SmallBASIC Guide

A User's Guide for SmallBASIC
Edition Alpha, for SmallBASIC Version 0.9.0
August 2003.

Nicholas D. Christopoulos

This is Edition Alpha of *SmallBASIC Guide: A User's Guide for SmallBASIC*, for the 0.9.0 (or later) version of the SmallBASIC language.

The SmallBASIC Team

Nicholas Christopoulos
nereus@freemail.gr, Athens - Greece.
Original author and project manager.

Chris Warren-Smith
cwarrens@twpo.com.au, Adelaide - South Australia.
Franklin's eBookMan version, and SB developer
http://www.twpo.com.au/cwarrens/ebm

Laurent Poujoulat
lpoujoulat@wanadoo.fr, Bondy - France.
PalmOS 5 version, and SB developer

Tim Corcoran
tim@whdl.com, USA.
Sony Clie version, and SB developer

Earle F. Philhower
earle@ziplabel.com, USA.
Helio (VTOS) version
http://www.ziplabel.com/

Web Site
http://smallbasic.sf.net

Forum
http://smallbasic.sf.net/forum

Table of Contents

1 Introduction

1.1 Welcome to SmallBASIC

SmallBASIC (SB) is a simple computer language, featuring a clean interface, strong mathematics and string library. We feel it is an ideal tool for experimenting with simple algorithms, for having fun.

1.1.1 About BASIC

BASIC is a very simple language and it is a perfect tool for calculations or utilities. Its name stands for (B)eginners (A)ll-purpose (S)ymbolic (I)nstruction (C)ode. It was developed by John Kemeny and Thomas Kurtz at Dartmouth College during the middle of 1960, and was one of the most popular languages for several decades.

However, at the last decades it was upgraded to survive on the new programming environments. It was modernized and that was hard required.

In the first upgrade, BASIC was transformed to a structured language. As far, as I known, the first structured BASIC was the QuickBASIC (QB), a Microsoft product. Several structured dialects was followed from other companies.

In the second upgrade, BASIC was transformed to an (almost) object-oriented language. As far, as I known, the first OO BASIC was the VisualBASIC (VB), a Microsoft product. In that stage BASIC was become very problematic, since, Microsoft was introduced ObjectPascal and C++ technologies in a language with very different design and purpose of existance!

Anyway, we strongly disagree with the "new" feautures and the way that are implemented in VB. Every language created for specified purposes, BASIC for beginners, C for low-level programming, Prolog for AI, etc. VB it is not object-oriented nor a simple language (anymore), but it is a bad designed mix of other languages.

1.1.2 About SmallBASIC

SmallBASIC was created by Nicholas Christopoulos in May of 2000, to be used as an advanced calculator for his Palm IIIx handheld device. In Jan of 2001, SB moved to the web as an GPL project.

Because SB was designed for that small device (Palm IIIx), and because was small compared to desktop-computer BASICs, it takes the prefix 'Small'.

SB is a structured version of BASIC and includes a lot of new feautures such matrices, algebra functions, powerfull string library, etc. A lot of its feautures does not exists in the most languages, but on the other hand, SB does not supports GUI and other feautures that are common in today languages.

1.1.2.1 Purpose

BASIC is easy to learn and simple to use, and this is the spirit of SB. Instead of other BASIC versions, as VB, our version intent to sucrifice everything in the altar of simplicity.

The world is full of languages, SB does not offers something new, but intents to offer what is lost in our days. A simple tool for easy to write programs, an easy way to do some maths and build some scripts.

Our priorities are to build

- An extremly easy learned language.
- An extremly easy to use language.
- An ideal tool for experimenting on programming.
- An excellent tool for mathematics.
- An excellent tool for shell-scripts.

1.1.2.2 Cross-platform

Now, SB can run on more platforms than PalmOS, such Linux, DOS, Win32, EBM and VTOS. An mechanism had inserted and porting to different platforms is an easy task. For this reason, SB claims that it is a cross-platform language.

However, SB is based primary on Unix systems. A lot of feautures (for example, Units, C-Modules) does not implemented on other systems yet.

1.2 Useful notes for beginners

1.2.1 What we must already know

Integer Number
: A number that does not have a fractional part.

Floating-Point Number
Real Number
: Often referred to in mathematical terms as real number, this is just a number that can have a fractional part.

Numeric Constants
: Numeric constants may be entered with any number of digits. For extremly large or small numbers, it is usually more convenient to use scientific notation.

  In scientific notation, a number is given as a mantissa (a number with one place to the left of the decimal point) times 10 raised to an integer power.

  Scientific Notation Examples:

```
15    is expressed as 1.5*10^1,  is typed as 1.5E+1
150   is expressed as 1.5*10^2,  is typed as 1.5E+2
1500  is expressed as 1.5*10^3,  is typed as 1.5E+3
1500  is expressed as -1.5*10^3, is typed as -1.5E+3
0.15  is expressed as 1.5*10^-1, is typed as 1.5E-1
```

Numeric Expressions

Numeric expressions are constructed from numeric constants, variables, and functions using the arithmetic operators for addition (+), substraction (-), multiplication (*), division (/) and exponentiation (^).

The minus sign (-) can be used either to indicate subtraction or as a unary minus.

The normal hierarchy for evaluating a numeric expression is exponentiation, followed by multiplication and division, and then by addition and subtraction. However, any part of a numeric expression that is enclosed in parenthesis is evaluated first.

In SB more operators are supported. For further reading please see 'Operators' section.

