x1,y1] TO] x2,y2	draw line	p.243
ELLIPSE x,y,a,b[,a1,a2]	draw an ellipse	p.248
FILL x,y	flood fill	p.287
GET x,y,w,h,g$	store a portion of the screen bitmap in g$	p.312
GPRINT	like PRINT, but the output goes to the graphic window	p.320
GRAPHMODE mode	set graphic-mode	p.325
LINE x1,y1,x2,y2	draw a line	p.366
LTEXT x,y,t$	Linegraphic-Text	p.384
PBOX x1,y1,x2,y2	draw filled box	p.443
PCIRCLE x,y,r[,a1,a2]	draw filled circle	p.445
PELLIPSE x,y,a,b[,a1,a2]	draw filled ellipse	p.447
PLOT x,y	draw point	p.453
POLYLINE n,x(),y()	draw polygon in (x(),y())	p.459
POLYFILL n,x(),y()	draw filled polygon	p.458

POLYMARK n,x(),y()	draw polygon points	p.460
PRBOX x1,y1,x2,y2	draw filled rounded box	p.462
PUT x,y,g$	map graphic at position	p.475
PUT_BITMAP t$,i,i,i,i	map bitmap	p.478
RBOX x1,y1,x2,y2	draws a rounded box	p.487
SCOPE a(),typ,ys,yo	fast plot of data a()	p.521
SCOPE y(),x(),typ,ys,yo,xs,xo	fast 2D plot of data	p.521
SETFONT f$	set bitmap font	p.533
SETMOUSE x,y	set mouse cursor	p.534
SGET screen$	capture graphic and store it in screen$	p.535
SPUT screen$	maps graphic to window/screen	p.561
TEXT x,y,t$	draw text (bitmap font)	p.582

6.14.2. Screen/Window commands

CLEARW [#n]	clear graphic window	p.193
CLOSEW [#n]	close graphic window	p.196
FULLW n	make window fullscreen	p.305
GET_GEOMETRY ,,,,	returns the size of the window or screen	p.314
GET_SCREENSIZE ,,,,	returns the size of the screen	p.316
INFOW n,t$	set window information	p.339
MOVEW n,x,y	move window	p.406
OPENW n	open window	p.436
ROOTWINDOW	draw on screen background	p.509
NOROOTWINDOW	switch back to normal output	p.420
SAVESCREEN file$	save screen bitmap into a file	p.519
SAVEWINDOW file$	save window bitmap into a file	p.520
SCREEN n	select Screen mode	p.523
SHOWPAGE	perform pending graphic operations	p.544
SIZEW n,w,h	size window	p.549
TITLEW n,t$	set window title	p.585
TOPW n	move window to front	p.586
USEWINDOW #n	direct graphics output to window n	p.605
VSYNC	same as SHOWPAGE	p.616

6.14.3. GUI/User input commands

`ALERT a,b$,c,d$,var[,new%]`	Show Alert/Infobox and wait for user input	p.141
`EVENT ,,,,,,,,`	Waits until an event occurs	p.267
`FILESELECT tit$,path$,file$,a%`	display a fileselector-box	p.286
`HIDEM`	hide the mouse cursor	p.331
`KEYEVENT a,b`	Waits until key is pressed	p.357
`LISTSELECT tit$,list$(),i`	display a selector-box	p.371
`MENUDEF m$(),proc`	read menu titles and items from m$()	p.391
`MENUKILL`	deletes menu	p.392
`MENUSET n,x`	change menu-point #n with value x	p.393
`MENU STOP`	switch off the menu	p.**??**
`ONMENU`	execute the menu and	p.**??**
`MENU`	wait for menu-events	p.390
`MOUSE x,y,k`	gets position and state of mouse	p.403
`MOUSEEVENT ,,,`	wait for mouse event	p.404
`MOTIONEVENT ,,,`	wait for mouse movement	p.402
`OBJC_ADD t%,o%,c%`	add object to tree	p.423
`OBJC_DELETE t%,o%`	delete object from tree	p.424
`RSRC_LOAD file$`	loads a GEM resource file	p.515
`RSRC_FREE`	unloads a GEM resource	p.514
`SHOWM`	show the mouse cursor	p.543

6.15. File Input/Output functions

`d%=DEVICE(file$)`	returns the device id of a file	p.234
`b=EOF(#n)`	TRUE if file pointer reached end of file	p.258
`b=EXIST(fname$)`	TRUE if file fname$ exist	p.273
`a=FREEFILE()`	Returns first free filenumber or -1	p.302
`a$=FSFIRST$(path$,,)`	searches for the first file in a filesystem	p.303
`a$=FSNEXT$()`	searches for the next file	p.304
`c=INP(#n)`	reads a byte from channel/file.	p.343
`c=INP?(#n)`	number of bytes which can be read	p.344
`a=INP&(#n)`	reads a word (2 Bytes) from channel/file.	p.343

`i=INP%(#n)`	reads a long (4 Bytes) from channel/file.	p.343
`t$=INPUT$(#n,num)`	reads num bytes from file/channel n	p.346
`ret=IOCTL(#n,d%,)`	performs settings on channel/file.	p.351
`t$=LINEINPUT$(#n)`	reads a line from file/channel n	p.367
`p=LOC(#n)`	Returns value of file position indicator	p.373
`l=LOF(#n)`	length of file	p.376
`l%=SIZE(file$)`	returns the size of a file	p.548
`t$=TERMINALNAME$(#n)`	returns device name of terminal connected to #n	p.580

6.16. Variable/String Manipulation functions

`adr%=ARRPTR(b())`	pointer to array descriptors	p.151
`a=ASC(t$)`	ASCII code of first letter of string	p.152
`b$=BIN$(a[,n])`	convert to binary number	p.164
`t$=CHR$(a)`	convert ASCII code to string	p.188
`a$=DECLOSE$(t$)`	removes enclosing characters from string	p.222
`a=DIM?(a())`	returns number of elements of array a()	p.235
`a$=ENCLOSE$(t$[,p$])`	encloses a string	p.250
`f=GLOB(a$,b$[,flags])`	TRUE if a$ matches pattern b$	p.317
`t$=HEX$(a[,n])`	a as hexadecimal number	p.330
`t$=INLINE$(a$)`	6-bit ASCII to 8-bit binary conversion	p.341
`a=INSTR(s1$,s2$[,n])`	tests if s2$ is contained in s1$	p.347
`a=TALLY(t$,s$)`	returns the number of occurrences of s$ in t$	p.577
`b%=INT(a)`	convert to integer	p.348
`t$=LEFT$(a$[,n])`	extracts n bytes from string a$ from the left	p.361
`t$=LEFTOF$(a$,b$)`	returns left part of a$ split at b$	p.362
`l=LEN(t$)`	length of string	p.363
`u$=LOWER$(t$)`	converts t$ to lower case	p.381
`l=LTEXTLEN(t$)`	size of text	p.385
`m$=MID$(t$,s[,l])`	extracts l bytes from string t$ from position s	p.396
`t$=MKA$(a())`	convert a whole array into a string	p.399
`t$=MKI$(i)`	convert integer to 2-byte string	p.399
`t$=MKL$(i)`	convert integer to 4-byte string	p.399
`t$=MKF$(a)`	convert float to 4 byte string	p.399
`t$=MKD$(a)`	convert float to 8 byte string	p.399

o$=OCT$(d,n)	convert integer d to string with octal number	p.428
t$=REPLACE$(a$,s$,r$)	replace s$ by r$ in a$	p.496
t$=REVERSE$(a$)	Return the reverses of a string	p.500
t$=RIGHT$(a$[,n])	returns right n characters of a$	p.501
t$=RIGHTOF$(a$,b$)	returns right part of a$ split at b$	p.502
a=RINSTR(s1$,s2$[,n])	tests from right if s2$ is contained in s1$	p.503
t$=SPACE$(i)	returns string consisting of i spaces	p.557
t$=STR$(a[,b,c])	convert number to string	p.566
t$=STRING$(i,w$)	returns string consisting of i copies of w$	p.567
u$=TRIM$(t$)	trim t$	p.589
u$=XTRIM$(t$)	trim t$	p.633
u$=UCASE$(t$)	converts t$ to upper case	p.597
u$=UPPER$(t$)	converts t$ to upper case	p.604
u$=USING$(a,f$)	formats a number	p.606
a=VAL(t$)	converts String/ASCII to number	p.609
i=VAL?(t$)	returns number of chars which are part of a number	p.609
adr%=VARPTR(v)	returns pointer to variable	p.613
u$=WORD$(b$,n)	returns n th word of b$	p.625
e=WORT_SEP(t$,d$,m,a$(),t$)	split t$ into parts	p.**??**

6.17. Data compression and coding functions

b$=ARID$(a$)	order-0 adaptive arithmetic decoding	p.147
b$=ARIE$(a$)	order-0 adaptive arithmetic encoding	p.148
b$=BWTD$(a$)	inverse Burrows-Wheeler transform	p.174
b$=BWTE$(a$)	Burrows-Wheeler transform	p.175
c$=COMPRESS$(a$)	lossless compression on the string	p.203
c$=UNCOMPRESS$(a$)	lossless uncompression on the string	p.598
c%=CRC(t$[,oc])	32 bit checksum	p.210
e$=ENCRYPT$(t$,key$)	encrypts a message	p.251
t$=DECRYPT$(e$,key$)	decrypts a message	p.223
b$=MTFD$(a$)	Move To Front decoding	p.409
b$=MTFE$(a$)	Move To Front encoding	p.410
b()=CVA(a$)	returns array reconstructed from the string	p.214
b%=CVI(a$)	convert 2-byte string to integer	p.216

b%=CVL(a$)	convert 4-byte string to integer	p.216
b=CVS(a$)	convert 4-byte string to float	p.217
b=CVF(a$)	convert 4-byte string to float	p.215
b=CVD(a$)	convert 8-byte string to double	p.215
t$=INLINE$(a$)	6-bit ASCII to 8-bit binary conversion	p.341
t$=REVERSE$(a$)	return the reverses of a string	p.500
b$=RLD$(a$)	run length decoding	p.504
b$=RLE$(a$)	run length encoding	p.505

6.18. Memory Manipulation functions

adr%=ARRPTR(b())	pointer to array descriptors	p.151
i%=DPEEK(adr%)	read word from pointer adr	p.241
b%=LPEEK(adr%)	reads long (4 Bytes) from address	p.382
adr%=MALLOC(n%)	allocates size bytes of memory	p.388
adr%=MSHRINK(adr%,n%)	reduces the size of a storage area	p.407
d%=PEEK(a%)	reads Byte from address a	p.446
adr%=REALLOC(oadr%,n%)	changes the size of a storage area	p.490
adr%=SHM_ATTACH(id)	attaches the shared memory segment	p.539
id=SHM_MALLOC(size,key)	returns the identifier of the shared memory segment	p.542
adr%=SYM_ADR(#n,s$)	return pointer to symbol from shared object file	p.573
adr%=VARPTR(v)	returns pointer to variable	p.613

6.19. Logic functions

c%=AND(a%,b%)	same as c=(a AND b)	p.143
c%=OR(a%,b%)	same as c=(a OR b)	p.437
c%=XOR(a%,b%)	same as c=(a XOR b)	p.630
c%=EQV(a%,b%)	same as c=(a EQV b)	p.259
c%=IMP(a%,b%)	same as c=(a IMP b)	p.337
b%=BCHG(x%,bit%)	changes the bit of x from 0 to 1 or from 1 to 0	p.160
b%=BCLR(x%,bit%)	sets the bit of x to zero.	p.160

`b%=BSET(x%,bit%)`	sets the bit of x to 1.	p.172
`b%=BTST(x%,bit%)`	returns -1 if the bit of x is 1.	p.173
`b%=BYTE(x%)`	same as b=x AND 255	p.176
`b%=CARD(x%)`	same as b=x AND 0xffff	p.180
`b%=WORD(x%)`	same as b=x AND 0xffff	p.624
`b%=EVEN(d)`	TRUE if d is an even number	p.266
`b%=ODD(d)`	TRUE if d is an odd number	p.429
`b%=GRAY(a)`	Gray code. if a<0: inverse Gray code	p.326
`b%=SHL(a)`	Shift bits to left	p.538
`b%=SHR(a)`	Shift bits to right	p.545
`b%=SWAP(a)`	Swaps High and Low words of a	p.571

6.20. 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)`	absolute value $b = \|a\|$	p.133
`c=ADD(a,b)`	add $c = a + b$	p.137
`b=CBRT(a)`	cube root $b = \sqrt[3]{a}$	p.182
`a=CEIL(b)`	truncate number	p.183
`a=CINT(b)`	truncate number (note: differs from INT() !)	p.190
`z=COMBIN(n,k)`	number of combinations $z = \frac{n!}{(n-k)!\cdot k!}$	p.202
`c=DIV(a,b)`	divide $c = a/b$	p.238
`b()=FFT(a()[,f%])`	discrete Fourier Transformation of a real array	p.283
`a=FIX(b)`	round number to integer	p.291

`a=FLOOR(b)`	round number down to integer	p.292
`b=FRAC(a)`	fractional (non-integer) part of a	p.300
`y=GAMMA(x)`	gamma function $y = \Gamma(x)$	p.309
`y=LGAMMA(x)`	logarithm of gamma function $y = \lvert \ln\Gamma(x) \rvert$	p.365
`a=HYPOT(x,y)`	returns $a = \sqrt{x^2 + y^2}$	p.333
`b=INT(a)`	convert to integer	p.348
`b()=INV(a())`	calculate inverse of a square matrix	p.349
`i=SGN(a)`	sign of a (-1,0,1)	p.536
`b=SQR(a)`	square root $b = \sqrt{a}$	p.562
`b=SQRT(a)`	square root $b = \sqrt{a}$	p.562
`b=TRUNC(a)`	truncate number	p.593
`b=LN(a)`	base e logarithm (natural log)	p.377
`b=LOG(a)`	base e logarithm (natural log)	p.377
`b=LOG10(a)`	base 10 logarithm	p.377
`b=LOGB(x)`	base 2 logarithm	p.378
`b=LOG1P(x)`	$b = \log(1 + x)$ accurate near zero	p.379
`c=MOD(a,b)`	same as c=(a MOD b)	p.400
`c=MUL(a,b)`	multiply $c = a \cdot b$	p.411
`b=EXP(a)`	exponential function $b = e^x$ (e to the x)	p.276
`b=EXPM1(a)`	exponential function minus 1 $b = e^x - 1$	p.277
`b=FACT(a)`	factorial $b = a!$	p.279
`a=PRED(x)`	returns the preceding integer of x	p.463
`a=SUCC(x)`	returns the next higher integer	p.570
`b()=SOLVE(a(),x())`	solve linear equation system	p.550
`z=VARIAT(n,k)`	number of permutations of n elements	p.612

6.20.1. Angles

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

`b=RAD(a)`	convert degrees to radians	p.482
`b=DEG(a)`	convert radians to degrees	p.231

6.20.2. Trigonometric functions

`b=SIN(a)`	sine	p.546
`b=COS(a)`	cosine	p.208
`b=TAN(a)`	tangent	p.578
`b=ASIN(a)`	arc-sine	p.153
`b=ACOS(a)`	arc-cosine	p.135
`b=ATAN(a)`	arc-tangent	p.156
`b=ATN(a)`	arc-tangent	p.156
`b=ATAN2(a,c)`	extended arc-tangent	p.156
`b=SINH(a)`	hyperbolic sine	p.547
`b=COSH(a)`	hyperbolic cosine	p.208
`b=TANH(a)`	hyperbolic tangent	p.579
`b=ASINH(a)`	hyperbolic arc-sine	p.153
`b=ACOSH(a)`	hyperbolic arc-cosine	p.135
`b=ATANH(a)`	hyperbolic arc-tangent	p.157

6.20.3. Random numbers

`a=GASDEV(dummy)`	random number Gauss distribution	p.310
`a=RAND(dummy)`	random integer number	p.484
`a=RANDOM(n)`	random integer number between 0 and n	p.485
`a=RND(dummy)`	random number between 0 and 1	p.507
`a=SRAND(seed)`	same as RANDOMIZE	p.563

6.21. System functions

`ret%=CALL(adr%[,par]`	Calls a machine code or C subroutine	p.178
`t$=ENV$(n$)`	read value of environment variable n$	p.257
`t$=ERR$(i)`	error message	p.262
`ret=EXEC(adr[,var])`	see command EXEC, returns int	p.271
`i%=FORK()`	creates a child process	p.295

d$=JULDATE$(a)	date$ by Julian day a	p.354
a=JULIAN(date$)	Julian day	p.355
a$=PARAM$(i)	i'th word from the commandline	p.441
t$=PRG$(i)	program line	p.465
a=SENSOR(i)	get the value from the i th sensor	p.531
t$=SYSTEM$(n$)	execute shell with command n$	p.574
t$=UNIXTIME$(i)	give time$ from TIMER value	p.600
d$=UNIXDATE$(i)	give date$ from TIMER value	p.600

6.22. Graphic functions

c=COLOR_RGB(r,g,b[,a])	allocate color by rgb(a) value	p.200
a=EVENT?(mask%)	returns TRUE if a graphics event is pending	p.268
a=FORM_ALERT(n,t$)	message box with default button n	p.296
~FORM_CENTER(adr%,x,y,w,h)	centers the object tree on screen	p.297
a=FORM_DIAL(,,,,,,,,)	complex function for screen preparation	p.298
a=FORM_DO(i)	do dialog	p.299
c=GET_COLOR(r,g,b)	allocate color by rgb value	p.313
d=OBJC_DRAW(,,,,)	draw object tree	p.425
ob=OBJC_FIND(tree,x,y)	return object number by coordinates	p.426
a=OBJC_OFFSET(t%,o,x,y)	calculate absolute object coordinates	p.427
c=POINT(x,y)	returns color of pixel of graphic in window	p.456
c=PTST(x,y)	returns color of pixel of graphic in window	p.474
a=RSRC_GADDR(typ,nr)	get pointer to object tree	p.**??**

6.23. Other functions

a=EVAL(t$)	evaluate expression contained in t$	p.264
m=MAX(a,b,c,...)	returns biggest value	p.389
m=MAX(f())	not implemented yet	
m=MIN(a,b,c,...)	returns smallest value	p.397
m=MIN(array())	not implemented yet	

`m=MIN(function())` not implemented yet

6.24. Subroutines and Functions

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
```

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
```

6.25. Error Messages

X11-Basic can produce a number of internal errors, which are referred 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 .. 2147483647

3 Value not byte 0 .. 255
4 Value not short -32768 .. 32767
5 Square root: only positive numbers
6 Logarithm only for positive numbers
7 Unknown Error
8 Out of Memory
9 Function or command || is not implemented in this version
10 String too long
11 Argument needs to be positive
12 Program too long, buffer size exceeded –> NEW
13 Type mismatch in expression
14 Array () is already dimensioned
15 Array not dimensioned: ()
16 Field index too large
17 Dim too large
18 Wrong number of indexes
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
29 Illegal address!
30 Merge - no ASCII file
31 Merge - line too long - CANCEL
32 ==> Syntax error
33 Label not defined
34 Not enough data
35 data must be numeric
36 Error in program structure
37 Disk full
38 Command not allowed in interactive mode
39 Program Error GOSUB impossible
40 CLEAR not allowed within For-Next-loops or procedures

41 CONT not possible
42 Not enough parameters
43 Expression too complex
44 Function not defined
45 Too many parameters
46 Incorrect parameter, must be number
47 Incorrect parameter, must be string
48 Open "R" - incorrect Field length
49 Too many "R"-files (max. 31)
50 No "R"-file
51 Parser: Syntax Error <>
52 Fields larger than field length
53 Wrong graphic format
54 GET/PUT wrong Field-String length
55 GET/PUT wrong number
56 Wrong number of parameters
57 Variable is not yet initialized
58 Variable has incorrect type
59 Graphic has wrong color depth
60 Sprite-String length wrong
61 Error with RESERVE
62 Menu wrong
63 Reserve wrong
64 Pointer wrong
65 Field size < 256
66 No VAR-Array
67 ASIN/ACOS wrong
68 Wrong VAR-Type
69 ENDFUNC without RETURN
70 Unknown Error 70
71 Index too large
72 Error in RSRC_LOAD
73 Error in RSRC_FREE
74 Array dimensioning mismatch
75 Stack overflow!
76 Illegal variable name . can not create.
77 Function not defined for complex numbers.
80 Matrix operations only allowed for one or two dimensional arrays

81 Matrices do not have the same order
82 Vector product not defined
83 Matrix product not defined
84 Scalar product not defined
85 Transposition only for two dimensional matrices
86 Matrix must be square
87 Transposition not defined
88 FACT/COMBIN/VARIAT/ROOT not defined
89 Array must be two dimensional
90 Error in Local
91 Error in For
92 Resume (next) not possible: Fatal, For or Local
93 Stack Error
94 Parameter must be float ARRAY
95 Parameter must be ARRAY
96 ARRAY has the wrong type. Can not convert.
97 This operation is not allowed for root window
98 Illegal Window number (0-16)
99 Window does not exist
100 X11-BASIC Version 1.24 Copyright (c) 1997-2016 Markus Hoffmann
101 ** 1 - Segmentation fault
102 ** 2 - Bus Error: peek/poke ?
103 ** 3 - Address error: Dpoke/Dpeek, Lpoke/Lpeek?
104 ** 4 - Illegal Instruction
105 ** 5 - Divide by Zero
106 ** 6 - CHK exception
107 ** 7 - TRAPV exception
108 ** 8 - Privilege Violation
109 ** 9 - Trace exception
110 ** 10 - Broken pipe
131 * Number of hash collisions exceeds maximum generation counter value.
132 * Wrong medium type
133 * No medium found
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
146     * Connection timed out
147     * Too many references: can not splice
148     * Can not send after transport endpoint shutdown
149     * Transport endpoint is not connected
150     * Transport endpoint is already connected
151     * No buffer space available
152     * Connection reset by peer
153     * Software caused connection abort
154     * Network dropped connection because of reset
155     * Network is unreachable
156     * Network is down
157     * Can not assign requested address
158     * Address already in use
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
169     * Too many users
170     * Streams pipe error
171     * Interrupted system call should be restarted
172     * Illegal byte sequence
173     * Can not 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
185 * Too many symbolic links encountered
186 * File name too long
187 * Resource deadlock would occur
188 * Advertise error
189 * memory page error
190 * no executable
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 in this channel not possible (any more)
207 * no more files
208 * Link number out of range
209 * Level 3 reset
210 * Illegal Drive identifier
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 * illegal page address
217 * Directory not empty
218 * Function not implemented
219 * Illegal Handle
220 * Access not possible
221 * Too many open files
222 * Path not found
223 * File not found
224 * Broken pipe
225 * Too many links
226 * Read-Only File-System
227 * Illegal seek
228 * No space left on device
229 * File too large
230 * Text file busy
231 * Not a typewriter
232 * Too many open files
233 * File table overflow
234 * Invalid argument
235 * Is a directory
236 * Not a directory
237 * No such device
238 * Cross-device link
239 * File exists
240 * Bad sector (verify)
241 * unknown device
242 * Disk was changed
243 * Permission denied
244 * Not enough core memory
245 * read error
246 * write error
247 * No paper
248 * Sector not found
249 * Arg list too long
250 * Seek Error
251 * Bad Request
252 * CRC Error wrong check sum

253 * No such process
254 * Timeout
255 * IO-Error

7 COMMAND REFERENCE

This chapter is a command reference for quick lookup of short explanations of all bult-in X11-Basic operators, variables, commands, and functions.

7.1. 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
```

7.2. A

Function: ABS()

Syntax:
```
a=ABS(b)
a%=ABS(b%)
a=ABS(b#)
a&=ABS(b&)
```

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. The absolute value of a complex number is a real positive number.

EXAMPLE:

```
PRINT ABS(-34.5),ABS(34)   !    Result: 34.5     34
PRINT ABS(4+3i)            !    Result: 5
```

SEE ALSO: SGN()

Command: ABSOLUTE

Syntax: `ABSOLUTE var,adr%`

DESCRIPTION:

ABSOLUTE assigns the address adr% to the variable var.

EXAMPLE:

```
a=3
b=4
ABSOLUTE a,VARPTR(b)
PRINT a    ! Result: 4  Variables a and b are now identical.
```

SEE ALSO: VAR

Function: ACOS()

Syntax: `a=ACOS(b)`

DESCRIPTION:

The ACOS() is the arc 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: `a=ACOSH(b)`

DESCRIPTION:

The ACOSH() is the inverse hyperbolic cosine function, i.e. the inverse of the COSH() function. It returns the angle (in radian), which, fed to the hyperbolic cosine function will produce the argument passed to the ACOSH() function.

EXAMPLE:

```
PRINT ACOSH(2),ACOSH(COSH(0))  ! Result: 1.316957896925 0
```

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

Command: ADD

Syntax:

```
ADD a,<num-expression>
ADD a%,<num-expression>
ADD a#,<num-expression>
ADD a&,<num-expression>
```

DESCRIPTION:

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

EXAMPLE:

```
a=0.5
ADD a,5
PRINT a     !   Result: 5.5
```

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

*

Function: ADD()

Syntax:
```
c=ADD(a,b)
c&=ADD(a&,b&)
c#=ADD(a#,b#)
c%=ADD(a%,b%)
```

DESCRIPTION:

The function ADD() returns the sum of its arguments.

EXAMPLE:

```
a=0.5
b=ADD(a,5)
PRINT b       !  Result: 5.5
```

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

Command: AFTER

Syntax: `AFTER <num-variable>, <procedure-name>`

DESCRIPTION:

Procedures can be called after the expiry of a set time. Time in seconds.

COMMENT: The currecnt implementation uses the alarm mechanism of the kernel of the operating system. This means, only one procedure can be scheduled for trigger at the same time. Once scheduled, AFTER cannot be canceled anymore. But you can overwrite it with following AFTER commands. If you use another AFTER command before the previous one has triggered the procedure the previous will not be triggered anymore.

The procedure is excecuted exactly at the given time, interrupting the currently running process, even in the middle of a command. This can lead to a mess in program stack, unpredictible crashes may be caused. Using AFTER (and EVERY) is not safe! The interrupt procedure should not do complicated things. Maybe just assign a constant to a variable.

Maybe also good to know: A PAUSE command will be immediately ended, when the AFTER procedure is triggered. The procedure will still be triggered, even when the main program has already ended.

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$,defaultbutton%,button$,click%[,text$]`

DESCRIPTION:

Creates and displays an alert box (with a message) and asks for user input. The message box can have one ore more buttons which can be clicked by the user to exit the message box. Also the user can enter text in several text input fields if they have been specified. The number of the button clocked is returned in click% and the entered text in text$.

type% chooses type of alert symbol, 0=none, 1="!", 2="?", 3="stop" message$ Contains main text. Lines are separated by the ’|’ symbol. Editable fields are started with a CHR$(27) followed by the default text to be edited (until "|"). button$ Contains text for the buttons (separated by ’|’). defaultbutton% is the button to be highlighted (0=none,1,2,...) to be selected by just pressing return. click% This variable is set to the number of the button selected. text$ 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).

COMMENT: The length of the text input fields is given by the length of the default text. If you want the user to be able to enter longer texts than the default, the default can be extended by zero bytes (CHR$(0)) which are invisible to the user.

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$="TEST01"+STRING$(4,CHR$(0))   ! maximum length 6+4=10
posx$="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>

EXAMPLE:

```
PRINT AND(TIMER,0xff)  ! Result: 67
```

SEE ALSO: `OR(), AND`

Variable: ANDROID?

Syntax: ANDROID?

DESCRIPTION:

This variable gives -1 (=TRUE) if the operating system is Android; else the variable has a value of 0. With testing this variable the program can find out if it is running on an Android device.

EXAMPLE:

```
IF NOT ANDROID?
  MOUSEEVENT
  QUIT
ELSE
  END
ENDIF
```

SEE ALSO: TRUE, FALSE, UNIX?, WIN32?

Function: ARG()

Syntax: `a=ARG(z#)`

DESCRIPTION:

Returns the argument of a complex value z#. Any complex number a# can be expressed as: a#=ABS(a#)*EXP(1i*ARG(a#))

The argument is the phase angle of the complex number. The return value is a real value in the range of [-PI,PI].

EXAMPLE:

```
PRINT DEG(ARG(4+4i))   !    Result: 45
```

SEE ALSO: `IMAG()`, `REAL()`, `ABS()`, `ATAN2()`, `DEG()`

Function: ARID$()

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

DESCRIPTION:

The arid$()-takes a string argument and returns the order-0 adaptive arithmetic decoding of that string.

Arithmetic coding is a form of entropy encoding used in lossless data compression.

EXAMPLE:

```
        t$="Hello, this is a test!!!! This shows arithmetric coding and decoding with X11-Basic."
        b$=arie$(t$)  ! encode it
        print "The string was compressed to ";round(len(b$)/len(t$)*1000)/10;"%"
        c$=arid$(b$)  ! decode it
        print c$
  Result:
The string was compressed to 88.1%
        Hello, this is a test!!!! This shows arithmetric coding and decoding with X11-Basic.
```

SEE ALSO: ARIE\$()

Function: ARIE$()

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

DESCRIPTION:

The ARIE$() takes a string argument and returns the order-0 adaptive arithmetic encoding of that string.

Frequently used characters will be stored with fewer bits and not-so-frequently occurring characters will be stored with more bits, resulting in fewer bits used in total.

EXAMPLE:

```
        t$="Hello, this is a test!!!! This shows arithmetric coding and decoding with X11-Basic."
        b$=ARIE$(t$)  ! encode it
        print "The string was compressed to ";round(len(b$)/len(t$)*1000)/10;"%"
        c$=arid$(b$)  ! decode it
        print c$
  Result:
The string was compressed to 88.1%
        Hello, this is a test!!!! This shows arithmetric coding and decoding with X11-Basic.
```

SEE ALSO: ARID\$()

Command: ARRAYCOPY

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

DESCRIPTION:

Copies 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. It can be used to give all array elements a defined value, e.g. just after dimensioning the array where the contents are yet undefined.

EXAMPLE:

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

SEE ALSO: DIM

Function: ARRPTR()

Syntax: `adr%=ARRPTR(a())`

DESCRIPTION:

Finds the address of the descriptor of an array.

EXAMPLE:

```
DIM a(100,4)
adr%=ARRPTR(a())
PRINT "Array has dimension: ";LPEEK(adr%)
PRINT "Array index list: ";
FOR i%=0 TO LPEEK(adr%)
  PRINT LPEEK(LPEEK(adr%+4)+4*i%);
  IF i%<LPEEK(adr%)
    PRINT ",";
  ENDIF
NEXT i%
PRINT
```

SEE ALSO: `VARPTR(),UBOUND(),DIM?`

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.

EXAMPLE:

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

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

Function: ASIN()

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

DESCRIPTION:

The ASIN() is the arc sine function, i.e. the inverse of the SIN() function. Or, more elaborate: It Returns the angle (in radian, not degrees !), which, fed to the sine function will produce the argument passed to the ASIN() function.

EXAMPLE:

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

SEE ALSO: ACOS(), SIN()

*

Function: ASINH()

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

DESCRIPTION:

The ASINH() function calculates the inverse hyperbolic sine of x, i.e. the inverse of the SINH() function. It returns the angle (in radian), which, fed to the hyperbolic sine function will produce the argument passed to the ASINH() function.

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

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 1 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 COLS and ROWS variables. To get the actual position of the text cursor you may use the CRSCOL and CRSLIN variables.

EXAMPLE:

```
PRINT AT(3,1);"   This is a Title   "
GPRINT AT(4,7);"Test"
```

SEE ALSO: PRINT, GPRINT, TAB(), SPC(), COLS, ROWS, CRSLIN, CRSCOL, LOCATE

Function: ATN(), ATAN()

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

DESCRIPTION:

ATN() and ATAN() both return the angle in radians, for the inverse tangent of the expression.

EXAMPLE:

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

SEE ALSO: ACOS(),ASIN(),ATAN2()

*

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) equivalently 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>)`

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, ATANH() returns not-a-number (NaN).

EXAMPLE:

```
PRINT DEG(ATANH(-0.5))     ! Result: -31.47292373095
```

SEE ALSO: ATAN()

7.3. B

Function: BCHG()

Syntax: `b%=BCHG(x%,bit%)`

DESCRIPTION:

Changes the bit% bit of x% from 0 to 1 or from 1 to 0.

EXAMPLE:

```
PRINT BCHG(1,2)     ! result: 5
PRINT BCHG(5,2)     ! result: 1
```

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

*

Function: BCLR()

Syntax: `b%=BCLR(x%,bit%)`

DESCRIPTION:

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

EXAMPLE:

```
PRINT BCLR(7,1)     ! result: 5
```

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 #n,adr%,len%`

DESCRIPTION:

Reads len% bytes from a data channel into an area of memory starting at address adr%.

Unlike with BLOAD, several different areas of memory can be read from a file.

EXAMPLE:

```
OPEN "I",#1,"test.rsc"
header$=SPACE$(32)
BGET #1,VARPTR(header$),32
CLOSE #1
```

SEE ALSO: BLOAD, BPUT

Function: BIN$()

Syntax: `a$=BIN$(<num-expression>[,len%])`

DESCRIPTION:

The bin$()-takes a numeric argument and converts it into a string of binary digits (i.e. '0' and '1'). The minimal length of the output, the minimal number of digits, can be specified by the optional second argument. If the specified length is bigger than needed, the string is filled with leading zeros. If you need binary representations with sign, use RADIX$() instead.

EXAMPLE:

```
       PRINT BIN$(64,8),BIN$(-2000)
Result: 01000000      11111111111111111111100000110000
```

SEE ALSO: `HEX\$(),OCT\$(),RADIX\$()`

Command: BLOAD

Syntax: `BLOAD filename$,adr%`

DESCRIPTION:

BLOAD reads the specified file into memory at address adr%. The memory adr% 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.

EXAMPLE:

```
IF EXIST("test.dat")
  adr%=MALLOC(SIZE("test.dat"))
  BLOAD "test.dat",adr%
ENDIF
```

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.

EXAMPLE:

```
a=1
b=2
BMOVE VARPTR(a),VARPTR(b),8  ! same as b=a
```

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

Command: BOUNDARY

Syntax: `BOUNDARY flag%`

DESCRIPTION:

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

EXAMPLE:

```
BOUNDARY FALSE
```

SEE ALSO: PBOX, PCIRCLE

Command: BOX

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

DESCRIPTION:

Draws a rectangle with corners at (x,y) and (x2,y2). The screen coordinates start in the upper left corner. X increases to the right and y down to the bottom of the screen or window.

EXAMPLE:

```
COLOR COLOR_RGB(1,1,0)
BOX 20,20,620,380
```

SEE ALSO:

PBOX, GET_GEOMETRY

Command: BPUT

Syntax: `BPUT #n,adr%,len%`

DESCRIPTION:

Writes len% bytes from an area of memory starting at adr% out to a data channel #n.

EXAMPLE:

```
OPEN "O",#1,"test.dat"
BPUT #1,VARPTR(t$),LEN(t$)
CLOSE #1
```

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).

EXAMPLE:

```
DO
  INC i
  IF i>5
    PRINT "i is big enough."
    BREAK
  ENDIF
LOOP
```

SEE ALSO: `EXIT IF`

Command: BSAVE

Syntax: `BSAVE filename$,adr%,len%`

DESCRIPTION:

Save len% bytes in memory from address adr% to a file named filename$. If filename does not exist, it will be created. If it does exist, the old content will be overwritten. 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.

EXAMPLE:

```
BSAVE "content-t.dat",VARPTR(t$),LEN(t$)
```

SEE ALSO: BLOAD, BPUT

Function: BSET()

Syntax: `b%=BSET(x%,bit%)`

DESCRIPTION:

BSET sets the bit%-th bit of x% to 1.

EXAMPLE:

```
PRINT BSET(0,2)    ! result: 4
```

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

Function: BTST()

Syntax: `<bool-result>=BTST(x%,bit%)`

DESCRIPTION:

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

EXAMPLE:

```
PRINT BTST(4,2)    ! result: -1
```

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

Function: BWTD$()

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

DESCRIPTION:

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

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

BWTD$() can restore the original content of a string which has been coded with BWTE$() before.

EXAMPLE:

```
        t$="Hello, this is the Burrows Wheeler transformation!"
        b$=bwte$(t$)  ! encode it
        print b$
        c$=bwtd$(b$)  ! decode it
        print c$

Result:
esss,rno ! rmhheHlstWtth eelroalifretoruwiin  a Bo
Hello, this is the Burrows Wheeler transformation!
```

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.

EXAMPLE:

```
t$="Hello, this is the Burrows Wheeler transformation!"
b$=bwte$(t$)  ! encode it
print b$
c$=bwtd$(b$)  ! decode it
print c$

Result:
esss,rno ! rmhheHlstWtth eelroalifretoruwiin  a Bo
Hello, this is the Burrows Wheeler transformation!
```

SEE ALSO: BWTD\$()

Function: BYTE()

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

DESCRIPTION:

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

EXAMPLE:

```
PRINT BYTE(-200)    ! Result: 56
```

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

7.4. C

Command: CALL

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

DESCRIPTION:

Calls a machine code or C subroutine at address <adr> without return value. Optional 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, pointer (4-Bytes)

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
```

SEE ALSO: `CALL(), EXEC`

*

Function: CALL()

Syntax: `ret%=CALL(adr%[,<parameter-list>])`

DESCRIPTION:

Calls a machine code or C subroutine at address <adr%> and returns an integer value ret%. Optional 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, pointer (4-Bytes)

EXAMPLE:

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

SEE ALSO: CALL, EXEC

Function: CARD()

Syntax: `a%=CARD(b%)`

DESCRIPTION:

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

EXAMPLE:

```
PRINT CARD(-200)     ! Result: 65336
```

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

Keyword: CASE

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

DESCRIPTION:

CASE takes a list of expressions to be compared with the expression of the corresponding SELECT statement.

EXAMPLE:

```
i=5
SELECT i
  CASE 1
    PRINT 1
  CASE 2,3,4
    PRINT "its 2,3, or 4"
  CASE 5
    PRINT 5
  DEFAULT
    PRINT "default"
ENDSELECT
```

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.

EXAMPLE:

```
PRINT CBRT(8)     ! Result: 2
```

SEE ALSO: SQRT()

Function: CEIL()

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

DESCRIPTION:

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

EXAMPLE:

```
PRINT CEIL(-1.5), CEIL(0.5)   ! result: -1  1
```

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: LOAD, MERGE, RUN

Command: CHDIR

Syntax: CHDIR <path-name>

DESCRIPTION:

CHDIR changes the current working directory to the directory specified in path-name.

EXAMPLE:

```
CHDIR "/tmp"
```

SEE ALSO: MKDIR, RMDIR, DIR\$()

Command: CHMOD

Syntax: `CHMOD <file-name>, <mode>`

DESCRIPTION:

CHMOD changes the permissions of a file. The new file permissions are specified in mode, which is a bit mask created by ORing (or adding) together zero or more of the following:

```
        -     1 execute/search by others ("search" applies for directories
       and means that entries within the directory can be accessed)
-     2 write by others
-     4 read by others
-     8 execute/search by group
- 0x010 write by group
        - 0x020 read by group
- 0x040 execute/search by owner
        - 0x080 write by owner
- 0x100 read by owner
- 0x200 sticky bit
- 0x400 set-group-ID
- 0x800 set-user-ID
```

EXAMPLE:

```
CHMOD "/tmp/file",0x1e8
```

SEE ALSO: OPEN

Function: CHR$()

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

DESCRIPTION:

CHR$() returns the character associated with a given ASCII code. If the argument is in the range of 0-255 it produces exactly one byte.

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  \verb|^|    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
```

COMMENT: You should avoid to pass an argument outside of the range 0-255 for compatibility reasons. Currently only the lowest 8 bits are taken, but in future the function could be extended to also produce unicode charackters (up to three bytes) taking the unicode values (0-0xffff).

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 are given in screen coordinates and pixels. 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 background color, which can be specified by COLOR.

EXAMPLE:

```
foreground=COLOR_RGB(1,1,1) ! white
background=COLOR_RGB(0,0,1) ! blue
COLOR foreground,background
CLEARW
SHOWPAGE
```

SEE ALSO: CLOSEW, COLOR

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 defines the clipping rectangle starting at the upper left coordinates x,y and extends w pixels wide and h high. The optional additional command parameters ox,oy make it possible to redefine the origin of the graphic display.

COMMENT: This command is still buggy. Do not use it.

EXAMPLE:

```
CLIP 0,0,100,100,50,50
CIRCLE 0,0,55
SHOWPAGE
```

SEE ALSO:

Command: CLOSE

Syntax: `CLOSE [[#]n[,[#]<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, all 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. This command has no effect on Android.

SEE ALSO: `CLEARW`

Command: CLR

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

DESCRIPTION:

Clear the variables given in the list. Sets specified variables or arrays to 0 or "".

EXAMPLE:

```
CLR a,t$,i%,b()
```

SEE ALSO: ARRAYFILL

Command: CLS

Syntax: `CLS`

DESCRIPTION:

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

EXAMPLE:

```
CLS
PRINT "This is now a title line on an empty text screen."
```

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 dependent of the color depth of the Screen. Usually the COLOR statement is used together with the COLOR_RGB() function, so arbitrary colors may be used.

EXAMPLE:

```
yellow=COLOR_RGB(1,1,0)
blue=COLOR_RGB(0,0,1)
COLOR yellow,blue
```

SEE ALSO: COLOR_RGB(), LINE

Function: COLOR_RGB()

Syntax: `c%=COLOR_RGB(r,g,b[,a])`

DESCRIPTION:

COLOR_GRB() returns a color number for the specified color. The rgb-values range from 0 (dark) to 1.0 (bright). The returned number depends on the screen depth of the bitmap used. 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 optional parameter a is the alpha value (0...1), which will be used if it is supported by the graphics system.

The color numbers may be passed to the COLOR command.

EXAMPLE:

```
yellow=COLOR_RGB(1,1,0)
COLOR yellow
```

SEE ALSO: COLOR

Variable: COLS

Syntax: `n%=COLS`

DESCRIPTION:

Returns the number of colums of the text terminal (console).

EXAMPLE:

```
PRINT COLS, ROWS     ! Result: 80      24
```

SEE ALSO: ROWS, PRINT AT(), CRSCOL, CRSLIN

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!).

EXAMPLE:

```
PRINT COMBIN(49,6)    ! result:  13983816
```

SEE ALSO: FACT(), VARIAT()

Function: COMPRESS$()

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

DESCRIPTION:

Performs a lossless 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$(

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

Function: CONJ()

Syntax: `x#=CONJ(z#)`

DESCRIPTION:

Returns the complex conjugate value of z#. That is the value obtained by changing the sign of the imaginary part.

EXAMPLE:

```
PRINT CONJ(1-2i)     Result: (1+2i)
```

SEE ALSO: IMAG(), REAL()

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 successfully connect only once; connectionless protocol sockets may use connect multiple times to change their association.

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

Command: CONTINUE

Syntax:
```
CONT
CONTINUE
```

DESCRIPTION:

This command has two different use cases. If used in direct mode, it continues the execution of a program after interruption (e.g. with STOP).

If used inside a SELECT/ENDSELECT block, it branches to the line following the next CASE or DEFAULT directive. If no CASE or DEFAULT statement is found, it branches to ENDSELECT.

EXAMPLE:

```
INPUT a
SELECT a
CASE 1
  PRINT 1
  CONTINUE
CASE 2
  PRINT "1 or 2"
CASE 3
  PRINT 3
DEFAULT
  PRINT "default"
ENDSELECT
```

SEE ALSO: STOP, SELECT, CASE, DEFAULT, BREAK

Command: COPYAREA

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 (this means on the screen directly without datatransfer to the program).

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

Function: COS()

Syntax:
```
b=COS(x)
z#=COS(x#)
```

DESCRIPTION:

Returns the Cosine of the expression in radians. Also returns the complex cosine of a complex expression. The complex cosine function is defined as:
cos#(z) := (exp#(1i*z#)+exp#(-1i*z#))/2

EXAMPLE:

```
PRINT COS(0)         ! Result: 1
PRINT COS#(0+1i)     ! Result: 1.543080634815+0i
```

SEE ALSO: SIN(),ASIN()

*

Function: COSH()

Syntax: `b=COSH(x)`
`z#=COSH(x#)`

DESCRIPTION:

The cosh() function returns the hyperbolic cosine of x, which is defined mathematically as (exp(x)+exp(-x))/2 Also returns the complex hyperbolic cosine of a complex number or expression.

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

Function: CRC()

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

DESCRIPTION:

Calculates a 32 bit checksum on the given string. Optionally another checksum can be passed as oc. 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 useful for benchmark applications on multi-tasking environments.

COMMENT: The UNIX standard allows for arbitrary values at the start of the program; subtract the value returned from a CTIMER at the start of the program to get maximum portability. It is also not guaranteed, that the values will not repeat itself. On a 32-bit system this function will return the same value approximately every 72 minutes.

EXAMPLE:

```
t=CTIMER
FOR i=0 TO 100000
  NOOP
NEXT i
ref=(CTIMER-t)/100000
print "Ref=",str$(ref*1000,5,5);" ms"
```

SEE ALSO: TIMER, STIMER

Command: CURVE

Syntax: `CURVE x0,y0,x1,y1,x2,y2,x3,y3`

DESCRIPTION:

The CURVE command draws a cubic Bezier-curve. The Bezier-curve starts at x0,y0 and ends at x3,y3. The curve at x0,y0 is at a tangent with a line from x0,y0 to x1,y1; and at x3,y3 is at a tangent with a line between x3,y3 and x2,y2.

SEE ALSO: `LINE`, `POLYLINE`

Function: CVA()

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

DESCRIPTION:

Returns array reconstructed from the string. This function is the complement 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 complement 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 a string. This function is the complement 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 a string. This function is the complement of MKI$(). Null string returns 0, For strings with only one byte length the ASCII value of that charackter will be returned.

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 a string. This function is the complement 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 a string. This function is the complement of MKS$().

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

7.5. 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. Strings not in quotes will be capitalzed.

SEE ALSO: READ, RESTORE

Variable: DATE$

Syntax: `d$=DATE$`

DESCRIPTION:

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

EXAMPLE:

```
PRINT TIME$,DATE$   ! 14:49:44       11.03.2014
```

SEE ALSO: TIME\$

Command: DEC

Syntax: `DEC <num-variable>`

DESCRIPTION:

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

SEE ALSO: `INC`

Function: DECLOSE$()

Syntax: `a$=DECLOSE$(t$)`

DESCRIPTION:

Removes enclosing characters from string t$. De-closing a string, following pairs are recognized: "" , ” , <> , () , , [], ´‘ If the string was not enclosed with one of these pairs of characters, the string will be returned unmodified.

SEE ALSO: ENCLOSE\$()

Function: DECRYPT$()

Syntax: `t$=DECRYPT$(message$,key$[,typ%])`

DESCRIPTION:

Decrypts a message, which has been encrypted with ENCRYPT$() before.
COMMENT: This function is only available if libgcrypt was compiled in.

SEE ALSO: `ENCRYPT\$()`

Keyword: DEFAULT

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

DESCRIPTION:

See SELECT.

SEE ALSO: `SELECT`

Command: DEFFILL

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

DESCRIPTION:

Sets fill color 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 @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 Long Dash dot dot .. 0x11-0xffffffff 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 The userdefined style of the line defines a dash-pattern in the nibbles: 0x11 means: 1 pixel dash, followed by 1 pixel gap. 0x61 means: 1 pixel dash, followed by 6 pixel gap. 0x6133 means: 3 pixel dash, followed by 3 pixel gap, followed by a 1 pixel dash followed by a 6 pixel gap.

SEE ALSO: LINE, DEFFILL

Command: DEFMARK

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

DESCRIPTION:

Sets color, 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 types are 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: DEFMOUSE

Syntax: `DEFMOUSE <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 flag%,width,height,angle`

DESCRIPTION:

Defines the style, rotation and size of the line font used by the LTEXT command. COLOR and linestyles (e.g. thickness) can be set with COLOR and DEFLINE.

flag% : text style - 0=monospace, 1=normal angle : rotation in degrees width and height : size of text in % (100% corresponds to 100 Pixel font)

EXAMPLE:

```
DEFTEXT 0,0.05,0.1,0      ! Size of the charackters is approx 10x5 pixels
LTEXT 100,100,"Hello"
```

SEE ALSO: LTEXT, TEXT, COLOR, DEFLINE

Function: DEG()

Syntax: `d=DEG(x)`

DESCRIPTION:

Converts x from radians to degrees.

EXAMPLE:

```
PRINT DEG(PI)        ! Result: 180
```

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) (two-dimensional) matrix a().

The determinant provides important information about a matrix of coefficients of a system of linear equations. The system has a unique solution exactly when the determinant is nonzero. When the determinant is zero there are either no solutions or many solutions.

EXAMPLE:

```
a()=[3,7,3,0;0,2,-1,1;5,4,3,2;6,6,4,-1]
PRINT DET(a())      ! Result: 105
```

SEE ALSO: SOLVE(), INV()

Function: DEVICE()

Syntax: `d=DEVICE(filename$)`

DESCRIPTION:

Returns the device id corresponding to a file.

Command: DIM

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

DESCRIPTION:

Sets the dimensions of an array or string array. Arrays can be re-dimensioned any time.

COMMENT: The argument determines the number of Entries in the array. The index count starts with 0. So DIM a(10) will produce the elements a(0), a(1), ... a(8), and a(9) (10 elements). Note: a(10) does not exist!

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.

EXAMPLE:

```
DIM a(10,10)
PRINT DIM?(A())     Result: 100
```

SEE ALSO:

`DIM`

Function: DIR$()

Syntax: `p$=DIR$(0)`

DESCRIPTION:

DIR$() returns the path of the current directory. The otional argument is ignored.

SEE ALSO: CHDIR, ENV\$()

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. DOWNTO indicates that the loop should count backwards. e.g.: FOR c=100 DOWNTO 1 is the same as FOR c=100 TO 1 STEP -1

EXAMPLE:

```
FOR i=10 DOWNTO 0
  PRINT i
NEXT i
```

SEE ALSO: FOR, TO, NEXT, STEP

Function: DPEEK()

Syntax: `value%=DPEEK(adr%)`

DESCRIPTION:

Reads 2 bytes from address adr% (a word).

EXAMPLE:

```
t$=MKI$(4711)
PRINT DPEEK(VARPTR(t$))
```

SEE ALSO: PEEK(), LPEEK(), DPOKE, MKI\$()

*

Command: DPOKE

Syntax: `DPOKE adr%,value%`

DESCRIPTION:

Writes value% as a 2 byte word to address adr%.

EXAMPLE:

```
t$=SPACE$(2)
DPOKE VARPTR(t$),4711
PRINT CVI(t$)
```

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

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: `DUMP [t$][,#n]`

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 all X11-Basic commands DUMP "F" – list of all X11-Basic functions
If a open file channel is giveb, DUMP outputs to that file.

EXAMPLE:

```
OPEN "O",#1,"debug.txt"
PRINT #1,"Variables:"
DUMP "",#1
PRINT #1,"Labels:"
DUMP ":",#1
CLOSE #1
```

SEE ALSO: LIST, PLIST, HELP

7.6. 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 standard 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
```

Function: ENCLOSE$()

Syntax: `e$=ENCLOSE$(t$[,c$])`

DESCRIPTION:

Encloses a string. With a character or a pair of characters. The default pair is ""

EXAMPLE:

```
PRINT enclose$("abc","()")  ! Result: (abc)
       PRINT enclose$("Hello","-") !   Result: -Hello-
```

SEE ALSO: `DECLOSE\$()`

Function: ENCRYPT$()

Syntax: `e$=ENCRYPT$(t$,key$[,typ%])`

DESCRIPTION:

This Function will encrypt a string with a given key. Typ% specifies, which algorithm is used. If typ% is not specified, the blowfish algorithm is used.

The encrypted message can be decrypted again using DECRYPT$() and the same key (or, in case it was encrypted with a public key, it must be decrypted with the corresponding private key.) The encrypted message has always the same length than the original message.

Following algorithms can be used: Typ%= 1 ! IDEA 2 ! 3DES 3 ! CAST5 4 ! BLOWFISH 5 ! SAFER_SK128 6 ! DES_SK 7 ! AES 8 ! AES192 9 ! AES256 10 ! TWOFISH 301 ! ARCFOUR Fully compatible with RSA's RC4 (tm). 302 ! DES this is single key 56 bit DES. 303 ! TWOFISH128 304 ! SERPENT128 305 ! SERPENT192 306 ! SERPENT256 307 ! RFC2268_40 Ron's Cipher 2 (40 bit). 308 ! RFC2268_128 Ron's Cipher 2 (128 bit). 309 ! SEED 128 bit cipher described in RFC4269. 310 ! CAMELLIA128 311 ! CAMELLIA192 312 ! CAMELLIA256

501 ! RSA 516 ! ELG_E 517 ! DSA 520 ! ELG 801 ! ECDSA 802 ! ECDH

COMMENT: This function is only available if libgcrypt was compiled in.

SEE ALSO: `COMPRESS\$(),DECRYPT\$()`

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 block. The function itself must return a value 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: ENDPROCEDURE

Syntax: `ENDPROCEDURE`

DESCRIPTION:

Terminates a user defined procedure. It has the same effect as RETURN.

SEE ALSO: `RETURN`, `ENDFUNCTION`

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.

EXAMPLE:

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

SEE ALSO: SETENV

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

EXAMPLE:

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

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

Command: ERASE

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

DESCRIPTION:

un-DIMs an array and removes it from the internal variables. Or remove a variable out of the memory. (This command need never be used in X11-Basic. Don't use it. An array can easily re-dimensioned with another DIM statement.)

SEE ALSO: `DIM`, `CLR`

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 message 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$
&a$      ! short form
```

SEE ALSO: `EVAL(), &`

*

Function: EVAL()

Syntax: `a=EVAL(b$)`

DESCRIPTION:

Evaluate expression, which is in b$.

EXAMPLE:

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

result=&b$     ! short form
```

SEE ALSO: EVAL, &

Function: EVEN()

Syntax: `e%=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,xr,yr,s,k,ks,t$,timestamp]`

DESCRIPTION:

EVENT waits for an event of the graphics i/o system. This very powerful command can wait for a big variaty of different user events which can occur. Following events can be watched for: - A mouse button is pressed or released, - A key on the keyboard is pressed or released, - The mouse pointer has moved to a new position. - The graphics window was clicked to be opened, closed or iconified.

typ determines which of the events have occured:

typ=2 — key pressed typ=3 — key released typ=4 — mouse button pressed typ=5 — mouse button released typ=6 — mouse motion event typ=10 — Window move event typ=13 — Window resize event

x,y — Mouse position relative to window xr,yr — Mouse position relative to screen or relative movement s — State of the Alt, Caps, Shift keys k — state of the mouse buttons or keycode ks — scancode of key t$ — Character of pressed key timestamp — timestamp of the time the event occured (in ms)

SEE ALSO: `KEYEVENT`, `MOUSEEVENT`, `MOTIONEVENT`, `EVENT?()`

Command: EVENT?()

Syntax: `a=EVENT?(mask%)`

DESCRIPTION:

Returns TRUE if a graphics event is pending which matches the types given by mask.

mask=1 — key press event mask=2 — key release event mask=4 — mouse button press event mask=8 — mouse button release event mask=0x40 — mouse motion event mask=0x40000 — window resize events mask=0x200000 — window focus change events

SEE ALSO: EVENT, KEYEVENT, MOUSEEVENT, MOTIONEVENT, INP?()

Command: EVERY

Syntax:
```
EVERY <seconds>, <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.

COMMENT: EVERY CONT and EVERY STOP are currently not implemented. Please also read the comments about AFTER.

EXAMPLE:

```
EVERY 1,progress
q=10000000
FOR p=0 TO q
  a=(1+a)/2
NEXT p
AFTER 1,progress  ! To stop the progress
PAUSE 3           ! will be interrupted after 1 second
PRINT "done -->";a
END
PROCEDURE progress
  PRINT p/q;"% done."
RETURN
```

SEE ALSO: AFTER

Command: EXEC

Syntax:
```
EXEC name$[,commandline$[,environment$]]
EXEC action$[,data$[,extra$]]
```

DESCRIPTION:

Calls an operating system service by name. The behavior is different on different operating systems.

The first argument for EXEC is the name of a file or a service that is to be executed.

If name$ is not an android-intent, the system searches for an executable file if the specified filename does not contain a slash (/) character.

The file is sought in the colon-separated list of directory pathnames specified in the PATH environment variable. If this variable isn't defined, the path list defaults to the current directory followed by the list of directories "/bin:/usr/bin".

If the specified filename includes a slash character, then PATH is ignored, and the file at the specified pathname is executed.

The filename must be either a binary executable, or a script starting with a line of the form:

#! interpreter [optional-arg]

In case, name$ is not an android intent, EXEC replaces the current process image with a new process image and execute it. So EXEC will not return to the X11-Basic program.

The following string argument describes a list of one or more arguments available to the executed program. The Arguments must be separated by a newline character (CHR$(10)).

The second string argument allows the caller to specify the environment of the executed program. The environment consists of a list of strings of format VAR=CONTENT, separated by a newline character (CHR$(10)).

If this argument is not present, EXEC takes the environment for the new process image from the calling process.

On Android:

If the name starts with "android.intent", a special operating system service, called INTENT is called. Following intents are currently available:

android.intent.action.EDIT — call a text editor android.intent.action.SEND — call email client android.intent.action.VIEW — call browser android.intent.action.DIAL — call phone

Execution will continue in X11-Basic as soon as the intent has finished. If you need a return value to find out if the intent was successful, use EXEC().

EXAMPLE:

```
EXEC "env","-u"+chr$(10)+"A","HOME=/tmp"+chr$(10)+"A=0"
EXEC "android.intent.action.VIEW",\
      "text/html:http://x11basic.sourceforge.net/"
EXEC "android.intent.action.EDIT","text/plain:new.bas"
EXEC "android.intent.action.SEND","message/rfc822:", \
"android.intent.extra.EMAIL=my@email.adr"+chr$(10)+\
"android.intent.extra.SUBJECT=Hello"
```

SEE ALSO: SYSTEM, EXEC(), CALL, SHELL

*

Function: EXEC()

Syntax:

```
<int-return>=EXEC(name$[,commandline$[,environment$]])
<int-return>=EXEC(action$[,data$[,extra$]])
```

DESCRIPTION:

Does the same as the command EXEC, but returns a return value. This is either the value passed to the system exit() function or the result of an intent.

EXAMPLE:

```
a=EXEC("android.intent.action.EDIT","text/plain:new.bas")
if a=-1
  print "OK."
else if a=0
  print "CANCELED"
endif
        b=EXEC("/usr/bin/busybox","/usr/bin/busybox"+chr$(10)+"-c","HOME=/tmp")
if b<>0
  print "program exited with return code: ";b
endif
```

SEE ALSO: CALL(), EXEC, SYSTEM\$()

Function: EXIST()

Syntax: `<boolean-result>=EXIST(<filename>)`

DESCRIPTION:

Returns TRUE (-1) if the file is present on a file system.

SEE ALSO: OPEN

Command: EXIT

Syntax: `EXIT`

DESCRIPTION:

The command EXIT will either exit a loop, return from a procedure or subroutine, or quit the interpreter. WHILE, REPEAT, DO and FOR loops can be aborted prematurely with the EXIT command. Here it has the same fuinction as BREAK. EXIT leaves the current (innermost) loop immediately. Outside a loop, but insidde a procedure or function, that procedure or function is left, like with RETURN. Outside any procedure or function, or invoked from the direct mode EXIT has the same effect like QUIT.

SEE ALSO: `EXIT IF`, `BREAK`, `RETURN`, `QUIT`

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.

EXAMPLE:

```
PRINT EXPM1(1)    !    Result: 1.718281828459
```

SEE ALSO: LOG1P(), EXP()

7.7. F

Function: FACT()

Syntax: `a&=FACT(n%)`

DESCRIPTION:

Calculates the factorial (n!). The factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n. The value of 0! is 1, according to the convention.

EXAMPLE:

```
PRINT FACT(5)   ! Result: 120
PRINT FACT(10)  ! Same as 10*9*8*...*2*1
Result: 3628800
PRINT FACT(50) ! Same as 50*49*48*...*2*1
Result: 30414093201713378043612608166064768844377641568960512000000000000
```

SEE ALSO: COMBIN(), VARIAT(), GAMMA()

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`

Variable: FATAL

Syntax: FATAL

DESCRIPTION:

Returns the value 0 or -1 according to the type of error. On normal errors the function returns 0. The value -1 is returned on all errors where the address of the last executed command is no longer known. In this case a RESUME is not possible anymore.

COMMENT: This variable is currently not used in X11-Basic.

SEE ALSO: RESUME

Command: FFT

Syntax: `FFT a()[,flag%]`

DESCRIPTION:

FFT calculates the discrete Fourier Transformation of a real periodic sequence stored in the float array a(). If flag% is <>0 the back transform is calculated. The result replaces the contents of a(). The method used is most efficient (=fast) when DIM?(a()) is a product of small primes.

This transform is un-normalized since a call of FFT followed by a call of FFT ,-1 will multiply the input sequence by DIM?(a()).

The output consists of an array with the Fourier coefficients as follows: For n=DIM?(a()) even and for i = 0,...,n-1 a(i) = a(0)+(-1)^(i)*a(n-1) plus the sum from k=2 to k=n/2 of

2*a(2*k-1)*cos((k-1)*i*2*pi/n) -2*a(2*k)*sin((k-1)*i*2*pi/n)

for n odd and for i = 0,...,n-1

a(i) = a(0) plus the sum from k=2 to k=(n+1)/2 of

2*a(2*k-1)*cos((k-1)*i*2*pi/n) -2*a(2*k)*sin((k-1)*i*2*pi/n)

COMMENT:

Two succeeding FFT (or FFT ,-1) calculations are faster if they use the same size of the array.

EXAMPLE:

```
l=2^10   ! It is faster to use a power of two
DIM a(l)
FOR i=0 TO l-1
```

```
  a(i)=200/100*@si(3*i/512*2*pi)+i/100*sin(20*i/512*2*pi)
NEXT i
SCOPE a(),1,-10,300  ! Draw the function
FFT a()              ! Do the Fourier transformation
' Normalize
FOR i=0 TO l-1
  a(i)=a(i)/SQRT(l)
NEXT i
' Clear some of the frequencies
FOR i=4 TO 86
  a(i)=0
NEXT i
FFT a(),-1           ! Do a back transformation
SCOPE a(),0,-10/SQRT(l),300  ! Draw the result (scaling=normalization)

DEFFN si(x)=x mod pi
```

SEE ALSO: `FFT()`

*

Function: FFT

Syntax: `b()=FFT(a()[,flag%])`

DESCRIPTION:

FFT calculates the discrete Fourier Transformation of a real periodic sequence stored in the float array a(). If flag% is <>0 the back transform is calculated. Unlike the command FFT, FFT() returns an array with the Fourier transform leaving the original array untouched.

SEE ALSO: `FFT()`

Function: FILEEVENT$

Syntax: `t$=FILEEVENT$`

DESCRIPTION:

Returns a string with event information on watched files and directories. If the string is empty, no events are pending. The events consist of 3 characters followed by a blank and optionally followed by a filename. The first three characters have the following meaning: 1st: "d" means: the file is a directory 2nd: "X" created, "O" OPENed, "C" closed, "M" moved, "D" deleted 3rd: "r" read, "w" write, "a" attributes were changed

When monitoring a directory, the events above can occur for files in the directory, in which case the name field in the returned string identifies the name of the file within the directory.

SEE ALSO: `WATCH`

EXAMPLE:

```
WATCH "/tmp"
DO
  a$=FILEEVENT$
  IF LEN(a$)
    PRINT a$
  ENDIF
LOOP
```

Function: FIB()

Syntax: `w&=FIB(i%)`

DESCRIPTION:

Returns the i'th Fibonacci number.

COMMENT: This function works only in the interpreter and only when used in a direct assignment to a big integer variable.

EXAMPLES:

```
w&=FIB(100)  --> Result: 354224848179261915075
```

SEE ALSO: LUCNUM()

Command: FILESELECT

Syntax: `FILESELECT title$,<path$>,<default$>,selectedfile$`

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$ 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 appear in the selection line. ("" for no default).

FILESELECT returns the selected filename (including path) in selectedfile$. 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 other color than the one of the starting coordinate. The fill color can be chosen with the command COLOR.

SEE ALSO: `COLOR`

Command: FIT

Syntax: `FIT x(),y(),yerr???,???[,???,???,???,???,???,???]`

DESCRIPTION:

Fits a user defined function to a set of data points, also using errorbars in y. TODO: The command needs to be described.

SEE ALSO: `FIT\_LINEAR`, `FIT\_POLY`

Command: FIT_LINEAR

Syntax: `FIT_LINEAR x(),y(),n%,a,b[,da,db,chi2,dy(),dx(),q]`

DESCRIPTION:

FIT_LINEAR calculates a linear regression to fit a straight line f(x)=a+b*x to the data x(),y(). n%=number of points. The fitted values are stored in a and b. If specified, the uncertainty of a and b are stored in da,db and chi2. Optional errors of the datapoints in x and y can be given by dy(),dx(). In this case q has a meaning.

EXAMPLE:

```
n=400
DIM x(n),y(n)
FOR i=0 TO n-1
  x(i)=(i+RANDOM(10))/400
  y(i)=(1*i+GASDEV(1)*20+50-i/30*i/30+(400-MOUSEY))/400
NEXT i
FIT_LINEAR x(),y(),,a,b,da,db,chi
PRINT "chi2=";chi
da=da*SQRT(chi/(DIM?(x())-2))
db=db*SQRT(chi/(DIM?(x())-2))
        PRINT a;"+/-";da
        PRINT b;"+/-";db
```

SEE ALSO: FIT, FIT_POLY

Command: FIT_POLY

Syntax: `FIT_POLY x(),y(),dy(),n%,a(),m%`

DESCRIPTION:

FIT_POLY fits a polynom of order m% to n% datapoints given by x() and y(). If present, dy() specifies the errors in y. The polynomial coefficients are returned in a() such that f(x)=a(0)+a(1)*x+a(2)*x^2+...

COMMENT: For higher orders m%>3 of the polynom, the range of x() must be small (in the order of 1) otherwise the algorithm can become unstable because of the high powers.

No information about the quality of the fit is returned.

SEE ALSO: `FIT`, `FIT\_LINEAR`, `SOLVE`

Function: FIX()

Syntax: `a=FIX(x)`

DESCRIPTION:

Returns the integer of x after it has been rounded. 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>=<expression> TO <expression> [STEP <expression>
FOR <variable>=<expression> DOWNTO <expression> [STEP <iexpres
```

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
```

SEE ALSO: SPAWN

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][Buttons]" where i = the required alert symbol - see ALERT. FORM_ALERT returns the number of the selected Button.

EXAMPLE:

```
~FORM_ALERT(1,"[0][This is my message!][OK]")
```

SEE ALSO: ALERT

Function: FORM_CENTER()

Syntax: `~FORM_CENTER(tree%,x,y,w,h)`

DESCRIPTION:

Centers the object 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. The returned value can safely be ignored.

EXAMPLE:

```
PROCEDURE info
  LOCAL adr%,x,y,w,h,ret%
  adr%=RSRC_GADDR(0,2)
  ~FORM_CENTER(adr%,x,y,w,h)
  ~FORM_DIAL(0,x,y,w,h,x,y,w,h)
  ~FORM_DIAL(1,x,y,w,h,x,y,w,h)
  ~OBJC_DRAW(adr%,0,-1,0,0)
  ret%=FORM_DO(adr%)
  ~FORM_DIAL(2,x,y,w,h,x,y,w,h)
  ~FORM_DIAL(3,x,y,w,h,x,y,w,h)
  IF ret%=35
    DPOKE adr%+ret%*24+10,0
  ENDIF
RETURN
```

SEE ALSO: OBJC_DRAW()

Function: FORM_DIAL()

Syntax: `ret%=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 occurred. 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 " " " " "

EXAMPLE:

```
PROCEDURE show_tabelle
  LOCAL ox,oy
  ox=(bx+bw-532)/2
  oy=(by+bh-242)/2
  ~FORM_DIAL(0,260,20,30,20,ox,oy,532,242)
  ~FORM_DIAL(1,260,20,30,20,ox,oy,532,242)
  @tabelle(ox,oy,532,242)
  MOUSEEVENT
  ~FORM_DIAL(2,260,20,30,20,ox,oy,532,242)
  ~FORM_DIAL(3,260,20,30,20,ox,oy,532,242)
RETURN
```

SEE ALSO: SGET, SPUT, GET, PUT, COPYAREA

Function: FORM_DO()

Syntax: `ret%=FORM_DO(tree%[,obj%])`

DESCRIPTION:

FORM_DO() manages an object tree. It interacts with the user until an object with EXIT or TOUCHEXIT 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).

EXAMPLE:

```
PROCEDURE info
  LOCAL adr%,x,y,w,h,ret%
  adr%=RSRC_GADDR(0,2)
  ~FORM_CENTER(adr%,x,y,w,h)
  ~FORM_DIAL(0,x,y,w,h,x,y,w,h)
  ~FORM_DIAL(1,x,y,w,h,x,y,w,h)
  ~OBJC_DRAW(adr%,0,-1,0,0)
  ret%=FORM_DO(adr%)
  ~FORM_DIAL(2,x,y,w,h,x,y,w,h)
  ~FORM_DIAL(3,x,y,w,h,x,y,w,h)
  IF ret%=35
    DPOKE adr%+ret%*24+10,0
  ENDIF
RETURN
```

SEE ALSO: OBJC_DRAW()

Function: FRAC()

Syntax: `a=FRAC(b)`

DESCRIPTION:

FRAC() returns the fractional part of its argument.

EXAMPLE:

```
PRINT FRAC(-1.234)
Result: -0.234
```

SEE ALSO: INT(), CINT(), TRUNC(), ROUND()

Command: FREE

Syntax: `FREE adr%`

DESCRIPTION:

Frees a previously allocated memory block.

SEE ALSO: `MALLOC()`

Function: FREEFILE()

Syntax: `a%=FREEFILE()`

DESCRIPTION:

FREEFILE() returns the first free filenumber available or -1 on error. the file numbers can be used together with other file functions like OPEN, LINK etc...

SEE ALSO: OPEN

Function: FSFIRST$()

Syntax: `a$=FSFIRST$(path$[,pattern$,attr$])`

DESCRIPTION:

FSFIRST$() searches for the first file in a filesystem of a given path path$, given match pattern pattern$ and given attributes. pattern$ can be a file name mask; default is "*". attr$ can be "d" for only list directories, "a" also list hidden files, "f" list regular files. The attributes can be combined. default is "df". If found, the filename and attributes are returned in a$. When path$ is empty, an empty string is returned. Otherwise the returned string consists of two words. E.g. "- filename.dat". The first word lists the attributes, the second word is the filename. Attributes can be "d" is a directory, "s" symbolic link, "-" a regular file.

EXAMPLE:

```
a$=FSFIRST$("/tmp","*.dat")
WHILE LEN(a$)
  PRINT WORD$(a$,2)
  IF LEFT$(a$)="d"
            PRINT "is a directory."
    ENDIF
  a$=FSNEXT$()
WEND
```

SEE ALSO: `FSNEXT\$()`

Function: FSNEXT$()

Syntax: `a$=FSNEXT$()`

DESCRIPTION:

FSNEXT$() searches for the next file in the filesystem specified by FSFIRST$(). When no more files can be found, an empty string is returned. Otherwise the returned string has the same meaning as the one returned by FSFIRST().

EXAMPLE:

```
a$=FSFIRST$("/tmp","*.dat")
WHILE LEN(a$)
  PRINT WORD$(a$,2)
  IF LEFT$(a$)="d"
            PRINT "is a directory."
    ENDIF
  a$=FSNEXT$()
WEND
```

SEE ALSO: `FSFIRST\$()`

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 preceded 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. (No Integer type functions with % are allowed.)

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.

EXAMPLE:

```
FUNCTION theta(x,a)
  IF x>a
    RETURN 0
  ELSE
    RETURN a
  ENDIF
ENDFUNCTION
```

SEE ALSO: ENDFUNCTION, RETURN, DEFFN, LOCAL, PROCEDURE

7.8. G

Function: GAMMA ( )

Syntax: `b=GAMMA(a)`

DESCRIPTION:

The Gamma function is defined by
GAMMA(x) = integral from 0 to infinity of t^(x-1) e^-t dt
It is defined for every real number except for non-positive integers. For non-negative integral m one has
GAMMA(m+1) = m!
and, more generally, for all x:
GAMMA(x+1) = x * GAMMA(x)
Furthermore, the following is valid for all values of x outside the poles:
Gamma(x) * Gamma(1 - x) = PI / sin(PI * x)

EXAMPLE:

```
PRINT GAMMA(4)    ! Result: 6
PRINT GAMMA(0)    ! Result: inf
```

SEE ALSO: `SIN()`, `COS()`, `LGAMMA()`

Function: GASDEV()

Syntax: `b=GASDEV(seed)`

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()`

Function: GCD()

Syntax: `c&=GCD(a&,b&)`

DESCRIPTION:

Returns the greatest common divisor of a and b. The result is always positive even if one or both input operands are negative. Except if both inputs are zero; then this function defines GCD(0,0)=0.

EXAMPLE:

```
PRINT GCD(120,200000)   ! Result: 40
```

SEE ALSO: LCM()

Command: GET

Syntax : GET x,y,w,h,var$[,bcolor]

DESCRIPTION:

GET puts a section of the graphic window or screen into a string variable (x,y,w,h define a rectangular portion of the screen). The coordinates must not lay outside of the screen. If the parameter bcolor is given, this color will be used as a transparency color to the bitmap. If no bcolor is given, no transparency will be created. The stored graphic can be put back on the screen with PUT.

COMMENT: The content of the string is bitmapdata. If saved into a file, the File will be in a common file format (currently .BMP 32bit-RGBA).

EXAMPLE:

```
GET 100,100,20,20,t$
PUT 200,200,t$        ! Put that portion at a different position
```

SEE ALSO:

PUT, COPY_AREA, SGET

Function: GET_COLOR()

Syntax: `<num-result>=GET_COLOR(<red-value>,<green-value>,<blue-value>)`

DESCRIPTION:

GET_COLOR() returns a color number for the specified color. The rgb-values range from 0 (dark) to 65535 (bright). The returned number depends on the screen depth. 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.

COMMENT: GET_COLOR does not support the rgba formt. Instead COLOR_RGB() should be used.

EXAMPLE:

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

COMMENT: This function should not be used anymore. Use COLOR_RGB() instead.
```

SEE ALSO: COLOR, COLOR_RGB()

Command: GET_GEOMETRY

Syntax: `GET_GEPMETRY <winnr>,<varx%>,<vary%>,<varw%>,<varh%>`

DESCRIPTION:

GET_GEOMETRY returns the size of the window or screen. The window needed to be opened before this command can be used. A SHOWPAGE will make sure, that the screen or window is allocated.

COMMENT: If this command is used on Android as one of the very first commands, a PAUSE 0.04 before will make sure, that the screen of the App has settled before the dimensions are taken.

EXAMPLE:

```
SHOWPAGE
GET_GEOMETRY ,x,y,w,h
```

SEE ALSO: SHOWPAGE

Command: GET_LOCATION

Syntax: `GET_LOCATION lat,lon,alt[,bearing,accuracy,speed,time[,provide`

DESCRIPTION:

GET_LOCATION returns various data from the location device. The location device can be a GPS or any other service which returns geo-information. The GPS needs to be turned on with GPS ON before.

Return values: lat – lattitude in degrees lon – longitude in degrees alt – altitude in meters

COMMENT: Works only on Android devices.

EXAMPLE:

```
GET_LOCATION x,y,a
```

SEE ALSO: GPS, GPS_LON, GPS_LAT, GPS_ALT

Command: GET_SCREENSIZE

Syntax: `GET_SCREENSIZE <varx%>,<vary%>,<varw%>,<varh%>`

DESCRIPTION:

GET_SCREENSIZE returns the size of the screen. This is the area where a window can be placed.

EXAMPLE:

```
GET_SCREENSIZE x,y,w,h
```

SEE ALSO: GET_GEOMETRY, MOVEW

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 specified 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("SA33333","*a[0-9]*",16)     -1
```

SEE ALSO: `INSTR(), WORT\_SEP`

Command: GOSUB ABBREV. @

Syntax: `GOSUB <procedure-name>[(<parameterlist>)]`

DESCRIPTION:

GOSUB initiates a jump to the procedure specified after GOSUB. The code reached that way must end with a RETURN statement which returns control to the calling line.

<parameterlist> contains expressions which are passed by value to local variables to the procedure. Variables can also be passed by reference (see the VAR statement). 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, VAR

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.
COMMENT: You should not jump into a procedure or FOR-NEXT loop. If you need to jump out of a loop it is better to use BREAK or EXIT IF.

EXAMPLE:

```
GOTO here
PRINT "never"
here:
PRINT "ever"
```

SEE ALSO: GOSUB, BREAK, EXIT IF

Command: GPRINT

Syntax: `GPRINT [[AT(),TAB(),SPC(),COLOR()]a$|b|const|USING|...;',]|`

DESCRIPTION:

The GPRINT statement writes all its arguments to the graphic window. It uses the same syntax as PRINT. Unlike PRINT the output goes to the graphic window, where a VT100-Terminal is emulated.

COMMENT: On Android PRINT and GPRINT share the same output screen, but use different terminals. This can lead to a mixture of charackters on the screen. Here do not use both, PRINT and GPRINT.

There is no INPUT for GPRINT. You would have to program it yourself using KEYEVENT.

EXAMPLE:

```
GPRINT chr$(27);"[2J";     ! Clear the graphics screen (like CLS for PRINT)
GPRINT AT(1,1);"This is a demo: ";1.23456 USING "##.##"
GPRINT "some more ...";
GPRINT " and more...";
GPRINT COLOR(43,35);" even color does work!"
SHOWPAGE
```

SEE ALSO: PRINT, TEXT

Command: GPS

Syntax:
```
GPS ON
GPS OFF
```

DESCRIPTION:

Switches the GPS (Global positioning System receiver) on or off.
COMMENT: If switched on this usually drains more power from the battery.

SEE ALSO: GPS?, GPS_ALT, GPS_LAT, GPS_LON, SENSOR

Variable: GPS?

Syntax: a=GPS?

DESCRIPTION:

This system variable is 0 (FALSE) if no gps receiver is available on this hardware platform.

SEE ALSO: GPS, GPS_ALT, GPS_LAT, GPS_LON, SENSOR

Variable: GPS_ALT

Syntax: `a=GPS_ALT`

DESCRIPTION:

This system variable returns the measured altitude from the GPS in meters.

SEE ALSO: GPS, GPS_LAT, GPS_LON

Variable: GPS_LAT

Syntax: `a=GPS_LAT`

DESCRIPTION:

This system variable returns the measured latitude from the GPS in degrees.

SEE ALSO: GPS, GPS_ALT, GPS_LON

*

Variable: GPS_LON

Syntax: `a=GPS_LON`

DESCRIPTION:

This system variable returns the measured longitude from the GPS in degrees.

SEE ALSO: GPS, GPS_ALT, GPS_LAT

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

COMMENT: GRAPHMODE does not yet work on all implmentations of X11-Basic. And it has limited effects (usually only on TEXT and GPRINT).

SEE ALSO: `GOLOR\_RGB()`, `TEXT`, `GPRINT`

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
```

7.9. 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: HASH$()

Syntax: `h$=HASH$(t$[,typ%])`

DESCRIPTION:

Executes a hash function on the data contained in t$. Depending on typ% the hash function used is: Typ%=1 ! MD5 2 ! SHA1 3 ! RMD160 5 ! MD2 6 ! TIGER/192 as used by gpg <= 1.3.2. 7 ! HAVAL, 5 pass, 160 bit. 8 ! SHA256 9 ! SHA384 10 ! SHA512 11 ! SHA224 301 ! MD4 302 ! CRC32 303 ! CRC32_RFC1510 304 ! CRC24_RFC2440 305 ! WHIRLPOOL 306 ! TIGER fixed. 307 ! TIGER2 variant.

If typ% is not specified, MD5 is used. HASH$() returns a string which contains the hash value in binary form.

COMMENT: This function is only available if libgcrypt was compiled in.

EXAMPLE:

```
h$=HASH$("Calculate a MD5 sum from this text.",1)
        PRINT LEN(h$)     ! Result: 16
```

SEE ALSO: CRC()

Function: HEX$()

Syntax: `h$=HEX$(d%[,n%])`

DESCRIPTION:

Converts an integer value d% into a string containing its hexadecimal number representation. The optional parameter n% specifies the minimal length of the output. If it is larger than needed, the string will be filled with leading zeros. Negative numbers are converted to unsigned int before processing. If you need binary representations with sign, use RADIX$() instead.

EXAMPLES:

```
PRINT HEX$(123)        Result:  7B
PRINT HEX$(17,8)       Result:  00000011
```

SEE ALSO: STR\$(), BIN\$(), OCT\$(), RADIX\$()

Command: HIDEM

Syntax: HIDEM

DESCRIPTION:

Hide the mouse cursor. (It will be invisible.)

COMMENT: Works only on the framebuffer (Android). On other platforms this command has no effect.

SEE ALSO: SHOWM, DEFMOUSE

Command: HOME

Syntax: `HOME`

DESCRIPTION:

moves text cursor home. (upper left corner)

SEE ALSO: `PRINT AT()`

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.

EXAMPLE:

```
PRINT HYPOT(3,4)    ! Result: 5
```

SEE ALSO: SQRT()

7.10. 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

Function: IMAG()

Syntax: `x=IMAG(z#)`

DESCRIPTION:

Returns the imaginary part of the complex number z#.

EXAMPLE:

```
PRINT IMAG(1-2i)     Result: -2
```

SEE ALSO: CONJ(), REAL()

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

EXAMPLE:

```
        PRINT BIN$((13 IMP 14) AND 15,4)
Result:  1110
```

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

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:

6-bit 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 6-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: INODE()

Syntax: `a%=INODE$(filename$)`

DESCRIPTION:

Returns the inode number associated with a file or directory on disk. Each inode stores the attributes and disk block location(s) of the filesystem object's data.

COMMENT: Works only on Unix-Style file systems.

SEE ALSO: `EXIST()`

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 standard 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 being pressed
PRINT INP%(#1)   ! reads a long from a previously opened file
```

SEE ALSO: OUT, INPUT\$()

Function: `INP?()`

Syntax: `<boolean-result>=INP?(<channel-nr>)`

DESCRIPTION:

Determine the input status of the device. TRUE(-1) is device is ready (chars can be read) otherwise FALSE(0).

SEE ALSO: `INP()`

Command: INPUT

Syntax: INPUT [#<device-number>]<prompt-expression>], <variable> [, ...]|

DESCRIPTION:

INPUT gets comma-delimited input from the standard input or from a previously opened file as specified by <device-number> (use the LINEINPUT function to 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 terminal screen or 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: `a%=INT(b)`

DESCRIPTION:

INT() cuts off the fractional part of the number a. and returns an integer number. The integer number has only 32bit, so a should be in the range of -2147483648 to 2147483647.

EXAMPLE:

```
PRINT INT(1.4), INT(-1.7)
        Result: 1, -1
```

SEE ALSO: CINT(), FRAC(), TRUNC(), ROUND(), FIX()

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.

EXAMPLE:

```
        a()=[3,7,3,0;0,2,-1,1;5,4,3,2;6,6,4,-1]
b()=INV(a())
PRINT DET(a())*DET(b())   ! Result: 1
PRINT DET(a()*b())        ! Result: 1
```

SEE ALSO: SOLVE(), DET()

Function: INVERT()

Syntax: `c&=INVERT(a&,b&)`

DESCRIPTION:

Compute the inverse of a modulo b and return the result. If the inverse exists, the return value is non-zero and c& will satisfy 0 <= c& < b&. If an inverse doesn't exist the return value is zero.

EXAMPLE:

```
PRINT INVERT(12,53)      ! result: 31
```

SEE ALSO: DIV()

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, socket or device.

The second argument is a device-dependent request code. The third argument is either another integer value or a pointer to memory.

An ioctl request has encoded in it whether the argument is an in parameter 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 non-negative value on success. On error, -1 is returned.

COMMENTS: In case of open USB-Devices, following IOCTL requests are implemented: 0 – USB Reset e.g. ret%=IOCTL(#1,0) 1 – get descriptor data structure. The data structure has a length of 4148 bytes. t$=SPACE$(4148) ret%=IOCTL(#1,1,VARPTR(Please see the example program usb.bas for details how to decode the information in this data structure. 2 – Set configuration, e.g. ret%=IOCTL(#1,2,confnr%) 3 – Claim Interface, e.g. ret%=IOCTL(#1,3,intrfnr%) 4 – control_msg, t$=MKL$(a%)+MKL$(b%)+MK ret%=IOCTL(#1,4,VARPTR(t$)) 5 – Set default blk_len, e.g. IOCTL(#1,5,blk_len%) 6 – Set default endpoint_in, e.g. IOCTL(#1,6,ep_in%) 7 – Set default endpoint_out, e.g. IOCTL(#1,7,ep_out%) 12 – get filename+path t$=SPACE$(4100) l%=IOCTL(#1,12,VAR devicefilenr$=LEFT$(t$,l%) 13 – get manufacturer t$=SPACE$(100) l%=IOCTL(#1,13,VARP manufacturer$=LEFT$(t$,l%) 14 – get product name t$=SPACE$(100) l%=IOCTL(#1,14,VAR product$=LEFT$(t$,l%) 15 – get serial number t$=SPACE$(100) l%=IOCTL(#1,15,VARPTR(serialnr$=LEFT$(t$,l%) 16 – get error string t$=SPACE$(100) l%=IOCTL(#1,16,VARPTR(t$) error$=LEFT$(t$,l%)

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
```

7.11. 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 standard 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\$

7.12. K

Command: KEYEVENT

Syntax: `KEYEVENT kc,ks[,t$,k,x,y,xroot,yroot]`

DESCRIPTION:

Waits until a key is pressed (in graphic window). After the key event has occurred, the variables have following content:

kc – Key-code ks – state of Shift/Control/Alt etc. t$ – corresponding 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`

Command: KILL

Syntax: `KILL <filename>`

DESCRIPTION:

KILL deletes a file from the file system.

EXAMPLE:

```
KILL "delme"
```

SEE ALSO: OPEN, RMDIR

7.13. L

Function: LCM()

Syntax: `c&=LCM(a&,b&)`

DESCRIPTION:

Returns the least common multiple of a and b. c is always positive, irrespective of the signs of a and b. c will be zero if either a or b is zero.

EXAMPLE:

```
PRINT LCM(12,18)     ! Result: 36
```

SEE ALSO: GCD()

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.

EXAMPLE:

```
PRINT LEFT$("Hello",1)     ! Result: H
```

SEE ALSO: RIGHT\$(), MID\$()

Function: LEFTOF$()

Syntax: `a$=LEFTOF$(t$,s$)`

DESCRIPTION:

LEFTOF$() returns the left part of t$ at the position of the first occurrence of s$ in t$. If s$ is not contained in t$, the whole string t$ is returned.

EXAMPLE:

```
PRINT LEFTOF$("Hello","ll")   !  Result: He
```

SEE ALSO: `RIGHTOF\$(),MID\$()`

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
```

Function: LGAMMA()

Syntax: `b=LGAMMA(a)`

DESCRIPTION:

The LGAMMA() function returns the natural logarithm of the absolute value of the Gamma function. LGAMMA(x)=ASB(LN(GAMMA(x)))

If x is a NaN, a NaN is returned. If x is 1 or 2, +0 is returned.

If x is positive infinity or negative infinity, positive infinity is returned.

If x is a non-positive integer, a pole error occurs, and the function returns inf.

If the result overflows, a range error occurs, and the function returns inf with the correct mathematical sign.

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

Command: LINE

Syntax: `LINE <x1>,<y1>,<x2>,<y2>`

DESCRIPTION:

Draws a straight line from (x1,y1) to (x2,y2). The line thickness as well as other drawing parameters can be set with DEFLINE and GRAPHMODE.

SEE ALSO: DRAW, PLOT, DEFLINE

Command: LINEINPUT

Syntax: `LINEINPUT [[#]<device-number>,] <string-variable>`

DESCRIPTION:

LINE INPUT reads an entire line from a standard 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`

*

Function: LINEINPUT$()

Syntax: `t$=LINEINPUT$([#1])`

DESCRIPTION:

LINEINPUT$() reads an entire line from a standard input or from a previously opened file as specified by <device-number> (to load a complete file, use BLOAD). Unlike the regular INPUT command, LINEINPUT$() does not stop at delimiters (commas).

SEE ALSO: `INPUT\$()`, `LINEINPUT`

Command: LINK

Syntax: `LINK #<device-nr>,<string-expression:name>`

DESCRIPTION:

LINK links a shared object file/library (*.so in /var/lib) dynamically. It will from now on be addressed via the device-nr.

The addresses 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 0, that the second line is line 1 etc.

EXAMPLE:

```
LIST
LIST 1,10
LIST 5
```

SEE ALSO: PLIST, PRG\$()

Function: LISTSELECT()

Syntax: `num=LISTSELECT(title$,list$())`

DESCRIPTION:

listselect opens a graphical list-selector, which enables the user to select one entry out of an array list$(). The index of the entry is returned or -1 in case no item was selected.

SEE ALSO: FILESELECT

Command: LOAD

Syntax: `LOAD name$`

DESCRIPTION:

Loads a program into memory.

EXAMPLE:

```
LOAD "testme.bas"
```

SEE ALSO: XLOAD, MERGE, CHAIN

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. This command is normally used inside PROCEDUREs and FUNCTIONs. LOCAL does not initialize the variables. If you need them to be initialized, use CLR after LOCAL.

EXAMPLE:

```
LOCAL a,b$,s()
```

Command: LOCATE

Syntax: `LOCATE <row>,<column>`

DESCRIPTION:

Positions the cursor to the specified location. The upper right corner of the screen is located at 0,0.

SEE ALSO: `PRINT AT()`, `CRSLIN`, `CRSCOL`

Function: LOF()

Syntax: `<int-result>=LOF(#n)`

DESCRIPTION:

Returns the length of the file with device number n.

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 integer 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.

EXAMPLE:

```
DO
  PRINT TIME$
  PAUSE 1
LOOP
```

SEE ALSO: `DO`, `EXIT IF`, `BREAK`

Function: LOWER$()

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

DESCRIPTION:

Transforms all upper case letters of a string to lower case. Any non-letter characters are left unchanged.

EXAMPLE:

```
PRINT LOWER$("Oh my GOD!")   ! Result: oh my god!
```

SEE ALSO: UPPER\$()

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()`

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 independent of the system fonts. The font style can be influenced with the DEFLINE and the DEFTEXT command.

SEE ALSO: `DEFTEXT`, `TEXT`, `DEFLINE`, `LTEXTLEN()`

Function: LTEXTLEN()

Syntax: `w=LTEXTLEN(t$)`

DESCRIPTION:

Returns the with of the text t$ in pixels. The font style can be influenced with the DEFLINE and the DEFTEXT command.

SEE ALSO: LTEXT

Function: LUCNUM()

Syntax: `w&=LUCNUM(i%)`

DESCRIPTION:

Returns the i'th Lucas number.

COMMENT: This function works only in the interpreter and only when used in a direct assignment to a big integer variable.

EXAMPLES:

```
w&=LUCNUM(100)  --> Result: 792070839848372253127
```

SEE ALSO: FIB(),PRIMORIAL()

7.14. 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-entries

- 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: MENUKILL

Syntax: MENUKILL

DESCRIPTION:

Erases the menu, which prior has been defined with MENUDEF.

SEE ALSO: MENUDEF

Command: MENUSET

Syntax: `MENUSET n,x`

DESCRIPTION:

Change appearance of menu-entry n with value x.
x=0 ’ ’ normal, reset marker ’^’ x=1 ’^’ set marker x=2 ’=’ set menu-point non selectable x=3 ’ ’ set menu-point selectable x=4 check the menu entry ’-’ permanent non selectable

SEE ALSO: `MENU`

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/mylibrary.bas"
```

Command: MFREE

Syntax: `MFREE adr%`

DESCRIPTION:

Frees a memory area which has been allocated with MALLOC() before. The address must be the same as previously returned by MALLOC().