String A datum consisting of a sequence of characters, such as '`I am a string`'.

String Constants

String constants are the texts enclosed in double quotation marks, like this:

```
"I am a string constant!"
```

String Expressions

String expressions are constructed from string variables, string constants, and function references using the operation for concatenation (+) to combine strings.

Example:

```
x = "HI" + " THERE!"
```

In this example, the *x* is equal to "HI THERE!".

Relational Expressions

Relational expressions are most often used in the IF-THEN statement, but may be used anywhere that numeric expressions are allowed. A relational expression has a value of non-zero if it is true and a value of 0 if it is false. Relational operators are performed, from left to right, after all arithmetic operations are completed. The most usual relational operators are:

```
Equal to (=),     Not equal to (<>)
Less than (<),    Less than or equal to (<=)
Greater than (>), Greater than or equal to (>=)
```

Boolean Expressions (also known as Logical Expressions)

Named after the English mathematician Boole.

Logical expressions are used usual with relational expressions. The logical operators are AND, OR and NOT. If true, logical expressions are given a value of non-zero. If false, they are given a value of 0.

A logical expression using `AND` is true if both its left and right clauses are true.

A logical expression using `OR` is true if either its left or its right, or both, clauses are true.

A logical expression using `NOT` is true if the following clause it is not true.

Variable A variable is a name which represents a value. Actually the value exists in memory, a variable represent the memory space that holds the value.

Array A grouping of multiple values under the same variable.

Keyword In a language, a keyword is a word that has special meaning. Keywords are reserved and may not be used as variable names.

Statement An important unit of the language

Command Also, known as build-in procedure

Comment ...

Assignment
An expression that changes the value of some variable. The value that you can assign to is called an *lvalue*. The assigned values are called *rvalues*.

Procedure
Routine
SubRoutine
A specialized group of statements used to encapsulate general or program-specific tasks. SB has a number of built-in procedures, and also allows you to define your own.

In older times those groups of statements was called routines. This is why the 'procedures' are called `SUB`(routines) in *BASIC*.

Function A specialized group of statements used to encapsulate general or program-specific tasks. SB has a number of built-in functions, and also allows you to define your own.

The difference between function and procedure is that, function can return a value and can be used inside expressions. Procedure can't do that.

Space The character generated by hitting the space bar on the keyboard.

Tab The character generated by hitting the ⟨TAB⟩ key on the keyboard. It usually expands to up to eight or four spaces upon output.

Whitespace
A sequence of space, TAB, vertical tab, from-feed, or newline characters occurring inside an input record or a string.

1.2.2 How to read the syntax

- Everything is written inside of [ ] characters are optional values.
- Everything is written inside of `{ }` characters means you must select one of them.
- The symbol `|` means OR.
- The symbols ... means you can repeat the previous syntax.
- The keywords are written with capital letters.
- The parameters are written with lower letters.
- The keywords with suffix ( ) are functions.
- The parameters with suffix ( ) are arrays.

Example #1:

```
FOO     <- This is keyword
FOO()   <- This is function
foo     <- This is variable/parameter
foo()   <- This is array/parameter
{A|B}   <- This means that you must type A or B
[{A|B}] <- This means that you must use A or B or nothing
```

Example #2:

```
FOO a[, x]
```

This means that you must give the first parameter (a) but you can use the second (x) only if you want to. But if you want to use the (x) you must also separate it from (a) with a comma.

Example #3:

```
FOO var [{,|;} var2 [...]]
```

This means that you must use the first parameter. You can also use second parameter but you must separate it with ',' or ';'. You can also repeat the last syntax more times

The following code respects this syntax

```
FOO a
FOO a, b
FOO a; b
FOO a, b; c
FOO a, b, c
FOO a; b; c, d; e, f, g
```

Example #4:

```
' Syntax: TEST {1|2}
TEST 1
TEST 2

' Syntax: TEST [HI]
TEST
TEST HI
```

1.3 Running SB Interactively

Interactive mode is supported only on console mode (Unix, DOS or Win32 Console). The SmallBASIC is started by typing `sbasic`. When the SB starts, a prompt appears at which we can run any OS command or starting typing the program.

```
# sbasic
SmallBASIC VERSION 0.9.0
        Copyright (c) 2000-2003 Nicholas Christopoulos

Type 'HELP' for help; type 'BYE' or press Ctrl+C for exit.

* READY *

/home/nikosc>
```

Type the following program, pressing ⟨ENTER⟩ at the end of each program line.

```
10 PRINT "Are you ready"
20 PRINT "to learn BASIC?"
30 END
```

Check the program now to see if there are any typing mistakes. If there are, use ⟨up-arrow⟩ or ⟨down-arrow⟩ to find the previously typed lines. Use ⟨right-arrow⟩ or ⟨left-arrow⟩ to move inside the line. Fix the problem and press ⟨ENTER⟩.

When you are ready to see the program in action, type *CLS*⟨ENTER⟩. The screen will be cleared.

Now, type *RUN*⟨ENTER⟩.

```
/home/nikosc> run

Are you ready
to learn BASIC?

 * DONE *
/home/nikosc>
```

Now, type *LIST*⟨ENTER⟩ to see your program lines.

```
/home/nikosc> list
  10: PRINT "Are you ready"
  20: PRINT "to learn BASIC?"
  30: END
/home/nikosc>
```

This is the simpliest way to run SB, usefull when we want to do some temporary calculations. It is also give us a taste of the old times.