SEE ALSO: `MALLOC()`

Function: MID$()

Syntax: `m$=MID$(t$,x[,l])`

DESCRIPTION:

Returns l characters in a string from the position x of the string t$. If x is larger than the length of t$, 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()

Command: MKDIR

Syntax: `MKDIR <path-name>[,mode%]`

DESCRIPTION:

MKDIR attempts to create a directory named path-name. The argument mode specifies the permissions to use. It is modified by the process's umask in the usual way: the permissions of the created directory are (mode% AND NOT umask AND (7*64+7*8+7)). Other mode bits of the created directory depend on the operating system.

EXAMPLE:

```
MKDIR "/tmp/myfolder"
```

SEE ALSO: CHDIR, RMDIR

Function: MKI$(), MKL$(), MKS$(

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 reversed 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

Function: MODE()

Syntax: `m%=MODE(filename$)`

DESCRIPTION:

Return the file permissions of the file or directory.
COMMENT: May not work on WINDOWS systems.

EXAMPLES:

```
PRINT OCT$(MODE(".")) ---> Result: 40750
```

SEE ALSO: INODE(), EXIST()

Command: MOTIONEVENT

Syntax: `MOTIONEVENT x,y,xroot,yroot,s`

DESCRIPTION:

Waits until the mouse has been moved. (graphic window). Returns new mouse coordinate (x,y) relative to window, mouse coordinate (xroot,yroot) relative to screen and state of the Alt/Shift/Caps keys (s).

SEE ALSO: MOUSE, MOUSEX, MOUSEY, MOUSEK, MOUSEEVENT

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) and the mouse wheel if present. k=0 no buttons pressed
k=1 left button
k=2 middle button
k=4 right button
k=8 wheel up
k=16 wheel down
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 the mouse coordinate (x,y) relative to the window, the mouse coordinate (xroot,yroot) relative to the screen, the mouse button (k) and state of the Alt/Shift/Caps keys (s). k=0 no buttons pressed
k=1 left button
k=2 middle button
k=3 right button
k=4 wheel up
k=5 wheel down

s=0 normal
s=1 Shift
s=2 CapsLock
s=4 Ctrl
s=8 Alt
s=16 NumLock
s=64 Windows-Key
s=128 AltGr

SEE ALSO: MOUSE, MOUSEX, MOUSEY, MOUSEK, KEYEVENT

Variable: MOUSEX, MOUSEY, MOUSE

Syntax:
```
x%=MOUSEX
y%=MOUSEY
k%=MOUSEK
s%=MOUSES
```

DESCRIPTION:

MOUSEX returns the current horizontal position of the mouse cursor, or of the last position of a touch on a touch screen. MOUSEY holds the vertial position accordingly. MOUSEK returns the current status of the mouse buttons: MOUSEK=0 no buttons pressed
MOUSEK=1 left button
MOUSEK=2 middle button
MOUSEK=4 right button
MOUSEK=8 wheel up
MOUSEK=16 wheel down
or any combinations.

MOUSES returns the current state (state when the touchscreen was last touched) of the SHIFT/CAPSLOCK/CTRL keys. MOUSES=0 no Keys
MOUSES=1 Shift
MOUSES=2 CapsLock
MOUSES=4 Control
MOUSES=8 Alt
MOUSES=16 NumLock
MOUSES=64 Windows-Key
MOUSES=128 AltGr
or any combination.

SEE ALSO: MOUSE, SETMOUSE, MOUSEEVENT

Command: MOVEW

Syntax: `MOVEW n,x,y`

DESCRIPTION:

Moves Window n to absolute screen position x,y

SEE ALSO: OPENW, SIZEW, TITLEW

Function: MSHRINK()

Syntax: `a=MSHRINK(adr%,size%)`

DESCRIPTION:

Reduces the size of a storage area previously allocated with MALLOC. adr% specifies the address of the area, size% gives the required size. Returns 0 if no error. This command does nothing (and always returns 0) on Linux and windows operating systems. It is implemented for compatibility reasons only. If you really need to resize a memory area, use REALLOC().

SEE ALSO: `MALLOC(), REALLOC()`

Command: MSYNC

Syntax: `MSYNC adr,length`

DESCRIPTION:

MSYNC flushes changes made to the in-core copy of a file that was mapped into memory using MAP back to disk. Without use of this call there is no guarantee that changes are written back before UNMAP is called. To be more precise, the part of the file that corresponds to the memory area starting at addr and having length length is updated.

SEE ALSO: `MAP`, `UNMAP`

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 room.

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()`

7.15. N

Operator: NAND

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

DESCRIPTION:

The result of the NAND operator is true, if not both operands are true, and false otherwise. Is the same as NOT (a AND b).

Returns -1 for true, 0 for false.

Also used to combine bits in binary number operations. E.g. (1 NAND -1) returns -2.

EXAMPLES:

```
PRINT 3=3 NAND 4<2  Result:  -1 (true)
PRINT 3>2 NAND 5>3  Result:   0 (false)

PRINT (1 NAND -3)       Result:   -2
```

SEE ALSO: NOR, AND, NOT

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
```

Function: NEXTPRIME()

Syntax: `p&=NEXTPRIME(x&)`

DESCRIPTION:

Returns the smallest prime number bigger than x.

EXAMPLE:

```
PRINT NEXTPRIME(200)      ! Result: 211
PRINT NEXTPRIME(20000000000000000000) ! Result: 20000000000000000089
```

SEE ALSO: `GCD()`

Function: NLINK()

Syntax: `m%=NLINK(filename$)`

DESCRIPTION:

Return the number of (hard) links to the file or directory.
COMMENT: May not work on WINDOWS systems.

EXAMPLES:

```
PRINT NLINK(".") ---> Result: 2
```

SEE ALSO: INODE(), EXIST(), MODE()

Command: NOP, NOOP

Syntax:
```
NOP
NOOP
```

DESCRIPTION:

No Operation: do nothing.

Operator: NOR

Syntax: `<num-expression1> NOR <num-expression2>`

DESCRIPTION:

The result of the NOR operator is true, if both operands are zero, and false otherwise. Used to determine if both of the conditions are false. Is the same as NOT (a OR b).

Returns -1 for true, 0 for false.

Also used to combine bits in binary number operations. E.g. (1 NOR -8) returns 6.

EXAMPLES:

```
PRINT 3=3 NOR 4<2  Result:   0 (false)
PRINT 3>3 NOR 5<3  Result:  -1 (true)

PRINT (4 NOR -128)       Result:   123
```

SEE ALSO:

NAND, OR, NOT, XOR

Command: NOROOTWINDOW

Syntax: NOROOTWINDOW

DESCRIPTION:

Switches back to normal graphic output (normally into a window), if it was switched to ROOTWINDOW before.

SEE ALSO: ROOTWINDOW

Operator: NOT

Syntax: `NOT <num-expression>`

DESCRIPTION:

The result of the NOT operator is true, if the following expression if false, and false if the expression is true. Used to produce the opposite/negation of a logical expression.

Returns -1 for true, 0 for false.

Also used to invert bits in binary number operations. E.g. (NOT -8) returns 7.

SEE ALSO: AND, OR, NAND, NOR

7.16. 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 function 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()

Function: OCT$()

Syntax: `o$=OCT$(d%[,n%])`

DESCRIPTION:

Converts an integer value d% into a string containing its octal number representation. The optional parameter n% specifies the minimal length of the output. If it is larger than needed, the string will be filled with leading zeros. Negative numbers are converted to unsigned int before processing. If you need binary representations with sign, use RADIX$() instead.

EXAMPLES:

```
PRINT OCT$(123)         Result:  173
PRINT OCT$(9,8)         Result:  00000011
```

SEE ALSO: BIN\$(), STR\$(), HEX\$(), RADIX\$()

Function: ODD()

Syntax: `a=ODD(number)`

DESCRIPTION:

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

SEE ALSO: `EVEN()`

Command: ON * GOSUB

Syntax: `ON a GOSUB proc1[,proc2,...]`

DESCRIPTION:

Calls a procedure out of the given list of procedures depending on the value of a. If a=1, the first procedure is used, if a=2 the second, and so on.

SEE ALSO: `GOSUB`

Command: ON * GOTO

Syntax: `ON a GOTO label1[,label2,...]`

DESCRIPTION:

Branches to a label out of the given list depending on the value of a. If a=1, the first label is used, if a=2 the second, and so on.

SEE ALSO: `GOTO`

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. Also one can branch to a label in case of an error. Program excecution can only be continued when RESUME can be used, and when the error is not FATAL.

ON ERROR CONT will ignore any error and will not print error messages.

SEE ALSO: GOSUB, ERROR, RESUME, FATAL

Command: OPEN

Syntax: `OPEN mode$,#<device-number>,filename$[,port%]`

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 correspond to the mode for which the file is opened: "I" – INPUT, "O" –OUTPUT, "A" – APPEND and "U" – UNSPECIFIED/UPDATE/ READandWRITE.

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 UPDATE ("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 datagram socket connection "C" opens a stream 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"

"Y" opens an USB connection. The filename specifies the vendor-ID and product-ID of the device to be opened.

<port-nr> The portnr is used only by the OPEN "UC" and OPEN "UU" statement. It specifies the TCP/IP Port of connection (FTP, WWW, TELNET, MAIL etc.).

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.
OPEN "UY",#1,"0x1c1e:0x0101" --- opens USB device VID=0x1c1e, PID=0x101
                                 for read and write.
```

SEE ALSO: CLOSE, EXIST(), INPUT, LINEINPUT, PRINT, SEEK, LOF(), EOF(), LOC(), BLOAD, LINK, FREEFILE(), CONNECT, IOCTL()

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 is 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`, `USEWINDOW`

Operator: OR

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

DESCRIPTION:

Used to determine if at least ONE OF the conditions is true. If both expression1 AND expression2 are FALSE (zero), the result is FALSE. Returns -1 for true, 0 for false.

Also used to combine bits in binary number operations. E.g. (1 OR 8) returns 9.

EXAMPLES:

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

PRINT (30>20 OR 20<30)  Result:  -1 (true)
PRINT (4 OR 128)        Result:   132
```

SEE ALSO: NAND, AND, NOT, NOR, XOR

*

Function: OR()

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

DESCRIPTION:

Returns <num-expression> OR <num-expression2>

SEE ALSO: `AND()`, `OR`, `AND`

Command: OUT

Syntax: `OUT #n,a`

DESCRIPTION:

Writes a byte a to an open (output) channel or file #n.

SEE ALSO: `PRINT`, `INP()`

7.17. P

Function: PARAM$()

Syntax: `p$=PARAM$(i%)`

DESCRIPTION:

Returns the i'th word from the commandline. Usually parameters are passed this way to a program. The PARAM$(0) usually is the name of the program, which has been excecuted. If there are no more parameter words, an empty string will be returned.

EXAMPLE:

```
i=1
WHILE LEN(param$(i))
  IF LEFT$(param$(i))="-"
    IF param$(i)="--help" OR param$(i)="-h"
      @intro
      @using
    ELSE IF param$(i)="--version"
      @intro
      QUIT
    ELSE IF param$(i)="-o"
      INC i
      IF LEN(param$(i))
        outputfilename$=param$(i)
      ENDIF
    ENDIF
  ELSE
    inputfile$=param$(i)
  ENDIF
  INC i
WEND
```

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 with corners x1,y1 and x2,y2.

SEE ALSO: BOX, RBOX, DEFFILL, COLOR

Variable: PC

Syntax: `i%=PC`

DESCRIPTION:

Returns the Program counter value. This is normally the line number of the line actually processed, or the pointer into bytecode of the code actually processed.

SEE ALSO: `SP`

Command: PCIRCLE

Syntax: `PCIRCLE x%,y%,r%[,a1,a2]`

DESCRIPTION:

Draws a filled circle (or sector) at center coordinates x,y with radius r and optional starting angle a1 and ending angle a2 (in radians).

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)'
  FOR iu=0 TO 15
    PRINT HEX$(PEEK(i+iu) and 255,2)'
  NEXT iu
  PRINT '
  FOR iu=0 TO 15
    a=PEEK(i+iu)
    IF a>31
      PRINT CHR$(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 (or or elliptic sector) at center coordinates x,y with radii a and b.

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

Variable: `PI`

Syntax: `a=PI`

DESCRIPTION:

Returns the value of PI. The value of PI is 3.1415926535... etc.

SEE ALSO: `SIN()`, `COS()`, `EXP()`

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.

EXAMPLE:

```
PIPE #1,#2
PRINT #2,"Hello, get me out of the pipe..."
FLUSH #2
LINEINPUT #1,t$
PRINT t$
```

SEE ALSO: `CLOSE`, `OPEN`, `FORK()`

Command: PLAYSOUND

Syntax: `PLAYSOUND channel,s$`

DESCRIPTION:

PLAYSOUND plays a WAV sample on the sound card. s$ must contain the data from a sound file. (WAV format). The saple is then played once on the channel c. If c is -1 a free channel is selected. There are 16 channels. (currently this only works in the MS WINDOWS or SDL version.)

EXAMPLE:

```
OPEN "I",#1,"sound.wav"
t$=INPUT$(#1,LOF(#1))
CLOSE #1
PLAYSOUND ,t$
```

SEE ALSO: SOUND

Command: PLAYSOUNDFILE

Syntax: `PLAYSOUNDFILE filename$`

DESCRIPTION:

PLAYSOUNDFILE play a soundfile of standard file formats like WAV, OGG, MP3. The sound is played in the background. Any previously played sounds will be stopped if they have not been finished so far. (currently this only works in the ANDROID version of X11-Basic.)

EXAMPLE:

```
PLAYSOUNDFILE "/mnt/sdcard/bas/explosion.ogg"
```

SEE ALSO:

SOUND, PLAYSOUND

Command: PLIST

Syntax: `PLIST [#n]`

DESCRIPTION:

Outputs a formatted listing of the actual program in memory. If an open file channel is given, the listing will be dumped into that file. Also 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)+"m *";
   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`, `DUMP`

Command: PLOT

Syntax: `PLOT x,y`

DESCRIPTION:

Draws a point (single pixel) at screen coordinate x,y

SEE ALSO: `LINE`, `POINT()`, `COLOR`

Function: PNGDECODE$()

Syntax: `bmp$=PNGDECODE$(png$)`

DESCRIPTION:

Converts the content of png$ which should contain data of a Portable Network Graphics format into a bmp data format which then can be pasted to the screen with PUT.

EXAMPLE:

```
OPEN "I",#1,"image.png"
        png$=INPUT$(#1,LOF(#1))
CLOSE #1
bmp$=PNGDECODE$(png$)
        PUT 32,32,bmp$        ! display the image
SHOWPAGE
```

SEE ALSO: PUT, PNGENCODE\$()

Function: PNGENCODE$()

Syntax: `png$=PNGENCODE$(bmp$)`

DESCRIPTION:

Converts bitmap data in bmp$ into the Portable Network Graphics format. The content of png$ can then be saved into a file with ending .png.

EXAMPLE:

```
SGET screen$    ! save the graphics screen content in screen$
png$=PNGENCODE$(screen$)
BSAVE "screen.png",varptr(png$),len(png$)
```

SEE ALSO:

GET, SGET, BSAVE, PNGDECODE\$()

Function: POINT()

Syntax: `c=POINT(x,y)`

DESCRIPTION:

Returns the color of the graphic point x,y in the current window. The color values are of the same format than those used by COLOR and returned by GET_COLOR().

SEE ALSO: PLOT, COLOR

Command: POKE

Syntax: `POKE adr%,byte%`

DESCRIPTION:

POKE writes a byte to address adr% of the program memory.

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

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 will be added to each of these coordinates.

POLYFILL fills the polygon with the color and pattern previously defined by COLOR and DEFFILL.

SEE ALSO: `COLOR`, `DEFFILL`, `POLYLINE`, `POLYMARK`

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 will be added to each of 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: 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 will be added to each of these coordinates.

POLYMARK marks the points with the shape defined by DEFMARK.

SEE ALSO: COLOR, DEFLINE, POLYLINE, POLYFILL

Function: POWM()

Syntax: `c&=POWM(base&,exp&,m&)`

DESCRIPTION:

Return (base raised to exp) modulo m. c=base^exp mod m
A negative exp is supported if an inverse base^-1 mod m exists.

SEE ALSO: INVERT()

Command: PRBOX

Syntax: `PRBOX x1,y1,x2,y2`

DESCRIPTION:

Draws a filled box with rounded corners at x1,y1 and x2,y2.

SEE ALSO: `BOX`, `PBOX`, `DEFFILL`, `COLOR`

Function: PRED()

Syntax: `i=PRED(x)`

DESCRIPTION:

PRED() returns the preceding integer of x. It returns the biggest integer value that is less than x.

EXAMPLE:

```
PRINT PRED(1.2345)   Result: 1
PRINT PRED(0.6)      Result: 0
PRINT PRED(-0.5)     Result: -1
PRINT PRED(0)        Result: -1
```

Function: PRIMORIAL()

Syntax: `w&=PRIMORIAL(i%)`

DESCRIPTION:

Returns the promorial of i%, i.e. the product of all positive prime numbers <=i%.
COMMENT: This function works only in the interpreter and only when used in a direct assignment to a big integer variable. The function is not implemented in some of the X11-Basic versions.

EXAMPLES:

```
w&=LUCNUM(100)  --> Result: 230556796394551842475310214733175607 0
```

SEE ALSO: FIB(), LUCNUM()

Function: PRG$()

Syntax: `t$=PRG$(i%)`

DESCRIPTION:

PRG$() returns the i'th BASIC program line (source code). It will of course only work in the interpreter.

COMMENT: This does not work in compiled programs.

SEE ALSO: TRON, TRACE, TRACE\$, PC

Command: PRINT

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 (?).

The PRINT command has special functions, which modify the appearance of the text and the position of the text on the screen. Namely AT(), TAB(), SPC() and COLOR(). Also a powerful formatting is possible with PRINT USING.

EXAMPLE:

```
PRINT "Hello ",a$," !"
PRINT "Please enter your Name:";
? "A short form..."
```

SEE ALSO: `PRINT AT()`, `PRINT COLOR()`, `PRINT TAB()`, `PRINT SPC()`, `PRINT USING`

*

Command: PRINT AT()

Syntax: `PRINT AT(line,row)[;...]`

DESCRIPTION:

For interactive programs you might want to print output at specific screen locations. PRINT AT(lin,row) will place the text cursor at row row line lin. The top left corner of the screen corresponds to the position (1,1).

EXAMPLE:

```
PRINT AT(4,7);"Test"
PRINT AT(3,1);"   This is a Title   "
```

SEE ALSO: GPRINT, COLS, ROWS, CRSCOL, CRSLIN

*

Command: PRINT TAB() SPC()

Syntax: `PRINT TAB(x)[;...]`
`PRINT SPC(x)[;...]`

DESCRIPTION:

TAB(x) and SPC(x) move the text cursor x positions to the right. TAB starts at the beginning of the line, SPC at current cursor position. So TAB is an absolute and SPC a relative movement.

EXAMPLE:

```
PRINT "Hallo";TAB(30);"Test"
PRINT "Hallo";SPC(30);"Test"
```

SEE ALSO: GPRINT, COLS, ROWS, CRSCOL, CRSLIN

*

Command: PRINT COLOR()

Syntax: `PRINT COLOR(s[,s2,...])`

DESCRIPTION:

Changes the foreground and background text color and also sets the style attributes for the console text.

The COLOR statement s can be of three types depending on their number range. Their meaning is:

Text Mode: Text color: Background 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

You can pass one or more arguments to the COLOR() function to combine the attributes and colors.

EXAMPLE:

```
PRINT COLOR(32,2);"Hallo"
```

SEE ALSO: `COLOR()`

Command: PRINT USING

Syntax: `PRINT a$ USING format$`

DESCRIPTION:

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. For all the details of this command please read the X11-Basic user manual.

EXAMPLE:

```
PRINT 12.34 USING "###.####"
```

SEE ALSO:

`STR\$(),USING\$()`

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. 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.

Procedures are usually defined at the end of the program source code. The program flow may not hit a procedure or function definition. In this case it would produce an error 36 - error in program structure. If you want them in the middle of the "main" part, use GOTOs to jump over them.

EXAMPLE:

```
PRINT "this is the main part of the program
GOTO a
PROCEDURE b
```

```
  PRINT "this is inside the procedure..."
RETURN
a:
PRINT "go on"
GOSUB b
END
```

SEE ALSO: GOSUB, RETURN, LOCAL, FUNCTION

Command: PROGRAM

Syntax: `PROGRAM <title>`

DESCRIPTION:

This statement does nothing. it is ignored. It can be used to specify a title to the program. In future releases this statement may be used to pass some options to the compiler.

SEE ALSO: `REM`

Function: PTST()

Syntax: `c=PTST(x,y)`

DESCRIPTION:

PTST returns the color of the graphic point x,y in the current window. It is the same as POINT().

SEE ALSO: `POINT(), PLOT`

Command: PUT

Syntax: `PUT <x>,<y>,<var$>[,<scale>[,<transparency>[,xs,ys,w,h],angle`

DESCRIPTION:

Maps a graphic bitmap contained in var$ into the graphic window at coordinate x,y. The picture can be scaled by <scale> factor (default: 1). The file or data format used in var$ is a BMP file format. It can contain uncompressed bitmaps optionally with alpha channel.

If you want to paint only a portion of the image you can specify the coordinates, width and height of a rectangular area of the source image (after scaling).

(*not implemented yet: a rotation angle and if transparency is given and the picture has a color table, this is interpreted as a color index, which will be treated as transparent. If an alpha channel is present, this is used as a threshold for the alpha value (0-255). Default is 32. If the coordinates xs,ys,w,h are given, only a rectangular part of the image is mapped.)

If you want to use .png files for your icons to be mapped with PUT, first convert them with PNGDECODE$() to BMP. Also the alpha channel can be preserved. The images also can be included into the sourcecode of the program. See the tool inline.bas on how to make inline data.

COMMENT: You must avoid to put the image or parts of the image outside of the screen. Always make sure, that the image or the specified portion of the image fits on the screen.

EXAMPLE:

```
OPEN "I",#1,"picture.bmp"
t$=INPUT$(#1,LOF(#1))
CLOSE #1
CLEARW
PUT 0,0,t$,2               ! scaled by a factor of 2
PUT 100,100,t$,,,0,32,32,32  ! 32x32 portion of the image is put
SHOWPAGE
```

SEE ALSO: GET, PUT_BITMAP, PNGDECODE\$()

Command: PUTBACK

Syntax: `PUTBACK [#n,]a`

DESCRIPTION:

Puts a character back into an input channel #n.

SEE ALSO: `OUT`

Command: PUT_BITMAP

Syntax: `PUT_BITMAP bitmp$,x,y,w,h`

DESCRIPTION:

Maps a monochrome bitmap contained in bitmp$ into the graphic window at coordinate x,y. The bitmap is stored in raw format, so you must specify the size of the bitmap with w (width) and h (height) in pixels. The Bitmap is drawn with the color set by COLOR and transparency if set with GRAPHMODE. You must avoid to put the bitmap or parts of it outside the screen. The bitmap data format is such that the least significant bit of each byte is drawn to the left. Each line must start on a new byte. So a 9x12 Bitmap stores in 24 Bytes.

SEE ALSO: PUT, COLOR, GRAPHMODE

7.18. 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`

7.19. R

Function: RAD()

Syntax: `r=RAD(x)`

DESCRIPTION:

Converts x from degrees to radians.

EXAMPLE:

```
PRINT RAD(180)      ! Result: 3.14159265359
```

SEE ALSO: `DEG()`

Function: RADIX$()

Syntax: `a$=RADIX$(x%[,base%[,len%]])`

DESCRIPTION:

RADIX$() returns a string containing the representation of the integer number x% in base base%. A minimal length of the string can be specified with len%. If len% is bigger then necessary, preceiding Zeros will be used to fill in. The base% can be between 2 and 62 inclusive. If base% is not specified it defaults to 62. The symbols used are digits from 0 to 9 then capital letters A to Z and then lowercase letters a to z, followed by @ and $.

EXAMPLE:

```
PRINT RADIX$(180,17)      ! Result: AA
```

SEE ALSO: `BIN\$(),OCT\$(),HEX\$()`

Function: RAND()

Syntax: `r%=RAND(0)`

DESCRIPTION:

RAND() returns a pseudo-random integer number between 0 (inclusive) and 2147483647. 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:

```
RANDOMIZE TIMER
FOR i=0 TO 10000
  a=MAX(a,RAND(0))
NEXT i
PRINT a,HEX$(a)
```

SEE ALSO: RANDOM(), RND(), RANDOMIZE, GASDEV

Function: RANDOM()

Syntax:
```
r%=RANDOM(maximum%)
r&=RANDOM(maximum&)
r=RANDOM(maximum)
r#=RANDOM(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(). If the argument is a floating point or complex expression, a random floatingpoint or complex number between 0 (inclusive) and maximum (exclusive) is returned.

EXAMPLE:

```
        PRINT RANDOM(10)    !   Result: 8
PRINT RANDOM(10.1)  !   Result: 8.065922004714
PRINT RANDOM(1+2i)  !   (0.9116473579368+1.596880066952i)
PRINT RANDOM(10000000000000000000) ! Result: 7314076133279565627
```

SEE ALSO: RND(), RANDOMIZE, GASDEV

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

Function: REAL()

Syntax: `x=REAL(z#)`

DESCRIPTION:

Returns the real part of the complex number z#.

EXAMPLE:

```
PRINT REAL(1-2i)     Result: 1
```

SEE ALSO: CONJ(), IMAG()

Function: REALLOC

Syntax: `adr_new%=REALLOC(adr%,newsize%)`

DESCRIPTION:

The realloc() function changes the size of the memory block pointed to by adr% to newsize% bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized. If adr% is 0, then the call is equivalent to MALLOC(newsize%), for all values of newsize; if newsize is equal to zero, and adr% is not 0, then the call is equivalent to FREE(adr%). Unless adr% is 0, it must have been returned by an earlier call to MALLOC(), or REALLOC(). If the area pointed to was moved, a FREE(adr%) is done.

SEE ALSO: `MALLOC(), FREE`

Command: RECEIVE

Syntax: `RECEIVE #n,t$[,a]`

DESCRIPTION:

RECEIVE is used to receive messages t$ from a socket or USB-device #n, which has been opened with OPEN before. If a is given, this variable will take the host address of the sender (IPv4 32 bit format).

SEE ALSO: OPEN, SEND

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, the slowdown is only marginally. A comment after '!' has no influence on the speed of a program at all, so you can use these everywhere.

SEE ALSO: !

Command: RENAME

Syntax: `RENAME oldpathfilename$,newpathfilename$`

DESCRIPTION:

RENAME renames a file, moving it between directories if required. If newpathfilename$ already exists it will be atomically replaced. oldpathfilename$ can specify a directory. In this case, newpathfilename$ must either not exist, or it must specify an empty directory.

EXAMPLE:

```
RENAME "myfile.dat","/tmp/myfile.dat"
```

SEE ALSO: KILL, OPEN

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: `n$=REPLACE$(t$,s$,r$)`

DESCRIPTION:

REPLACE$() returns string-expression where all search strings s$ have been replaced by r$ in t$.

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

COMMENT: **** 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 somewhere 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 run-time. 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:

Return the reverses of a string.

EXAMPLE:

```
print reverse$("Markus Hoffmann")
Result: nnamffoH sukraM
```

SEE ALSO: UPPER\$(), TRIM\$()

Function: RIGHT$()

Syntax: `a$=RIGHT$(t$[,number])`

DESCRIPTION:

RIGHT$() returns the specified number of characters from its string argument, beginning at its right side. If the number of characters is not specified then RIGHT$() returns only the rightmost character.

SEE ALSO: `LEFT\$(),MID\$()`

EXAMPLE:

```
PRINT RIGHT$("Hello",1)
Result: o
```

Function: RIGHTOF$()

Syntax: `a$=RIGHTOF$(t$,s$)`

DESCRIPTION:

RIGHTOF$() returns the right part of t$ at the position of the first occurrence of s$ in t$. If s$ is not contained in t$, an empty string is returned.

SEE ALSO: RIGHTOF\$(),MID\$()

EXAMPLE:

```
PRINT RIGHTOF$("Hello","ll")
Result: o
```

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$.

If the string s2$ is not found in s1$, a 0 is returned. If found, the start position of s2$ in s1$ is returned.

If n is specified, the comparison starts at at position n instead of the end of the string s1$

SEE ALSO: INSTR()

Function: RLD$()

Syntax: `a$=RLD$(a$)`

DESCRIPTION:

Does a run length decoding of string a$. This function reverses the run length encoding function RLE$() 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.

EXAMPLE:

```
PRINT RLD$("1233"+CHR$(8)+"456")
Result: 123333333333456
```

SEE ALSO: `RLE\$()`

Function: RLE$()

Syntax: `a$=RLE$(a$)`

DESCRIPTION:

Does a run length encoding of string a$.

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.

EXAMPLE:

```
PRINT RLE$("123.....................................456")
Result: 123..#456
```

SEE ALSO: `RLD\$()`

Command: RMDIR

Syntax: `RMDIR <path-name>`

DESCRIPTION:

RMDIR deletes a directory, which must be empty.

EXAMPLE:

```
RMDIR "old"
```

SEE ALSO: MKDIR, CHDIR

Function: RND()

Syntax: `r = RND([<dummy>])`

DESCRIPTION:

RND() returns a pseudo-random number between 0 (inclusive) and 1 (exclusive) with a uniform distribution. 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(). The optional dummy parameter is ignored. The granularity of the random values depends on the operating system and is usually only 32 bits.

SEE ALSO: RANDOMIZE, GASDEV(), RANDOM()

EXAMPLE:

```
PRINT RND(1)
Result: 0.3352227557149
```

Function: ROL()

Syntax: `i%=ROL(j%,n%[,b%])`

DESCRIPTION:

Returns the bit pattern in j% rotated left by n% bits. The optional field length b% defaults to 32.

EXAMPLE:

```
PRINT ROL(8,2)     !  Result: 32
PRINT ROL(8,2,4)   !  Result: 2
```

SEE ALSO: SHL(), ROR()

Command: ROOTWINDOW

Syntax: ROOTWINDOW

DESCRIPTION:

Directs all following graphic output to the root window of the screen. (root window = desktop background).

COMMENT: The root window is usually the desktop background. Not in any case is the root window really shown. On linux systems the GNOME desktop always overwrites the root window, so output of X11-Basic is not visible. Use another windowmanager like fvwm2 instead.

SEE ALSO: USEWINDOW

Function: ROOT()

Syntax: `b&=ROOT(a&,n%)`

DESCRIPTION:

Returns the truncated integer part of the nth root of a.

SEE ALSO: `SQRT()`

Function: ROR()

Syntax: `i%=ROR(j%,n%[,b%])`

DESCRIPTION:

Returns the bit pattern in j% rotated right by n% bits. The optional field length b% defaults to 32.

EXAMPLE:

```
PRINT ROR(8,2)     !  Result: 2
PRINT ROR(8,2,3)   !  Result: 8
PRINT ROR(8,4,8)   !  Result: 128
```

SEE ALSO: SHR(), ROL()

Function: ROUND()

Syntax: `b=ROUND(a[,n])`

DESCRIPTION:

Rounds off a value to n fractional digits. If n<0: round to digits in front of the decimal point.

SEE ALSO: `INT()`, `FIX()`, `FLOOR()`, `TRUNC()`

Variable: ROWS

Syntax: `n%=ROWS`

DESCRIPTION:

Returns the number of rows of the text terminal (console).

EXAMPLE:

```
PRINT COLS, ROWS     ! Result: 80      24
```

SEE ALSO: COLS, PRINT AT(), CRSCOL, CRSLIN

Command: RSRC_FREE

Syntax: `RSRC_FREE`

DESCRIPTION:

RSRC_FREE unloads the graphical resources loaded with RSRC_LOAD and frees any memory assigned to it.

SEE ALSO: `RSRC\_LOAD`

Command: RSRC_LOAD

Syntax: `RSRC_LOAD filename$`

DESCRIPTION:

RSRC_LOAD loads a GEM resource file (*.rsc)-File (ATARI ST format) into memory and prepares it to be used.

SEE ALSO: `RSRC\_FREE`, `OBJ\_DRAW()`, `FORM\_DO()`

Command: RUN

Syntax: `RUN`

DESCRIPTION:

starts program execution (RUN)

SEE ALSO: `STOP`, `CONT`, `LOAD`

7.20. S

Command: SAVE

Syntax: `SAVE [a$]`

DESCRIPTION:

SAVE writes the BASIC-program into a file with the name a$. If no filename is specified the program will be saved to the file which was loaded before.

EXAMPLE:

```
SAVE "new.bas"
```

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 image data) on UNIX systems and BMP (device independent bitmap image) else.

EXAMPLE:

```
SAVESCREEN "fullscreen.bmp"
```

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 image data) on UNIX systems and BMP (device independent bitmap image) else.

EXAMPLE:

```
SAVEWINDOW "window.bmp"
```

SEE ALSO:

SAVESCREEN, SGET

Command: SCOPE

Syntax:
```
SCOPE a(),typ,yscale,yoffset
SCOPE y(),x(),typ,yscale,yoffset,xscale,xoffset
```

DESCRIPTION:

SCOPE performs an extended polyline on one or two dimensional data. Drawing and scaling is done very fast. It is possible to plot a million points and lines at once.

The variable typ specifies the type of plot:

0 – draw a polyline 1 – draw points (without lines) 2 – draw impulses

With xscale, yscale, xoffset and yoffset you can specify a scaling function to the data.

EXAMPLE:

```
l=2^10
DIM a(l)
SIZEW ,l,400
CLEARW
FOR i=0 TO l-1
  a(i)=200/100*@si(3*i/512*2*pi)+i/100*sin(20*i/512*2*pi)
NEXT i
COLOR COLOR_RGB(1,0.5,0)
SCOPE a(),1,-10,300  ! Plot the original function
FFT a()              ! Do a Fourier transformation
' Normalize
FOR i=0 TO l-1
  a(i)=a(i)/SQRT(l)
NEXT i
SHOWPAGE
PAUSE 1
FOR i=4 TO 86
  a(i)=0
NEXT i
FOR i=l-1 DOWNTO l-86
```

```
  a(i)=0
NEXT i
FFT a(),-1
COLOR COLOR_RGB(0,1/2,1)
SCOPE a(),0,-10/SQRT(l),300  ! Plot the modified function
SHOWPAGE
END
DEFFN si(x)=x mod pi
```

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, the Android 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 the file.)

SEE ALSO: `RELSEEK`, `LOC()`, `EOF()`, `LOF()`

Command: SELECT

Syntax: `SELECT <expression>`

DESCRIPTION:

Divides a program up into different blocks depending on the result of the expression. Only the integer part of the result of the expression is used to compare with the values given by CASE statements. Program flow branches to the block of code, given by the CASE statement which matches the value of expression. If no CASE block matches, it branches to the DEFAULT block. If no DEFAULT block is given and none of the CASE blocks match, the program resumes after the ENDSELECT. Also after the CASE block is finished, the program resumes after the ENDSELECT. You must not use GOTO out of the SELECT–ENDSELECT block. (although in the interpreter this works, the compiler will not compile it correctly.) You can leave the block any time with BREAK.

COMMENT:

The statement after SELECT will be evaluated/calculated only once, then compared to all the values given by the CASE statements, one by one in the order given. If the first matches, the others will not be evaluated anymore. The DEFAULT section will always match, if no other match before was found. BREAK can be used, but there is no way to CONTINUE with the SELECT after one (CASE) BLOCK was entered.

Code between SELECT and the first CASE or DEFAULT statement will be dead code and should be avoided.

If you need to compare floating point (real) numbers, you must use a IF – ELSE IF – ELSE construct.

EXAMPLE:

```
i=5
SELECT i
  CASE 1
    PRINT 1
  CASE 2,3,4
    PRINT "its 2,3, or 4"
  CASE 5
    PRINT 5
  DEFAULT
    PRINT "default"
ENDSELECT
```

SEE ALSO: CASE, DEFAULT, ENDSELECT, BREAK, IF

Command: SEND

Syntax: `SEND #n,msg$[,adr%,port%]`

DESCRIPTION:

SEND is used to transmit a message via fast UDP datagrams to another socket which may be on another host. Or, send is used to send a data packet to a previously opened USB device. Send with only two parameters may be used only when the socket is in a connected state (see CONNECT), otherwise the destination address and the port has to be specified.

The address of the target is given by adr%, which usually contains a IP4 address (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.

EXAMPLE:

```
port=5555
server$="localhost"   ! if the receiver runs on the same computer
OPEN "UU",#1,"sender",port+1
CONNECT #1,server$,port
i=0
DO
  @sendmessage(i,"The time is: "+date$+" "+time$+" "+str$(i))
  WHILE INP?(#1)
```

```
      t$=@getmessage$()
      IF LEN(t$)
        a=CVI(LEFT$(t$,2))
        PRINT "received: ";a;" ";right$(t$,len(t$)-2)
      ENDIF
    WEND
    INC i
    PAUSE 1
  LOOP
  CLOSE #1
  END
  PROCEDURE sendmessage(id,m$)
    PRINT "sending packet #";id
    SEND #1,mki$(id)+m$
  RETURN
  FUNCTION getmessage$()
    LOCAL t$,adr
    RECEIVE #1,t$,adr
    pid=CVI(MID$(t$,1,2))
    IF pid=0
      @sendACK(pid,adr)
    ENDIF
    RETURN t$
  ENDFUNCTION
  PROCEDURE sendACK(pid,adr)
    @sendmessage(6,CHR$(pid),adr)
  RETURN
```

SEE ALSO: OPEN, CLOSE, CONNECT, RECEIVE, OUT?()

Command: SENSOR

Syntax:
```
SENSOR ON
SENSOR OFF
```

DESCRIPTION:

Switches the sensor phalanx on or off. Sensors can be accelerometer, temperature, pressure, light, humidity, gyroscope, etc... They need to be switched on, before you can use them. If you do not use them anymore, you should switch them off to save battery.

SEE ALSO: GPS, SENSOR?, SENSOR()

Variable: SENSOR?

Syntax: `a=SENSOR?`

DESCRIPTION:

This system variable is 0 if no sensors are available on this hardware platform, otherwise the number of usable sensors is returned. Sensors can be accelerometer, temperature, pressure, light, humidity, gyroscope, etc...

SEE ALSO: `GPS`, `SENSOR`, `SENSOR()`

Function: SENSOR()

Syntax: `a=SENSOR(n)`

DESCRIPTION:

Readout the n-th value of the sensor-phalanx. Usually n=0 –> Temperature 1 –> Ambient light 3 –> Proximity 4,5,6 –> Orientation x,y,z 7,8,9 –> Gyroscope x,y,z 10,11,12 –> Magnetic field x,y,z 13,14,15 –> Accelerometer x,y,z

SEE ALSO: GPS, SENSOR, SENSOR?, ANDROID?

Command: SETENV

Syntax: `SETENV t$,a$`

DESCRIPTION:

Sets the environment variable t$ of the operating system to the value given by a$. The environment variables are not persistent after a reboot or restart of X11-Basic.
(This command is not yet implemented)

EXAMPLE:

```
SETENV "LASTLAUNCHED",DATE$+" "+time$
```

SEE ALSO: ENV\$()

Command: SETFONT

Syntax: `SETFONT t$`

DESCRIPTION:

Loads and sets a font for graphical text commands. t$ may be "SMALL" or "5x7" for a small font "MEDIUM or "8x8 for a medium sized font "BIG" or "8x16" for a big font. "LARGE" or "16x32" for an even bigger font. "HUGE" or "24x48" for an even bigger font. "GIANT" or "32x64" for a gigantic font size.

With UNIX and the X-WINDOW system, the font name may be any valid font name or pattern. On other platforms true-type font filenames can be used. This way, also proportional fonts can be used.

EXAMPLE:

```
SETFONT "BIG"
TEXT 100,100,"Hi, this is a big font"
SETFONT "5x7"
TEXT 100,150,"The small variant"
SETFONT "-*-lucidatypewriter-medium-r-*-*-10-*-*-*-m-*-*-*"
TEXT 100,200,"This may work on a UNIX system."
SETFONT "C:\Arial.ttf"
TEXT 100,200,"This may work on some other platform."
```

SEE ALSO: `TEXT`

Command: SETMOUSE

Syntax: `SETMOUSE x,y[,k[,m]]`

DESCRIPTION:

The SETMOUSE command permits the positioning of the mouse cursor under program control. The optional parameter k can simulate the mouse button being pressed or released. The optional parameter m specifies if the coordinates are relative to the windows origin (m=0, default) or relative to the mouses current position (m=1).

EXAMPLE:

```
ROOTWINDOW
i=0
REPEAT
  SETMOUSE 2,i,,1  ! Move relative by 2 in x and by i in y
  SHOWPAGE
  PAUSE 0.04
  INC i
UNTIL MOUSEY>800
```

SEE ALSO: MOUSE

Command: SGET

Syntax: `SGET screen$`

DESCRIPTION:

SGET stores the content of the graphics window or screen in screen$. The data format is BMP (in case you want to write this into a file).

EXAMPLE:

```
CLEARW           ! clear the screen, otherwise sometimes there is garbage left
FOR i=1 TO 64
  FOR j=1 TO 40
    COLOR COLOR_RGB(i/64,j/40,SQRT(1-(i/64)^2-(j/40)^2))
    CIRCLE i*10,j*10,3    ! draw something
  NEXT j
NEXT i
CIRCLE 100,100,30
SHOWPAGE

SGET screen$
BSAVE "screen.bmp",VARPTR(screen$),LEN(screen$)
```

SEE ALSO: SPUT, SAVEWINDOW, GET, PUT

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()`

Command: SHELL

Syntax: `SHELL file$[,argument$,...]`

DESCRIPTION:

This command executes an executable program which name and path is given in file$. The text console/terminal will be connected to the running program. Optional string arguments can be specified. The difference to SYSTEM (which executes a shell command) is, that with SHELL you execute the file and not a command.

EXAMPLE:

```
SHELL "/usr/bin/sh"  ! starts the sh shell interactively
```

SEE ALSO: SYSTEM

Function: SHL()

Syntax: `i%=SHL(j%,n%)`

DESCRIPTION:

Returns the bit pattern in j% shifted left by n% bits.

EXAMPLE:

```
PRINT SHL(8,2)     Result: 32
```

SEE ALSO: SHR()

Function: SHM_ATTACH()

Syntax: `adr=SHM_ATTACH(id)`

DESCRIPTION:

SHM_ATTACH() attaches the shared memory segment identified by id (see SHM_MALLOC to the programs address space. The address is returned.

You can also attach shared memory segments, which are originally created by another process, but you must know the id, and the process must have read and write permission for the segment.

SEE ALSO: SHM_MALLOC(), SHM_DETACH, SHM_FREE

Command: SHM_DETACH

Syntax: `SHM_DETACH adr`

DESCRIPTION:

SHM_DETACH detaches the shared memory segment located at the address specified by adr from the address space of the program. The to-be-detached segment must be currently attached with adr equal to the value returned by the attaching SHM_ATTACH() call.

SEE ALSO: `SHM\_MALLOC()`, `SHM\_ATTACH()`

Command: SHM_FREE

Syntax: `SHM_FREE id`

DESCRIPTION:

Mark the shared memory segment identified by id to be destroyed. The segment will only actually be destroyed after the last process detaches it. You can only free a shared memory segment, if you are the owner, means, you must have created it with SHM_MALLOC().

SEE ALSO: `SHM\_MALLOC()`

Function: SHM_MALLOC()

Syntax: `id=SHM_MALLOC(size,key)`

DESCRIPTION:

SHM_MALLOC() returns the identifier of the shared memory segment associated with key. A new shared memory segment, with size equal to the value of size rounded up to a multiple of the operating system internal page size, is created if no shared memory segment corresponding to key exists.

Open the shared memory segment - create if necessary.

Return value is the id of the shared memory segment. The id can be used by different processes to attach and access the segment (read and write). In case of an error, -1 is returned.

SEE ALSO: SHM_FREE, SHM_ATTACH()

Command: SHOWM

Syntax: SHOWM

DESCRIPTION:

Show the mouse cursor (make it visible).

SEE ALSO: HIDEM

Command: SHOWPAGE

Syntax: `SHOWPAGE`

DESCRIPTION:

SHOWPAGE refreshes the graphic output. Usually the drawing to the graphic output window or screen is not visible until SHOWPAGE is performed. (Only on TomTom devices this command has no effect, because all graphics drawn is immediately visible).

SEE ALSO: `VSYNC`

Function: SHR()

Syntax: `i%=SHR(j%,n%)`

DESCRIPTION:

Returns the bit pattern in j% shifted right by n% bits.

EXAMPLE:

```
PRINT SHR(8,2)     Result: 2
```

SEE ALSO: SHL(), ROR(), ROL()

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()

Function: SIZE()

Syntax: `l%=SIZE(file$)`

DESCRIPTION:

Returns the size of a file given by its filename (including path).

SEE ALSO: `LOF()`

Command: SIZEW

Syntax: `SIZEW nr,w,h`

DESCRIPTION:

Resizes the graphic window #nr with width w and height h.

SEE ALSO: OPENW, MOVEW

Function: SOLVE()

Syntax: `x()=SOLVE(m(),d())`

DESCRIPTION:

Solves a set of linear equations of the form M()*x()=d(). M() has to be a 2 dimensional array (a matrix) not necessarily a square matrix. d() must be a 1 dimensional array (a vector) with exactly as many elements as lines of M(). x() will be a 1 dimensional array (a vector) with exactly as many elements as rows of M(). Internally a singular value decomposition is used to solve the equation. If the linear equation system does not have an exact solution, the returned vector is the one which minimizes (least square) |M*x-d|.

EXAMPLE:

```
r=3
c=5
DIM a(r,c),b(r)
ARRAYFILL a(),0
a(0,0)=1
a(0,2)=1
a(1,1)=10
a(2,2)=100
b(0)=4
b(1)=2
b(2)=300
PRINT "solve:"
FOR i=0 TO r-1
    PRINT i;": (";
    FOR j=0 TO c-1
      PRINT a(i,j);
      IF j<c-1
        PRINT ", ";
      ENDIF
```

```
    NEXT j
    PRINT ")(x)=(";b(i);")"
NEXT i
PRINT
er()=SOLVE(a(),b())
PRINT "solution:"
FOR i=0 TO c-1
  PRINT "(x";i;")=(";er(i);")"
NEXT i

COMMENT:
This function is only available, if X11-Basic was compiled and linked
together with the LAPACK library. (currently only on linux).
```

SEE ALSO: INV(),DET()

Command: SORT

Syntax:
```
SORT array()[,n%[,idx%()]]
SORT array%()[,n%[,idx%()]]
SORT array$()[,n%[,idx%()]]
```

DESCRIPTION:

SORT sorts the one-dimensional array array(), array%() or array$(). Numeric arrays and string arrays can be sorted. If n% is given, only the first n% values are sorted. If idx%() is given, this (numerical) array will also be sorted corresponding to the first one. This is useful for creating an index table. SORT uses the canonical ASCII coding for sorting strings. If you want a string array sorted by a different alphabet or being sorted case insensitive, you can implement such sorting functions with the index tables.

EXAMPLE:

```
DIM test$(100)
CLR anzdata
DO
  READ a$
  EXIT IF a$="*"
  test$(anzdata)=a$
  INC anzdata
LOOP
SORT test$(),anzdata    ! normal sort according to ASCII
@asort(test$(),anzdata) ! special alphabet sort
@usort(test$(),anzdata) ! sort ignoring the case of characters
PRINT "Result of SORT:"
FOR x=0 TO anzdata-1
  PRINT test$(x)
NEXT x
END
```

```
'
' Sort case insensitive
'
PROCEDURE usort(VAR s$(),anz%)
  LOCAL k%,t$(),i%(),t2$()
  t$()=s$()
  DIM i%(anz%)            ! create index table
  FOR k%=0 TO anz%-1
    i%(k%)=k%
    t$(k%)=UPPER$(t$(k%))
  NEXT k%
  SORT t$(),anz%,i%()     ! Sort with index table
  DIM t2$(DIM?(s$()))
  FOR k%=0  TO anz%-1
    t2$(k%)=s$(i%(k%))
  NEXT k%
  s$()=t2$()
RETURN
'
' Sort with a custom alphabet
'
PROCEDURE asort(VAR s$(),anz%)
  LOCAL k%,alphabet$,j%,t$(),i%(),t2$()
  t$()=s$()
  alphabet$="0123456789AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz"
  DIM i%(anz%)           ! create index table
  FOR k%=0 TO anz%-1
    i%(k%)=k%
    FOR j%=0 TO LEN(t$(k%))-1
      POKE VARPTR(t$(k%))+j%,INSTR(alphabet$,MID$(t$(k%),j%+1))
    NEXT j%
  NEXT k%
  SORT t$(),anz%,i%()     ! Sort with index table
  DIM t2$(DIM?(s$()))
  FOR k%=0  TO anz%-1
    t2$(k%)=s$(i%(k%))
  NEXT k%
  s$()=t2$()
RETURN
```

SEE ALSO: DIM

Command: SOUND

Syntax: `SOUND <channel>,<frequency>[,volume[,duration]]`

DESCRIPTION:

SOUND sets a tone for the sound generator for channel c%. There are 16 sound channels which are mixed together, so c may be between 0 and 15. If c is omitted or -1, a free channel (which is quiet at that time) will be used.

The tone has frequency [Hz], volume [0-1] and a duration [s]. If frequency=0 (or volume=0) the channel will be switched off. If duration is omitted or -1 a permanent sound will be played (infinite duration, until it is cleared by the next SOUND command to that channel).

Each of the 16 channels also support sound samples, which can be set via PLAYSOUND. Volume can be 0 (off) to 1 (maximum). The duration is counted in seconds. The parameters of the sound synthesizer of this channel can be set with WAVE (envelope and wave form). Also noise can be set for a channel.

COMMENT: On systems without ALSA/PCM sound the internal speaker is used. The internal speaker has only one channel (and produces sort of a square wave). The internal speaker is accessed via a console device and needs privileges. Except for on ANDROID devices, the sound currently does not work under UNIX/LINUX.

EXAMPLE:

```
WAVE 1,1,0.05,0.1,0.5,0.1    ! set the instrument parameters
DO
  MOUSEEVENT                 ! wait for mouseclick
  SOUND 1,2*MOUSEX+50,1,0.3  ! play a nice sound
LOOP
```

```
' Also this is possible:
SOUND 1,500    ! SOUND ON
PAUSE 0.1
SOUND 1,0      ! SOUND OFF
```

SEE ALSO: WAVE, PAUSE, PLAYSOUND

Variable: SP

Syntax i%=SP

DESCRIPTION:

The variable SP represents the internal X11-Basic stack pointer. Do not name any other variable SP (or PC) since no value can be assigned to it.

SEE ALSO:

PC

Function: SPACE$()

Syntax: `t$=SPACE$(n)`

DESCRIPTION:

Returns a string containing n spaces.

SEE ALSO: `STRING\$()`

Command: SPAWN

Syntax: `SPAWN procedure`

DESCRIPTION:

Spawns a new thread using the given procedure as an entry point. This entry point can be considered the "main" function of that thread of execution. The new thread will run in parallel to the main thread and can access the same memory (unlike a process which was forked with fork()).

This command is not fully implemented and at the moment messes up the program execution stack of the interpreter since all internal control structures are accessed by two threads. Anyway, in a natively compiled program this can work as expected.

EXAMPLE:

SEE ALSO: `FORK()`

Command: SPEAK

Syntax: `SPEAK t$[,pitch,rate,locale$]`

DESCRIPTION:

Reads text t$ loud. (It uses Text-to-speech synthesis if it is available). You can adjust a factor for pitch (<1 male, >1 female) and rate (<1 slow, >1 fast).

The locale can be: "de" for German pronunciation, "en" for English pronunciation, "us" for English pronunciation, "fr" for French pronunciation, "es" for Spanish pronunciation, "it" for Italian pronunciation.

COMMENT: This command is implemented in the Android version of X11-Basic only. Not all locales might be installed. If a locale is missing, you will be asked to install it.

EXAMPLE:

```
SPEAK "" ! The first SPEAK command initializes the text-to-speech engine
SPEAK "Are you hungry?"
SPEAK "Ich glaube nicht.",1,1,"de"
```

SEE ALSO: PLAYSOUND, WAVE

Command: SPLIT

Syntax: `SPLIT 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 search in parts of t$ which are in brackets.

Quoted parts of the string are not split up.

EXAMPLE:

```
SPLIT "Hello, this is a string."," ",0,a$,b$
```

SEE ALSO: `WORT\_SEP`, `WORD\$()`

Command: SPUT

Syntax: `SPUT screen$`

DESCRIPTION:

Map a screen bitmap, which was stored in screen$ (and saved with SGET) back to the screen.

EXAMPLE:

```
CLEARW           ! clear the screen, otherwise sometimes there is garbage left
FOR i=1 TO 64
  FOR j=1 TO 40
    COLOR COLOR_RGB(i/64,j/40,SQRT(1-(i/64)^2-(j/40)^2))
    CIRCLE i*10,j*10,3    ! draw something
  NEXT j
NEXT i
CIRCLE 100,100,30
SHOWPAGE

SGET b$     ! get the whole screen bitmap and save it in b$
CLEARW
SHOWPAGE    ! screen is now blank
PRINT "now reput the screen"
PAUSE 1
SPUT b$     ! put back the saved screen content
SHOWPAGE
```

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. The function can also be used on complex numbers, then returning a complex result. You can always force the function return the complex sqare root (given a real argument) by using: SQRT(a+0i).

EXAMPLES:

```
PRINT SQR(25)      ! Result: 5
PRINT SQRT(-1+0i)  ! Result: (0+1i)

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:

STIMER returns the integer part of TIMER. So the resolution is 1 second and the value fits in 32 bit integers. (And it is a bit faster than 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

Function: STR$()

Syntax: `t$=STR$(a[,b,c])`

DESCRIPTION:

STR$() converts a number into a string of length b with c significant digits. If b or c are omitted, the string will contain as much digits as the number requires.

SEE ALSO: `VAL(), PRINT USING`

Function: STRING$()

Syntax: `a$=STRING$(i%,b$)`

DESCRIPTION:

The STRING$() function returns a string consisting of i% copies of b$.

SEE ALSO: `SPACE\$()`

Command: SUB

Syntax:

```
SUB a,b
SUB a%,b%
SUB a#,b#
SUB a&,b&
```

DESCRIPTION:

Decrease the value of the variable a by the result of b.

EXAMPLE:

```
a=0.5
SUB a,5
Result: -4.5
```

SEE ALSO: ADD, MUL, DIV

*

Function: SUB ( )

Syntax:
```
c=SUB(a,b)
c%=SUB(a%,b%)
c#=SUB(a#,b#)
c&=SUB(a&,b&)
```

DESCRIPTION:

Returns the result of a minus b.

SEE ALSO: SUB, ADD()

Function: SUCC()

Syntax: `a=SUCC(b)`

DESCRIPTION:

Determines the next higher integer number.

SEE ALSO: `PRED()`

Command: SWAP

Syntax:

```
SWAP a,b
SWAP a%,b%
SWAP a$,b$
SWAP a(),b()
```

DESCRIPTION:

Exchanges the values of the variables a and b. A and b can be of any type, but the types of a and b must be equal. SWAP a(0),b would also be possible.

EXAMPLE:

```
a=4
b=5
SWAP a,b
print a      ! Result: 5
```

SEE ALSO: LET, Operator: =

*

Function: SWAP ()

Syntax: `a%=SWAP(b%)`

DESCRIPTION:

Swaps High and Low words of b and returns the result. b is always treated as a 32 bit unsigned integer.

EXAMPLE:

```
PRINT HEX$(SWAP(5))    ! Result: 0000000050000
```

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

Function: SYM_ADR()

Syntax: `adr=SYM_ADR(#n,sym_name$)`

DESCRIPTION:

SYM_ADR() resolves the address 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 the shell command. SYSTEM provides a way to use alle commands like rm, rmdir, mkdir etc. which are not implemented in X11-Basic, but which are available from a command shell. (This is usually sh or busybox on UNIX/LINUX/ANDROID and TomTom systems and DOS on MS WINDOWS).

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

7.21. T

Function: TALLY()

Syntax: `a%=TALLY(t$,s$[,start%])`

DESCRIPTION:

TALLY() returns the number of occurrences of s$ in t$, starting at the given position start% in t$. If s$ is not present in t$, zero is returned.

EXAMPLE:

```
PRINT TALLY("Hello","l")    ! Result: 2
```

SEE ALSO: INSTR()

Function: TAN()

Syntax: `b=TAN(a)`

DESCRIPTION:

Returns the tangens of the expression in radians.

SEE ALSO: SIN(), ATAN(), TANH()

Function: TANH()

Syntax: `b=TANH(a)`

DESCRIPTION:

Returns the tangens hyperbolicus of the expression in radians.

SEE ALSO: SIN(), ATANH(), TAN()

Variable: TERMINALNAME$

Syntax: `a$=TERMINALNAME$`

DESCRIPTION:

Returns the device name of the terminal connected to the stdout standard output (if a terminal device is connected).

EXAMPLE:

```
PRINT TERMINALNAME$
Result: /dev/pts/0
```

*

Function: TERMINALNAME$()

Syntax: `t$=TERMINALNAME$(#n)`

DESCRIPTION:

Returns the device name of the terminal connected to file #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, DEFTEXT

Variable: TIME$

Syntax: `a$=TIME$`

DESCRIPTION:

Returns the system time as a string. Format: hh:mm:ss and is updated every second.

EXAMPLE:

```
PRINT TIME$,DATE$   ! 14:49:44       11.03.2014
```

SEE ALSO: DATE\$, TIMER, UNIXTIME\$()

Variable: TIMER

Syntax: `TIMER`

DESCRIPTION:

Returns actual time in number of seconds since 01.01.1970 00:00 GMT. The value has milliseconds resolution. TIMER is often used to measure times.

EXAMPLE:

```
n%=1000000
DIM t$(n%),u%(n%)
t=TIMER
PRINT "filling the string array with ";n%;" strings ..."
FOR i%=0 TO n%-1
  t$(i%)=STR$(RANDOM(n%))
  u%(i%)=i%
NEXT i%
PRINT "this took ";TIMER-t;" seconds."
END
```

SEE ALSO: `STIMER`, `CTIMER`, `TIME\$`, `DATE\$`, `UNIXTIME\$()`, `UNIXDATE\$()`

Command: TITLEW

Syntax: `TITLEW n,title$`

DESCRIPTION:

Gives the window #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 front of the screen.

SEE ALSO: BOTTOMW, MOVEW

Command: TOUCH

Syntax: `TOUCH #n`

DESCRIPTION:

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

EXAMPLE:

```
OPEN "U",#1,"test.txt"
   TOUCH #1
   CLOSE #1
```

SEE ALSO: OPEN, CLOSE

Variable: TRACE$

Syntax: `a$=TRACE$`

DESCRIPTION:

The variable TRACE$ contains the command which is next to be processed.

EXAMPLE:

```
PRINT TRACE$      ! Result: END
END
```

SEE ALSO: TRON, TROFF, PC

Function: TRIM$()

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

DESCRIPTION:

TRIM$(a$) returns a modified string taken a$
1. replace Tabs by space 2. replace double spaces by single ones 3. remove leading and trailing spaces 4. Parts of the string which are in quotes ("") will not be changed

SEE ALSO: XTRIM\$(), REPLACE\$()

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`

Variable: TRUE

Syntax: `TRUE`

DESCRIPTION:

Constant -1. This is simply another way of expressing the value of a condition when it is true and is equal to -1 (all bits not zero).

SEE ALSO: `FALSE`

Function: TRUNC()

Syntax: `a=TRUNC(x)`

DESCRIPTION:

TRUNC() rounds x to the nearest integer not larger in absolute value. TRUNC complements FRAC: TRUNC(x)=x-FRAC(x)

SEE ALSO: `FRAC(),FLOOR(),FIX()`

Function: TYP?()

Syntax: `a%=TYP?(<var>)`

DESCRIPTION:

Returns the type of a variable. 0 – invalid 1 – 32 bit integer 2 – 64 bit floating point 3 – big integer 4 – ARBFLOATTYP 5 – complex 6 – ARBCOMPLEXTYP 7 – String +8 – Array of typ 0-7 +32 – it is a constant of typ 0-7

COMMENT: This function is nearly useless. It only shows internals of X11-Basic.

EXAMPLES:

```
PRINT typ?(a)  -> 2
PRINT typ?(a$) -> 7
PRINT typ?(a())-> 10
```

7.22. U

Function: UBOUND()

Syntax: `n%=UBOUND(array()[,i%])`

DESCRIPTION:

The UBOUND function returns the largest subscript for the indicated dimension of an array plus one. This is the size of the i'th dimension. i% specifies which dimension's upper bound to return. 0 = first dimension, 1 = second dimension, and so on. Default is 0.

SEE ALSO: `DIM`, `DIM?()`, `ARRPTR()`

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:

Un-compresses the content of a string which has been compressed with the COMPRESS$() function before.

SEE ALSO: `COMPRESS\$()`

Variable: UNIX?

Syntax: `<boolean-result>=UNIX?`

DESCRIPTION:

Returns TRUE (-1) If the program is running under a UNIX environment.

SEE ALSO: `WIN32?`, `ANDROID?`

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. Time and Date returned are local times adjusted to summer and winter time and based on CET.

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:

Un-links a shared object which has been linked before and occupies file channel #n.

SEE ALSO: `LINK`, `CLOSE`

Command: UNMAP

Syntax: `UNMAP adr%,len%`

DESCRIPTION:

Un-map files or devices out of memory.

The UNMAP command deletes the mappings for the specified address range. Further PEEK() and POKEs to addresses within the old range will produce an error (crash). The region is also automatically unmapped when X11-Basic is terminated. On the other hand, closing the file does not un-map the region.

SEE ALSO: `MAP`

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\$()`

Command: USEWINDOW

Syntax: `USEWINDOW #n`

DESCRIPTION:

Use the window n for all following graphic commands.

SEE ALSO: OPENW, ROOTWINDOW

Function: USING$()

Syntax: `<string-result>=USING$(a,format$)`

DESCRIPTION:

The function USING$() returns a formatted string made out of a numeric value a. How the number is formatted can be set by a format$.

# Denotes a numerical digit (leading spaces).
0 Denotes a numerical digit (leading zeros).
* Denotes a numerical digit (leading asterisks).
$ Denotes a numerical digit (single leading Dollar).
? Denotes a numerical digit (single leading EURO).
^^^^ After # digits prints numerical value in exponential e+xx format. Use ^^^^^ for E+xxx values. The exponent is adjusted with significant digits left-justified.
. Period sets a number's decimal point position. Digits following determine rounded value accuracy.
+ Plus sign denotes the position of the number's sign. + or - will be displayed.
- Minus sign (dash) placed before or after the number, displays only a negative value's sign.
_ Underscore preceding a format symbol prints those symbols as literal string characters.

Note: Any string character not listed above will be printed as a literal text character (useful to add commas or units). If the number cannot be expressed with the given format, a series of "*" will be displayed. The returned string will always have exactly the same length than format$. USING$() rounds to the nearest printed digit.

EXAMPLE:

```
PRINT USING$(1.23456,"+##.###^^^^")   ! Result: + 1.235e+00
```

SEE ALSO: PRINT USING, STR\$()

7.23. V

Function: VAL()

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

DESCRIPTION:

VAL() converts String representing a floating point number into a numeric value. If the string does not represent a valid number 0 is returned.

SEE ALSO: `VAL?(), STR\$()`

EXAMPLE:

```
a=VAL("3.1415926")
```

*

Function: VAL?()

Syntax: `a=val?(t$)`

DESCRIPTION:

Returns the number of characters from a string which can be converted to a number.

EXAMPLE:

```
print val?("12345.67e12Hallo")  Result: 11
```

SEE ALSO: `VAL()`

Operator: VAR

Syntax:
```
PROCEDURE name(...,VAR a,...)
FUNCTION name(...,VAR z,...)
```

DESCRIPTION:

This operator can declare a variable in a parameter list of a procedure or a function to be passed by reference instead of by value. This is useful to pass (more than one) return values.

EXAMPLE:

```
@sum(13,12,a)
        @sum(7,9,b)
        PRINT a,b
        '
        PROCEDURE sum(x,y,VAR z)
          z=x+y
        RETURN
```

SEE ALSO: PROCEDURE, FUNCTION

Function: VARIAT()

Syntax: `a%=VARIAT(n%,k%)`

DESCRIPTION:

Returns the number of permutations of n elements to the k-th order without repetition.

EXAMPLE:

```
print VARIAT(6,2)  Result: 30
```

SEE ALSO: COMBIN(),FACT()

Function: VARPTR()

Syntax: `<adr>=VARPTR(<variable>)`

DESCRIPTION:

Determines the address of a variable and returns a pointer. Usually this is used together with PEEK() and POKE to modify the content of the variable. VARPTR() can also be used to determine the address of an array index.

EXAMPLE:

```
PRINT VARPTR(t$),VARPTR(a(2,4))
POKE VARPTR(t$),ASC("A")
```

SEE ALSO: ARRPTR(), PEEK(), POKE

Command: VERSION

Syntax: `VERSION`

DESCRIPTION:

Shows X11-Basic version number and date.

EXAMPLE:

```
VERSION
Result: X11-BASIC Version: 1.08 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, e.g. 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 synonym for SHOWPAGE. Graphic output will not be shown in the window until SHOWPAGE (or VSYNC). (This command has no effect on the TomTom Device.)

SEE ALSO: `SHOWPAGE`

7.24. W

Command: WATCH

Syntax: `WATCH filename$`

DESCRIPTION:

WATCH can be used to monitor individual files, or to monitor directories. When a directory is monitored, FILEEVENT$ will return events for the directory itself, and for files inside the directory. Note that WATCH is not available on every operating system.