1.4 Running SB

The usual way is type our program to an editor and save that in a file. Typically an SB program file must be terminated with '`.sb`' or '`.bas`'. That helps the OS to understand the type of the file.

Create a file with an editor like `joe`, `kate` or `EDIT`. Give to them a name, for example '`myprog.sb`'. Type some commands like our previous example, save it and exit from the editor. Now, run SB by using the file-name as parameter.

```
# sbasic -q myprog.sb
Hello, world!
#
```

The -q option tells to SB to be quite.

There are also more advanced ways to run a program with SB. For example, type a program that prints out SB commands!

myprog.sb

```
PRINT "PRINT 3/4"
```

Now, run it by using `|` (pipe) symbol.

```
# sbasic -q myprog.sb | sbasic -q
0.75
#
```

We did something very simple. The first sbasic runs the '`myprog.sb`', this program prints out the `PRINT 3/4` text. The second `sbasic` was execute the result of the first `sbasic` which was the code `PRINT 3/4`.

1.4.1 Unix script executables

In Unices we can create script executables. Those script are working similar to the common executables.

We need only two things.

a) A line at the beginning of our program
`#!/usr/bin/sbasic -q`

b) And sets the executable attribute of the file
`# chmod 0777 myprog.sb`

Now we can run it as usual.

```
# ./myprog.sb
Hello, world!
#
```

We can find more on scripts, paths and Unix attributes on Unix manuals.

2 The language

This chapter documents language structure.

2.1 Constants and Variables

- All user variables (include arrays) are 'Variant'. That means the data-type is invisible to user.
- User-defined data types are not allowed.
- Arrays are always dynamic, even if you had declared their size, with dynamic size and type of elements.

However, SmallBASIC uses, internally, 4 data-types

1. Integer (32bit)
2. Real (64bit)
3. String (<32KB on 16bit / 2GB on 32bit)
4. Array (~2970 elements on 16bit / ~50M elements on 32bit)

Convertions between those types are performed internaly. In anycase there are functions for the user to do it manualy.

2.1.1 Variable names

Variable names can use any alphanumeric characters, extended characters (ASCII codes 128-255 for non-English languages) the symbol '_', and the symbol '$'.

The first character of the name cannot be a digit nor a '$'.

2.1.2 About the dollar-symbol

The symbol '$' is supported for compatibility. Since in SmallBASIC there are no data-types its use is meaningless.

The dollar in function names will be ignored

The dollar in variable names will be count as part of the name (that means v and v$ are two different variables). It can be used only as the last character of the name, and only one allowed.

The dollar in system variables names will be ignore it (that means COMMAND and COMMAND$ is the same)

Example of variable names:

```
abc, a_c, _bc, ab2c, abc$ -> valid names
1cd, a$b, $abc            -> invalid names
```

2.1.3 Integers

This is the default data type. You can declare integers in decimal, hexadecimal, octal and binary form.

```
x = 256   ’
x = 0x100 ’ Hexadecimal form 1
x = &h100 ’ Hexadecimal form 2


x = 0o400 ’ Octal form 1
x = &o400 ’ Octal form 2


x = 0b111 ’ Binary form 1
x = &b111 ’ Binary form 2
```

2.1.4 Reals

Any number which out-bounds the limits or an ’integer’ or had decimal digits will be converted automatically to real.

```
x = .25
x = 1.2
```

Reals can be also written by using scientific notation. 1E+2 or 1E-+2, 5E–2, 2.6E-0.25, etc

2.1.5 Strings

Strings may be appended to one another using the + operator.

```
b = "Hello, " + "world!"
```

2.1.6 Constants

Constant variables can be declared by using the keyword CONST

```
CONST my_pi = 3.14
```

2.2 System Variables

System variables, are constant variables for the programmer. Those variables get values or modified at run-time by the SB’s subsystem.

OSNAME	Operating System name
OSVER	Operating System Version (0xAABBCC (A=major, B=minor, C=patch))
SBVER	SmallBASIC Version (0xAABBCC)
PI	3.14..

XMAX	Graphics display, maximum x (width-1)
YMAX	Graphics display, maximum y (height-1) value
BPP	Graphics display: bits per pixel (color resolution)
VIDADR	Video RAM address (only on specific drivers)
CWD	Current Working Directory
HOME	User's home directory
COMMAND	Command-line parameters
TRUE	The value 1
FALSE	The value 0

2.3 Operators

Sorted by priority

( )	Parenthesis
+, -	Unary
~	bitwise NOT
NOT or !	Logical NOT (NOT false = true)
^	Exponentiation
*, /, \	Multiplication, Division, Integer Division
% or MOD	Reminder (QB compatible: a=int(a), b=int(b), a-b*(a/b))
MDL	Modulus (a%b+b*(sgn(a)<>sgn(b)))
+, -	Addition/Concatenation, Subtraction
=	Equal
<> or !=	Not Equal
>, <	Less Than, Greater Than
=>, =<	Less or Equal, Greater or Equal
>=, <=	Less or Equal, Greater or Equal
IN	belongs to ... (see "The IN operator")
LIKE	Regular expression match (see "The LIKE operator")
AND or &&	Logical AND
OR or ||	Logical OR
BAND or &	bitwise AND

BOR or	bitwise OR
|	
EQV	bitwise EQV
IMP	bitwise IMP
XOR	bitwise XOR
NAND	bitwise NAND
NOR	bitwise NOR
XNOR	bitwise XNOR

2.4 Special Characters

&h or 0x	Prefix for hexadecimal constant (0x1F, &h3C)
&o or 0o	Prefix for octal constant (0o33, &o33)
&b or 0b	Prefix for binary constant (0b1010, &b1110)
[,;]	Array definition (function ARRAY()) ($1)
<<	Appends to an array (command APPEND) ($1)
++	Increase a value by 1 (x = x + 1) ($1)
–	Decrease a value by 1 (x = x - 1) ($1)
p=	Another LET macro (x = x p ...). Where p any character of -+/*^%&
:	Separates commands typed on the same line
&	Join code lines (if its the last character of the line). The result line its must not exceed the max. line size.
#	Meta-command (if its the first character of the line) or prefix for file handle
@	The 'at' symbol can by used instead of BYREF ($1)
'	Remarks

$1 *Pseudo operators. These operators are replaced by compiler with a command or an expression.*

2.5 The OPTION keyword

OPTION *keyword parameters* [Statement]

This special command is used to pass parameters to the SB-environment. There are two styles for that, the run-time (like BASE) which can change the value at run-time, and the compile-time (like PREDEF) which used only in compile-time and the value cannot be changed on run-time.

2.5.1 Run-Time

OPTION *BASE lower-bound* [Statement]

The `OPTION BASE` statement sets the lowest allowable subscript of arrays to *lower-bound*. The default is zero. The `OPTION BASE` statement can be used in any place in the source code but that is the wrong use of this except if we have a **good** reason.

In most cases the `OPTION BASE` must declared at first lines of the program before any `DIM` declaration.

OPTION *MATCH* {*PCRE* [*CASELESS*]|*SIMPLE*} [Statement]
Sets as default matching algorithm to (P)erl-(C)ompatible (R)egular (E)xpressions library or back to simple one. Matching-algorithm is used in `LIKE` and `FILES`.

PRCE works only in systems with this library and it must be linked with. Also, there is no extra code on compiler which means that SB compiles the pattern everytime it is used.

2.5.2 Compile-Time

OPTION *PREDEF parameter* [Statement]
Sets parameters of the compiler. Where *parameter*

'`QUITE`' Sets the quite flag (-q option)

'`COMMAND cmdstr`'
Sets the `COMMAND$` string to *cmdstr* (useful for debug reasons)

'`GRMODE [widthxheight[xbpp]]`'
Sets the graphics mode flag (-g option) or sets the prefered screen resolution. Example: (Clie HiRes)

```
OPTION PREDEF GRMODE 320x320x16
```

'`TEXTMODE`'
Sets the text mode flag (-g- option)

'`CSTR`' Sets as default string style the C-style special character encoding ('\')

2.6 Meta-commands

#!... [Macro]
Used by Unix to make source runs as a script executable

#sec: *section-name* [Macro]
Used internally to store the section name. Sections names are used at limited OSes like PalmOS for multiple 32kB source code sections. With a few words **DO NOT USE IT**!

#inc: *file* [Macro]
Used to include a SmallBASIC source file into the current BASIC code

#unit-path: *path* [Macro]
Used to setup additional directories for searching for unit-files This meta does nothing more than to setting up the environment variable SB_UNIT_PATH. Directories on Unix must separated by ':', and on DOS/Windows by ';'

Examples

```
...
#inc:"mylib.bas"
...
MyLibProc "Hi"
```

2.7 Arrays and Matrices

Define a 3x2 matrix

```
A = [11, 12; 21, 22; 31, 32]
```

That creates the array

```
| 11  12 |
| 21  22 | = A
| 31  32 |
```

The comma used to separate column items; the semi-colon used to separate rows. Values between columns can be omitted.

```
A = [ ; ; 1, 2 ; 3, 4, 5]
```

This creates the array

```
| 0  0  0 |
| 1  2  0 | = A
| 3  4  5 |
```

Supported operators:

Add/sub:

```
B = [1, 2; 3, 4]: C = [5, 6; 7, 8]

A = B + C
C = A - B
```

Equal:

```
bool=(A=B)
```

Unary:

```
A2 = -A
```

Multiplication:

```
A = [1, 2; 3, 4]: B = [5 ; 6]
C = A * B
D = 0.8 * A
```

Inverse:

```
A = [ 1, -1, 1; 2, -1, 2; 3, 2, -1]
? INVERSE(A)
```

Gauss-Jordan:

```
? "Solve this:"
? "  5x - 2y + 3z = -2"
? " -2x + 7y + 5z =  7"
? "  3x + 5y + 6z =  9"
```

```
?
A = [ 5, -2, 3; -2, 7, 5; 3, 5, 6]
B = [ -2; 7; 9]
C = LinEqn(A, B)
? "[x;y;z] = "; C
```

There is a problem with 1 dimension arrays, because 1-dim arrays does not specify how SmallBASIC must see them.

```
DIM A(3)

| 1 2 3 | = A

or

| 1 |
| 2 | = A
| 3 |
```

And because this is not the same thing. (ex. for multiplication) So the default is columns

```
DIM A(3) ' or A(1,3)

| 1 2 3 | = A
```

For vertical arrays you must declare it as 2-dim arrays Nx1

```
DIM A(3,1)

| 1 |
| 2 | = A
| 3 |
```

2.8 Nested arrays

Nested arrays are allowed

```
A = [[1,2] , [3,4]]
B = [1, 2, 3]
C = [4, 5]
B(2) = C
print B
```