EXAMPLE:

```
WATCH "/tmp"
DO
  a$=FILEEVENT$
  IF LEN(a$)
    PRINT a$
  ENDIF
LOOP
```

SEE ALSO: FILEEVENT\$

Command: WAVE

Syntax: `WAVE c%,form%[,attack,decay,sustain,release]`

DESCRIPTION:

WAVE controls the internal sound synthesizer. You can specify a waveform generator and an envelope for each of the 16 sound channels.

Set the given parameters for channel c%. There are 16 sound channels which are mixed together, so c% may be between 0 and 15.

If c% is omitted or -1, the parameters are set for all channels.

form% specifies the tone generator for the specified channel:

Tone Generators: 0 - silence (default for channels 1-15) 1 - sin wave (default for channel 0) 2 - square wave 3 - triangular wave 4 - sawtooth wave 5 - white noise

The envelope of the tones are specified using 4 parameters attack,decay,sustain and release. attack, decay and release values are specified in seconds. sustain level values are between 0 and 1.

* "Attack time" is the time taken for initial run-up of level from nil to peak, beginning when the SOUND command is executed. * "Decay time" is the time taken for the subsequent run down from the attack level to the designated sustain level, after the attack part of the envelope is over. * "Sustain level" is the level during the main sequence of the sound's duration, until duration time is reached (e.g. 0.8). * "Release time" is the time taken for the level to decay from the sustain level to zero after the duration time is over.

If you want a permanent tone, set attack to 0, sustain to 1 and decay as well as release to any value.

The WAVE commands allow to simulate real instruments, e.g. strings, trumpet or piano. A realistic sound can only be achieved by also using higher harmonics. To simulate this, you will have to use more than one channel and play them simultaneously.

Volume, frequency and duration for the specified sound channel are set by the SOUND command.

SEE ALSO: SOUND, PLAYSOUND

EXAMPLE:

```
WAVE 1,1,0,,1     ! set sine wave, no attack
SOUND 1,500,1     ! play a permanent tone on channel 1
```

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
```

Variable: WIN32?

Syntax: `<boolean-result>=WIN32?`

DESCRIPTION:

Returns TRUE (-1) If the program is running under MS WINDOWS (32 bit).

EXAMPLE:

```
IF WIN32?
  a$=FSFIRST$("C:\","*.dat")
ELSE
  a$=FSFIRST$("/tmp","*.dat")
ENDIF
```

SEE ALSO: UNIX?, TT?, ANDROID?

Function: WORD()

Syntax: `a=WORD(b)`

DESCRIPTION:

Returns lower 16 bits of b and expands sign. B is always treated as an integer.

SEE ALSO: BYTE(), CARD(), SWAP()

Function: WORD$()

Syntax: `a$=WORD$(b$,n[,delimiter$])`

DESCRIPTION:

Returns the n'th word of b$. Words are separated by space or by the first character of delimiter$.

EXAMPLE:

```
a$=WORD$("Hello, this is a string.",3)
b$=WORD$("Hello, this is a string.",2,",")
```

SEE ALSO: SPLIT, WORT_SEP()

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 search parts of t$ which are in brackets.

Quoted parts of the string are not split up.

EXAMPLE:

```
WORT_SEP "Hello, this is a string."," ",0,a$,b$

COMMENT:
This command should not be used anymore. Please use SPLIT instead.
```

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 search parts of t$ which are in brackets.

Quoted parts of the string are not split 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

7.25. X

Command: XLOAD

Syntax: XLOAD

DESCRIPTION:

Opens a fileselector where the user can select a basic source file which then will be loaded into memory.

SEE ALSO: XRUN, LOAD, FILESELECT

Operator: XOR

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

DESCRIPTION:

Logical exclusive OR operator. XOR returns FALSE (0) if both arguments have the same logical value. The operator also works on each bit.

Table: ——————————— A | B | A XOR B —-+——+————— -1 | -1 | 0 -1 | 0 | -1 0 | -1 | -1 0 | 0 | 0 ———————————

EXAMPLE:

```
PRINT 3=3 XOR 4>2  Result:   0 (false)
      PRINT 3>3 XOR 5>3  Result:  -1 (true)
PRINT (4 XOR 255)        Result:   251
```

SEE ALSO: NAND, OR, NOT, AND

*

Function: XOR ( )

Syntax: `c%=XOR(a%,b%)`

DESCRIPTION:

XOR(a,b) returns the bit-wise exclusive or of the two arguments.

EXAMPLE:

```
PRINT XOR(7,5)     ! Result: 2
```

SEE ALSO: `OR()`, `AND`, `XOR`

Command: XRUN

Syntax: `XRUN`

DESCRIPTION:

Opens a fileselector where the user can select a basic source file which then will be loaded into memory and executed.

SEE ALSO: `XLOAD`, `RUN`, `FILESELECT`

Function: XTRIM$()

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

DESCRIPTION:

XTRIM$(a$) returns a$ with following modifications:
1. replace Tab's (CHR$(9)) by space, 2. replace double spaces by single ones, 3. remove leading and trailing spaces, 4. parts of the string which are in quotes ("") will not be changed, 5. convert all parts of the string, which are outside quotes ("") to upper case.

SEE ALSO: TRIM\$(), REPLACE\$(), UPPER\$()

8 Frequently asked Questions

How easy is it to hack into my programs?

Well, first of all: it is possible. The basic source files (.bas) are of course readable by any text editor and such modifiable. The bytecode compiled code (.b) is already harder to read and nearly impossible to convert back into source code. However, since X11-basic is open source, everybody who wants to can look into the sourcecode and can read all information necessary to decode the bytecode and also modify it. It's possible but a real big job to do. On the level of bytecode translated to C source also here someone could modify it. Once the bytecode is compiled into real machine language, the code is as safe from hackers as any other code is (means that there is nearly no way back).

Even if you incorporate the bytecode into the virtual machine, your program should be safe from snoopers, they might not even know your program is byte-code generated. You can also instruct the bytecode compiler not to attach any symbol table or extra debugging information.

Do I need a license to distribute my programs?

No. You don't need a license to use X11-Basic (it's free), and you definitely don't need any license to distribute or sell your programs. The only agreement you have to worry about is that if you choose to use X11-Basic, you assume any and all consequences, direct or indirectly from the use of X11-Basic. Which means: don't blame me if it doesn't work as you think it should. X11-Basic can be used for any task, whether it's profit-seeking or otherwise. I do not want to know, and you don't pay me a cent. You don't even have to acknowledge that your program was created with X11-Basic (although this would be a nice gesture). You're allowed to bundle X11-Basic along with your program(s), as long as the user is well informed that it's not buying into X11-Basic, but rather, buying into your program. How is that done? By not even advertising that your distribution includes a copy of X11-Basic. However, if you want to distribute or modify X11-Basic itself, or if you want to incorporate parts of the X11-Basic sourcecode, you will need to follow the GNU public license.

Recently asked question

Q: I downloaded the last update to x11-basic but I have a problem with the UTF-8 character set... I cannot use no more the ascii set the graphic part of it ...I made a small game that use them now it is not working no more...there's a way to fix this problem?

A: Yes, there is. All characters are still there, but you cannot access them with a simple CHR$(). One method is to copy the characters from a unicode table like this: [http://de.wikipedia.org/wiki/Unicodeblock_Rahmenzeichnung Frames] with the mouse into the editor.
You need to use a UTF-8 capable editor, e.g. pico, nano, gedit.
If this is not working for you, alternatively you can code the character yourself by the unicode number:

```
FUNCTION utf8$(unicode%)
  IF unicode%<0x80
    RETURN CHR$(unicode%)
  ELSE IF unicode%<0x800
    RETURN CHR$(0xc0+(unicode%/64 AND 0x1f))+CHR$(0x80+(unicode% AND 0x3f))
  ELSE
    RETURN CHR$(0xe0+(unicode%/64/64 AND 0xf))+CHR$(0x80+(unicode%/64 AND 0x3f))+CHR$(0x80+(unicode%
  ENDIF
ENDFUNCTION
```

So e.g. the charackter 0x250C can be coded with `@utf8$(0x250C)`.

Q: Is there a GUI-Designer for the graphical user unterface functions of X11Basic ?

A: Well, so far nobody has made a real efford to write a real graphical GUI_designer. But the program gui2bas may help creating GUI forms. The input is a very siple ASCII-File (*.gui) which defines the interfa
So far many GEM object types are supportet (and even Atart ST *.rsc-fi may be converted to *.gui files with the rsc2gui program.) but support listboxes, popup-menues and Tooltips may be included in future.

Q: My old ANSI Basic Programs (with line-Numbers) produce lots of errors the interpreter. How can I run classic (ANSI) Basic programs?

A: Classic Basic programs have to be converted before they can be run wit

X11-Basic. With the bas2x11basic converter program most of this conver
will be done automatically.

9 COMPATIBILITY

9.1. 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 tries to be compatible and 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 from which the value came from. 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 GOSUB statement). The same rules apply to FUNCTION: VAR in the parameter list of a function call allows a FUNCTION to get a variable parameter "by-reference". In contrast, traditional BASIC interpreters always pass variables in parameter lists "by-reference". The problem with "by-reference" parameters is that you must be fully aware of what happens inside the subroutine: assignments 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 l(eft)value

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.

INT() function

In X11-Basic INT() gives probably different results for negative numbers and for numbers bigger than 2147483648. INT() is internally implemented as "cast to int" (has ever be like this, very fast, and also the compiler relies on this). This means, that the argument must not contain numbers which cannot be converted to 32bit integers. And also the fractional part of the floating point numbers is cut off (like TRUNC()) instead of rounded down (like on most other BASIC dialects). If you rely on a correct behaviour for negative numbers and for big numbers you probably want to use FLOOR() instead.

DIM Statement

In X11-Basic the DIM statement probably behaves different compared to other dialects of BASIC. DIM in X11-Basic will reserve space in memory for exacly the number of indexes specified. Other BASIC dialect do reserve one more than specified. If you are surprised getting "Field index out of range" errors, this probably comes from accessing a field index which is not there. For example: DIM a(5) will reserve memory for exactly 5 values: a(0), a(1), a(2), a(3), and a(4). a(5) does not exist and therefor you will get an error if you try to access it. The way X11-Basic

implemented it is more logical and similar to C and JAVA. But if you are used to thinking that an array starts with the first index 1 (instead of 0) you will probably be a little confused.

LET Statement

Al though it is implemented into X11-Basic, there is no benefit in using the LET statement. On contrary, using LET makes your program slower than necessary. Just leave it out, do not use it. Assignments can be made without the LET statement.

TOS/GEM implementation

Because Gfa-Basic on ATARI-ST makes much use of the built in GUI functions of the ATARI ST, which are not available on other operating systems, X11-Basic can only have limited compatibility. GEM style (and compatible) ALERT boxes, menus and object trees are supported by X11-Basic 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 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$()`).

Incompatible data types

X11-Basic uses the default datatype (without suffix) and the integer data type (Suffix %). This is compatible with most of the BASIC dialects. However the complex data type (suffix #) is not supported by most of the BASIC dialects and the suffix # is sometimes optionally used by regular float variables (like in GFA-Basic).

The suffix & which is used for big integer variables could be confused with the short int data type of X11-Basic which also uses this suffix. However, in general

these programs will run and give correct results. Using the infinite precision routines is just slower.

Short int and byte data types will not be used by X11-Basic. There used be useful only on computers with short memory do save some RAM. These times have passed, so that the statndard integer data type (with the suffix %) will do.

The suffix | will be reserved for future use, most likely for multiple precision floating point variables.

9.2. 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"
DRAW	Draws points and lines. –> PLOT, LINE
FILES	Lists the files on a disk. –> SYSTEM "ls -l"
FRE()	Returns the amount of memory free (in bytes). –> see below
MAT CLR	clears a matrix/makes a zero matrix –> ARRAYFILL, CLR, 0()
MAT DET	calculates the determinat of a matrix –> DET()
MAT INV	calculates the inverse of a matrix –> INV()
MAT ONE	creates a unit matrix –> 1()
MAT TRANS	calculates the transverse of a matrix –> TRANS()
MSHRINK()	Reduces the size of a storage area –> REALLOC()
NAME AS	Renames an existing file. –> RENAME
QSORT	Sorts the elements of an array. –> SORT
RC_COPY	Copies rectangular screen sections (–> COPYAREA)
RESERVE	Increases or decreases the memory used by basic (obsolete)
RND as a sysvar	see RND()
ROL&(), ROL%()	Rotates a bit pattern left. –> ROL()

ROR&(), ROR%()	Rotates a bit pattern right. –> ROR()
SHEL_FIND()	⟶ SYSTEM "find ..."
SHL&(), SHL%()	Shifts a bit pattern left –> SHL()
SHR&(), SHR%()	Shifts a bit pattern left –> SHR()
SYSTEM	obsolete ⟶ QUIT
SHEL_ENVRN()	⟶ ENV$()
SHEL_READ	obsolete ⟶ PARAM$()
SSORT	Sorts using the Shell-Metzner method. –> SORT
THEN	keyword in If statements (obsolete)

For some GFA-Basic commands you can construct 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 a,t$,a$,unit$,s$
  IF n=0
    s$="MemFree:"
  ELSE
    s$="SwapFree:"
  ENDIF
  a=FREEFILE()
  OPEN "I",#a,"/proc/meminfo"
  WHILE NOT EOF(#a)
    LINEINPUT #a,t$
    EXIT IF word$(t$,1)=s$
  WEND
  CLOSE #a
  t$=TRIM$(t$)
  a$=word$(t$,2)
  unit$=word$(t$,3)
  IF unit$="kB"
    RETURN VAL(a$)*1024
  ELSE
    RETURN VAL(a$)
  ENDIF
ENDFUNCTION
```

or

```
DEFFN ob_x(adr%,idx%)=DPEEK(adr%+24*idx%+16)
DEFFN ob_y(adr%,idx%)=DPEEK(adr%+24*idx%+18)
DEFFN ob_w(adr%,idx%)=DPEEK(adr%+24*idx%+20)
DEFFN ob_h(adr%,idx%)=DPEEK(adr%+24*idx%+22)
```

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 routines
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 routine
GINTIN, GINTOUT	Address of the AES Integer input/output block.
HIMEM	address of the area of memory which is not allocated by interpr
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 or SDL platforms:

CHDRIVE	Sets the default disk drive –> CHDIR

DMACONTROL, DMASOUND	Controls the DMA sound (see PLAYSOUND)
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
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 variable type
DFREE()	free space on a disc
GETSIZE()	return the number of Bytes required by a screen area
HARDCOPY	Prints the screen –> save screen
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.
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-Window-Function

These GFA-Basic commands may be supported in a later version of X11-Basic:

APPL_FIND(fname$)	Returns the ID of the sought after application.
BYTE{x}	read the contents of the address x
C:	Calls a C or assembler program with parameters as in C
CARD{x}	Reads/writes a 2-byte unsigned integer
CHAR{x}	Reads a string of bytes until a null byte is encountered
DEFLIST x	Defines the program listing format.
DEFNUM n	Affects output of numbers by the PRINT command
DELETE x(i)	Removes the i-th element of array x.
DO UNTIL	extension
DO WHILE	extension
DOUBLE{x}	reads/writes an 8-byte floating point variable
EVNT_BUTTON()	Waits for one or more mouse clicks
EVNT_DCLICK()	Sets the speed for double-clicks of a mouse button.
EVNT_KEYBD()	Waits for a key to be pressed and returns a word
EVNT_MESAG()	Waits for the arrival of a message in the event buffer.
EVNT_MOUSE()	Waits for the mouse pointer to be located inside
EVNT_MULTI()	Waits for the occurrence of selected events.
EVNT_TIMER()	waits for a period of time
FATAL	Returns the value 0 or -1 according to the type of error
FIELD	Divides records into fields.
FORM INPUT	Enables the insertion of a character string
FORM INPUT AS	the current value of a$ 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
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
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 put back chars
`KEYTEST n`	similar to INKEY$
`KEYPAD n`	Sets the usage of the numerical keypad.
`KEYPRESS n`	This simulates the pressing of a key.
`LONG{x}`	Reads/writes a 4 byte integer from/to address x. Abbrev
`LOOP UNTIL condition`	extension
`LOOP WHILE condition`	extension
`LSET var=string`	Puts the 'string' in the string variable 'var' left justified

```
MAT ADD a(),b()
MAT ADD a(),x
MAT CPY a([i,j])=b([k,l])[,h,w]
MAT INPUT #i,a()
MAT MUL
MAT MUL a(),x
MAT MUL x=a()*b()
MAT MUL x=a()*b()*c()
MAT NORM a(),{0/1}
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 XCPY a([i,j])=b([k,l])[,h,w]
MAT BASE {0/1}
```

`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/lvalue)

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 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
ON BREAK	influence behavior of CTRL-C
OPTION BASE	determine whether an array is to contain a zero element
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#)
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 <filename>	see RUN
SETDRAW	see DRAW
SINGLE{x}	Reads/writes a 4 byte floating point
SPRITE	Puts a sprite
STORE	Fast save of a string array as a text file.
TRON#	Tron to file
TRON proc	procedure is called before the execution of each command
VTAB	positions the cursor to the specified line number
WRITE	Stores data in a sequential file
_DATA	Specifies the position of the DATA pointer.
STICK	control the joystick (via SDL only)

Following commands have a different meaning and/or syntax in X11-Basic:

GFA-BASIC	X11-Basic
SYSTEM	QUIT
LINE INPUT	LINEINPUT
SOUND	SOUND
WAVE	WAVE
VSYNC	-
ON MENU	MENU
ON MENU GOSUB ...	MENUDEF
MENU a$()	MENUDEF
MENU OFF	-
MENU KILL	MENUKILL
MENU()	-
MONITOR	SYSTEM
EXEC	EXEC
RENAME AS	RENAME

Compiler specifics

- PRINT statements will not compile correctly sometimes. Avoid to use functions and variables in print statements, which are not used anywhere else.
- ON ERROR GOSUB will not work correctly in compiled programs.
- ON ERROR GOTO will not work correctly in compiled programs.
- ON BREAK GOSUB will not work correctly in compiled programs.
- ON BREAK GOTO will not work correctly in compiled programs.

Differences between UNIX, WINDOWS, TomTom and Android versions

9.3. 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:

```
SPRINT var$;[USING...;]...  similar to sprintf() in C
MAT_PRINT or PRINT a()
```

```
========================
implementation of mmap() in X11-Basic:

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().

"I"   --> PROT_READ MAP_PRIVATE
"O"   --> PROT_WRITE MAP_SHARED
"U"   --> PROT_READ PROT_WRITE MAP_SHARED

"*L"  --> MAP_LOCKED

============================================

modifiable lvalues:
MID$()=
CHAR{}=
PRG$()=
new command (for threads):

EXSUB (instead of gosub) procedure

alternative:
FIRE procedure()
or
KICK procedure()

SPAWN  ....
```

```
it must be guaranteed that the program flow control and the access to
variables etc, is thread-safe. This might be difficult....


================================================
USB support:  (not completely done, I need someone who uses this for testi

OPEN "UU",#2,devicename%,vid,pid,class,endpoint
SEND_CONTROL #2,t$
SEND #2,t$
RECEIVE #2,t$
RECEIVE_BULK #2,t$
EOF(#2)
INP?(#2)
CLOSE #2


================================================
SQL support ???

use the sqlite excecutable and SYSTEM$()

================================================
arbitrary precision (floatingpoint) numbers
** integers and floatingpoint/complex numbers
a##=1.84902948755588584888888888888888834

a|=


we need new parsers, type guessing routine,
all operations need to work, complex functions..... Casts...

================================================
Support for the gcrypt library.
-------encryption---------------

LIBGCRYPT:
```

```
hash$=HASH$(data$[,typ%])
sdata$=SIGN$(data$,privkey$)
verify%=VERIFY$(sdata$,pubkey$)
cdata$=ENCRYPT$(data$,key$[,typ%])
data$=DECRYPT$(cdata$,key$[,typ%])

err=KEYGEN(typ%,pubkey$,privkey$)
```

A GNU LICENSE

GNU Free Documentation License

```
  Version 1.2, November 2002

 Copyright (C) 2000,2001,2002  Free Software Foundation, Inc.
     51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
```

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgments", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or non-commercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying

with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C State on the Title page the name of the publisher of the Modified Version, as the publisher.

D Preserve all the copyright notices of the Document.

E Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H Include an unaltered copy of this License.

I Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K For any section Entitled "Acknowledgments" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.

L Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N Do not re-title any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgments", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not

used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgments", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sub-license, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sub-license or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

ACKNOWLEDGEMENTS

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.

I would like to thank every people who help me out with source code, patches, program examples, bug tracking, help and documentation writing, financial support, judicious remarks, and so on...

And here thanks to people, who helped me recently:

in 2012: * Marcos Cruz (beta testing and bug fixing) * Bernhard Rosenkraenzer (send me a patch for 64bit version)

in 2013: * Matthias Vogl (`va_copy` patch for 64bit version) * Eckhard Kruse (for permission to include ballerburg.bas in the samples) * Stewart C. Russell (helped me fix some compilation bugs on Raspberry PI) * Marcos Cruz (beta testing and bug fixing) * James V. Fields (beta testing and correcting the manual)

in 2014: * John Clemens, Slawomir Donocik, Izidor Cicnapuz, Christian Amler, Marcos Cruz, Charles Rayer, Bill Hoyt, and John Sheales (beta testing and bug fixing), Nic Warne and Duncan Roe for helpful patches for the linux target.

in 2015:

* Guillaume Tello, Wanderer, and John Sheales (beta testing and bug fixing)

X11-Basic is build on top of many free softwares, and could not exist without them.

X11-Basic uses functionallity of the gmp library, the GNU multiple precision arithmetic library, Version 5.1.3. Copyright 1991, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007 Free Software Foundation, Inc.

For details, see: https://gmplib.org/

X11-Basic also uses functionallity if the LAPACK library. LAPACK (Linear Algebra Package) is a standard software library for numerical linear algebra.

For details, see: http://www.netlib.org/lapack/

X11-Basic uses the lodepng code for the PNG bitmap graphics support. Copyright (c) 2005-2013 Lode Vandevenne

So I would like to thank every people involved in the following projects:

Linux GCC and all of the GNU tools, of course. The readline library. The SDL

(Simple Direct Media) library. Creator of the 8x16 font "spat_a". The Burrow-Wheeler-Transform. And any other libraries used by X11-Basic.

Some pieces of code of X11-Basic are based on third party software:

The AES user interface routines are based on OpenGEM. The FloodFill algorithm is based on xxxx. Pictograms for mousecursers and fill patterns are based on TOS / ATARI ST. The order-0 adaptive arithmetic decoding function contains the source code from the 1987 CACM article by Witten, Neal, and Cleary.

Parts of this manual (the chapter about CGI programming) are taken from or based on a documentation which used to be freely available on the internet in the 1990ies. (I hope, it was public domain or similar.) Unfortunately I have lost the link to its source. If you happen to know its potential source, please let me know so I can reference it here and give credits to its author.