This will be printed

```
[1, 2, [4, 5], 3]
```

You can access them by using a second (or third, etc) pair of parenthesis.

```
B(2)(1) = 16
print B(2)(1)

Result:
    16
```

2.9 The operator IN

IN operator is used to compare if the left-value belongs to right-value.

```
' Using it with arrays
print 1 in [2,3]        :REM FALSE
print 1 in [1,2]        :REM TRUE
print "b" in ["a", "b", "c"] :REM TRUE
...
' Using it with strings
print "na" in "abcde"   :REM FALSE
print "cd" in "abcde"   :REM TRUE
...
' Using it with number (true only if left = right)
print 11 in 21          :REM FALSE
print 11 in 11          :REM TRUE
...
' special case
' auto-convert integers/reals
print 12 in "234567"    :REM FALSE
print 12 in "341256"    :REM TRUE
```

2.10 The operator LIKE

LIKE is a regular-expression operator. It is compares the left part of the expression with the pattern (right part). Since the original regular expression code is too big (for handhelds), I use only a subset of it, based on an excellent old stuff by J. Kercheval (match.c, public-domain, 1991). But there is an option to use PCRE (Perl-Compatible Regular Expression library) on systems that is supported (Linux); (see OPTION).

The same code is used for filenames (FILES(), DIRWALK) too.

In the pattern string:

*	matches any sequence of characters (zero or more)
?	matches any character
[SET]	matches any character in the specified set,
[!SET] or [^SET]	matches any character not in the specified set.

A set is composed of characters or ranges; a range looks like character hyphen character (as in 0-9 or A-Z). [0-9a-zA-Z_] is the minimal set of characters allowed in the [..] pattern construct.

To suppress the special syntactic significance of any of '[]*?!^-\', and match the character exactly, precede it with a '\'.

```
? "Hello" LIKE "*[oO]" : REM TRUE
? "Hello" LIKE "He??o" : REM TRUE
? "Hello" LIKE "hello" : REM FALSE
? "Hello" LIKE "[Hh]*" : REM TRUE
```

2.11 The pseudo-operator <<

This operator can be used to append elements to an array.

```
A << 1
A << 2
A << 3

? A(1)
```

2.12 Subroutines and Functions

Syntax of procedure (SUB) statements

```
SUB name [([BYREF] par1 [, ...[BYREF] parN)]]
  [LOCAL var[, var[, ...]]]
  [EXIT SUB]
  ...
END
```

Syntax of function (FUNC) statements

```
FUNC name[([BYREF] par1 [, ...[BYREF] parN)]]
  [LOCAL var[, var[, ...]]]
  [EXIT FUNC]
  ...
  name=return-value
END
```

On functions you must use the function's name to return the value. That is, the function-name acts like a variable and it is the function's returned value.

The parameters are 'by value' by default. Passing parameters by value means the executor makes a copy of the parameter to stack. The value in caller's code will not be changed.

Use `BYREF` keyword for passing parameters 'by reference'. Passing parameters by reference means the executor push the pointer of variable into the stack. The value in caller's code will be the changed.

```
' Passing 'x' by value
SUB F(x)
  x=1
END


x=2
F x
? x:REM displays 2
' Passing 'x' by reference
SUB F(BYREF x)
  x=1
END
```

```
x=2
F x
? x:REM displays 1
```

You can use the symbol '@' instead of BYREF. There is no difference between @ and BYREF.

```
SUB F(@x)
  x=1
END
```

On a multi-section (PalmOS) applications sub/funcs needs declaration on the main section.

```
#sec:Main
declare func f(x)

#sec:another section
func f(x)
...
end
```

Use the LOCAL keyword for local variables. LOCAL creates variables (dynamic) at routine's code.

```
SUB MYPROC
  LOCAL N:REM LOCAL VAR
  N=2
  ? N:REM displays 2
END

N=1:REM GLOBAL VAR
MYPROC
? N:REM displays 1
```

You can send arrays as parameters.

When using arrays as parameters its better to use them as BYREF; otherwise their data will be duplicated in memory space.

```
SUB FBR(BYREF tbl)
  ? FRE(0)
  ...
END

SUB FBV(tbl)
  ? FRE(0)
  ...
END

' MAIN
DIM dt(128)
...
? FRE(0)
```

```
FBR dt
? FRE(0)
FBV dt
? FRE(0)
```

Passing & returning arrays, using local arrays.

```
func fill(a)
  local b, i

  dim b(16)
  for i=0 to 16
    b(i)=16-a(i)
  next
  fill=b
end

DIM v()
v=fill(v)
```

2.13 Single-line Functions

There is also an alternative `FUNC/DEF` syntax (single-line functions). This is actually a macro for compatibility with the BASIC's DEF FN command, but quite usefull.

Syntax:

```
FUNC name[(par1[,...])] = expression
or
DEF name[(par1[,...])] = expression
```

```
DEF MySin(x) = SIN(x)
? MySin(pi/2)
```

2.14 Nested procedures and functions

One nice feauture, are the nested procedures/functions. The nested procedures/functions are visible only inside the "parent" procedure/function.

There is no way to access a global procedure with the same name of a local... yet...

```
FUNC f(x)
    Rem Function: F/F1()
    FUNC f1(x)
        Rem Function: F/F1/F2()
        FUNC f2(x)
            f2=cos(x)
        END
        f1 = f2(x)/4
    END
    Rem Function: F/F3()
```

```
    FUNC f3
        f3=f1(pi/2)
    END
REM
? f1(pi) : REM OK
? f2(pi) : REM ERROR
f = x + f1(pi) + f3 : REM OK
END
```

2.15 Units (SB libraries)

* Linux ONLY for now *

Units are a set of procedures, functions and/or variables that can be used by another SB program or SB unit. The main section of the unit (commands out of procedure or function bodies) is the initialization code.

A unit declared by the use of `UNIT` keyword.

```
UNIT MyUnit
```

The functions, procedure or variables which we want to be visible to another programs must be declared with the `EXPORT` keyword.

```
UNIT MyUnit
EXPORT MyF
...
FUNC MyF(x)
...
END
```

***** *Keep file-name and unit-name the same. That helps the SB to automatically recompile the required units when it is needed.*

To link a program with a unit we must use the `IMPORT` keyword.

```
IMPORT MyUnit
```

To access a member of a unit we must use the unit-name, a point and the name of the member.

```
IMPORT MyUnit
...
PRINT MyUnit.MyF(1/1.6)
```

Full example:

file my_unit.bas:

```
UNIT MyUnit

EXPORT F, V

REM a shared function
FUNC F(x)
```

```
    F = x*x
END

REM a non-shared function
FUNC I(x)
    I = x+x
END

REM Initialization code
V="I am a shared variable"
L="I am invisible to the application"
PRINT "Unit 'MyUnit' initialized :)"
```

file my_app.bas:

```
IMPORT MyUnit

PRINT MyUnit.V
PRINT MyUnit.F(2)
```

2.16 The pseudo-operators ++/–/p=

The ++ and – operators are used to increase or decrease the value of a variable by 1.

```
x = 4
x ++ : REM x <- x + 1 = 5
x -- : REM x <- x - 1 = 4
```

The generic p= operators are used as in C Where p any character of -+/*^%&|

```
x += 4 : REM x <- x + 4
x *= 4 : REM x <- x * 4
```

All these pseudo-operators are not allowed inside of expressions

```
y = x ++ ' ERROR
z = (y+=4)+5 ' ALSO ERROR
```

2.17 The USE keyword

This keyword is used on specific commands to passing a user-defined expression.

Example:

```
SPLIT s," ",v USE TRIM(x)
```

In that example, every element of V() will be 'trimmed'.

Use the x variable to specify the parameter of the expression. If the expression needs more parameter, you can use also the names y and z

2.18 The DO keyword

This keyword is used to declare single-line commands. It can be used with WHILE and FOR-family commands.

Example:

```
FOR f IN files("*.txt") DO PRINT f
...
WHILE i < 4 DO i ++
```

Also, it can be used by IF command (instead of THEN), but is not suggested.

3 Programming Tips

Programmers must use clean and logical code. Weird code may be faster but it is not good.

3.1 Using LOCAL variables

When a variable is not declared it is by default a global variable. A usual problem is that name may be used again in a function or procedure.

```
FUNC F(x)
  FOR i=1 TO 6
    ...
  NEXT
END

FOR i=1 TO 10
  PRINT F(i)
NEXT
```

In this example, the result is a real mess, because the *i* of the main loop will always (except the first time) have the value 6!

This problem can be solved if we use the LOCAL keyword to declare the *i* in the function body.

```
FUNC F(x)
  LOCAL i

  FOR i=1 TO 6
    ...
  NEXT
END

FOR i=1 TO 10
  PRINT F(i)
NEXT
```

It is good to declare all local variables on the top of the function. For compatibility reasons, the func./proc. variables are not declared as 'local' by default. That it is WRONG but as I said ... compatibility.

3.2 Loops and variables

When we write loops it is much better to initialize the counters on the top of the loop instead of the top of the program or nowhere.

```
i = 0
REPEAT
  ...
  i = i + 1
```

```
UNTIL i > 10
```

Initializing the variables at the top of the loop, can make code better readable, and can protect us from usual pitfalls such as forgeting to giving init value or re-run the loop without reset the variables.

3.3 Loops and expressions

FOR-like commands are evaluate the 'destination' everytime. Also, loops are evaluate the exit-expression everytime too.

```
FOR i=0 TO LEN(FILES("*.txt"))-1
    PRINT i
NEXT
```

In that example the 'destination' is the LEN(FILES("*.txt"))-1 For each value of i the destination will be evaluated. That is WRONG but it is supported by BASIC and many other languages.

So, it is much better to be rewritten as

```
idest=LEN(FILES("*.txt"))-1
FOR i=0 TO idest
    PRINT i
NEXT
```

Of course, it is much faster too.

4 Commands

REM *comment* [Statement]

Adds explanatory text to a program listing. *comment* commentary text, ignored by BASIC.

Instead of the keyword we can use the symbol ' or the #. The # can be used as remarks only if its in the first character of the line.

Example:

```
' That text-line is just a few remarks
...
REM another comment
...
# one more comment
```

LET *var* = *expr* [Statement]

Assigns the value of an expression to a variable. The `LET` is optional.

var A valid variable name.

expr The value assigned to variable.

Example:

```
LET x = 4
x = 1                ' Without the LET keyword
z = "String data"    ' Assign string
...
DIM v(4)
z=v                  ' Assign array (z = clone of v)
```

CONST *name* = *expr* [Statement]

Declares a constant.

name An identifier that follows the rules for naming BASIC variables.

expr An expression consisting of literals, with or without operators, only.

Example:

```
COSNT G = 6.67259E-11
```

DIM *var*([*lower TO*] *upper* [, ...]) [, ...] [Statement]

The `DIM` statement reserves space in computer's memory for arrays. The array will have (upper-lower)+1 elements. If the *lower* is not specified, and the `OPTION BASE` hasn't used, the arrays are starting from 0.

Example:

```
REM One dimension array of 7 elements, starting from 0
DIM A(6)
...
REM One dimension array of 6 elements, starting from 1
DIM A(1 TO 6)
```

```
...
REM Three dimension array
DIM A(1 TO 6, 1 TO 4, 1 TO 8)
...
REM Allocating zero-length arrays:
DIM z()
...
IF LEN(Z)=0 THE APPEND Z, "The first element"
```

LABEL *name* [Statement]

Defines a label. A label is a mark at this position of the code.

There are two kinds of labels, the 'numeric' and the 'alphanumeric'.

'Numeric' labels does not needed the keyword `LABEL`, but 'alphanumeric' does.

Example:

```
1000 ? "Hello"
...
LABEL AlphaLabel: ? "Hello"
...
GOTO 1000
GOTO AlphaLabel
```

GOTO *label* [Statement]

Causes program execution to branch to a specified position (label).

GOSUB *label* [Statement]

Causes program execution to branch to the specified label; when the `RETURN` command is encountered, execution branches to the command immediately following the most recent `GOSUB` command.

RETURN [Statement]

Execution branches to the command immediately following the most recent `GOSUB` command.

```
...
GOSUB my_routine
PRINT "RETURN sent me here"
...
LABEL my_routine
PRINT "I am in my routine"
RETURN
```

ON *GOTO|GOSUB label1* [*, ..., labelN*] [Statement]

Causes BASIC to branch to one of a list of labels.

expr A numeric expression in the range 0 to 255. Upon execution of the ON...GOTO command (or ON...GOSUB), BASIC branches to the nth item in the list of labels that follows the keyword GOTO (or GOSUB).

FOR *counter = start TO end [STEP incr] ... NEXT* [Statement]

Begins the definition of a FOR/NEXT loop.

counter A numeric variable to be used as the loop counter.

start A numeric expression; the starting value of counter.

end A numeric expression; the ending value of counter.

incr A numeric expression; the value by which counter is incremented or decremented with each iteration of the loop. The default value is +1.

BASIC begins processing of the FOR/NEXT block by setting counter equal to start. Then, if 'incr' is positive and counter is not greater than end, the commands between the FOR and the NEXT are executed.

When the NEXT is encountered, counter is increased by 'incr', and the process is repeated. Execution passes to the command following the NEXT if counter is greater than end.

If increment is negative, execution of the FOR/NEXT loop is terminated whenever counter becomes less than end.

FOR/NEXT loops may be nested to any level of complexity, but there must be a NEXT for each FOR.

Example:

```
FOR C=1 TO 9
    PRINT C
NEXT
```

FOR *element IN array ... NEXT* [Statement]

Begins the definition of a FOR/NEXT loop.

element A variable to be used as the copy of the current element.

array An array expression

The commands-block will repeated for LEN(array) times. Each time the 'element' will holds the value of the current element of the array.

FOR/NEXT loops may be nested to any level of complexity, but there must be a NEXT for each FOR.

Example:

```
A=[1,2,3]
FOR E IN A
    PRINT E
NEXT
...
' This is the same with that
A=[1,2,3]
FOR I=LBOUND(A) TO UBOUND(A)
    E=A(I)
    PRINT E
NEXT
```

WHILE *expr ... WEND* [Statement]

Begins the definition of a WHILE/WEND loop.

expr An expression

BASIC starts by evaluating expression. If expression is nonzero (true), the next command is executed. If expression is zero (false), control passes to the first command following the next `WEND` command.

When BASIC encounters the `WEND` command, it reevaluates the expression parameter to the most recent `WHILE`. If that parameter is still nonzero (true), the process is repeated; otherwise, execution continues at the next command.

WHILE/WEND loops may be nested to any level of complexity, but there must be a WEND for each WHILE.

Example:

```
C=1
WHILE C<10
    PRINT C
    C=C+1
WEND
...
' This is the same with that
FOR C=1 TO 9
    PRINT C
NEXT
```

REPEAT *... UNTIL expr* [Statement]

Begins the definition of a REPEAT/UNTIL loop.

expr An expression

BASIC starts executing the commands between the REPEAT and UNTIL commands. When BASIC encounters the UNTIL command, it evaluates the expression parameter. If that parameter is zero (false), the process will be repeated; otherwise, execution continues at the next command.

REPEAT/UNTIL loops may be nested to any level of complexity, but there must be an UNTIL for each REPEAT.

Example:

```
C=1
REPEAT
    PRINT C
    C=C+1
UNTIL C=10
...
' This is the same with that
FOR C=1 TO 9
    PRINT C
NEXT
```

IF ... [Statement]

Syntax:

```
IF expression1 [THEN]
    .
    . [commands]
    .
[ [ELSEIF | ELIF] expression2 [THEN]
    .
    . [commands]
    .
]
[ELSE
    .
    . [commands]
    .
]
{ ENDIF | FI }
```

Block-style IF.

Causes BASIC to make a decision based on the value of an expression.

expression An expression; 0 is equivalent to FALSE, while all other values are equivalent to TRUE.

commands
One or more commands.

Each expression in the IF/ELSEIF construct is tested in order. As soon as an expression is found to be TRUE, then its corresponding commands are executed. If no expressions are TRUE, then the commands following the ELSE keyword are executed. If ELSE is not specified, then execution continues with the command following the ENDIF.

IF, ELSE, ELSEIF, and ENDIF must all be the first keywords on their respective lines.

THEN is optional, but if its defined it must be the last keyword on its line; if anything other than a comment follows on the same line with THEN, BASIC thinks it's reading a single-line IF/THEN/ELSE construct.

IF blocks may be nested.

Example:

```
x=1
IF x=1 THEN
    PRINT "true"
ELSE
    PRINT "false"
ENDIF
...
' Alternate syntax:
x=1
```

```
IF x=1
    PRINT "true"
ELSE
    PRINT "false"
FI
```

Single-line IF.

Syntax:

```
IF expression THEN [num-label]|[command] [ELSE [num-label]|[command]]
```

Causes BASIC to make a decision based on the value of an expression.

expression An expression; 0 is equivalent to FALSE, while all other values are equivalent to TRUE.

command Any legal command or a numeric label. If a number is specified, it is equivalent to a GOTO command with the specified numeric-label.

Example:

```
' Single-line IF
x=1
IF x=1 THEN PRINT "true" ELSE PRINT "false"
...
IF x=1 THEN 1000
...
1000 PRINT "true"
```

IF (*expression, true-value, false-value*) [Function]

Returns a value based on the value of an expression.

Example:

```
x=0
PRINT IF(x<>0,"true","false") : REM prints false
```

END [*error*] [Statement]
STOP [*error*] [Statement]

Terminates execution of a program, closes all files opened by the program, and returns control to the operating system.

error A numeric expression.

The *error* is the value which will returned to operating system; if its not specified the BASIC will return 0.

DOS/Windows *The 'error' value is very well known as ERRORLEVEL value.*

RESTORE *label* [Statement]

Specifies the position of the next data to be read.

label A valid label.

READ *var*[, *var* ...] [Command]

Assigns values in DATA items to specified variables.

var Any variable.

Unless a RESTORE command is executed, BASIC moves to the next DATA item with each READ assignment. If BASIC runs out of DATA items to READ, an run-time error occurs.

Example:

```
FOR c=1 TO 6
    READ x
    PRINT x
NEXT
...
DATA "a,b,c", 2
DATA 3, 4
DATA "fifth", 6
```

DATA *constant1* [,*constant2*]... [Statement]

Stores one or more constants, of any type, for subsequent access via READ command.

DATA commands are nonexecutable statements that supply a stream of data constants for use by READ commands. All the items supplied by all the DATA commands in a program make up one continuous "string" of information that is accessed in order by your program's READ commands.

Example:

```
RESTORE MyDataBlock
FOR I=1 TO 3
    READ v
    PRINT v
NEXT
END
...
LABEL MyDataBlock
DATA 1,2,3
```

ERASE *var*[, *var*[, ... *var*]] [Statement]

var Any variable.

Deallocates the memory used by the specified arrays or variables. After that these variables turned to simple integers with zero value.

Example:

```
DIM x(100)
...
PRINT FRE(0)
ERASE x
PRINT FRE(0)
PRINT x(1):REM ERROR
```

EXIT [*FOR*|*LOOP*|*SUB*|*FUNC*] [Statement]

Exits a multiline function definition, a loop, or a subprogram. By default (if no parameter is specified) exits from last command block (loop, for-loop or routine).

FOR Exit from the last FOR-NEXT loop

LOOP Exit from the last WHILE-WEND or REPEAT-UNTIL loop

SUB Return from the current routine

FUNC Return from the current function

LEN (*x*) [Function]

x Any variable.

If x is a string, returns the length of the string. If x is an array, returns the number of the elements. If x is an number, returns the length of the STR(x).

EMPTY (*x*) [Function]

x Any variable.

If x is a string, returns true if the len(x) is 0. If x is an integer or a real returns true if the x = 0. If x is an array, returns true if x is a zero-length array (array without elements).

ISARRAY (*x*) [Function]

x Any variable.

Returns true if the x is an array.

ISNUMBER (*x*) [Function]

x Any variable.

Returns true if the x is a number (or it can be converted to a number)

Example:

```
? ISNUMBER(12)          :REM true
? ISNUMBER("12")        :REM true
? ISNUMBER("12E+2")     :REM true
? ISNUMBER("abc")       :REM false
? ISNUMBER("1+2")       :REM false
? ISNUMBER("int(2.4)")  :REM false
```

ISSTRING (*x*) [Function]

x Any variable.

Returns true if the x is a string (and cannot be converted to a number)

Example:

```
? ISSTRING(12)      :REM false
? ISSTRING("12")    :REM false
? ISSTRING("12E+2") :REM false
? ISSTRING("abc")   :REM true
? ISSTRING("1+2")   :REM true
```

APPEND *a, val* [, *val* [, ...]] [Command]

a An array-variable.

val Any value or expression

Inserts the values at the end of the specified array.

INSERT *a, idx, val* [, *val* [, ...]]] [Command]

a An array-variable.

idx Position in the array.

val Any value or expression.

Inserts the values to the specified array at the position idx.

DELETE *a, idx* [, *count*] [Command]

a An array-variable.

idx Position in the array.

count The number of the elements to be deleted.

Deletes 'count' elements at position 'idx' of array A

5 System

FRE (*x*) [Function]

Returns system information

Where x:

QB-standard:

0	free memory
-1	largest block of integers
-2	free stack
-3	largest free block

Our standard (it is optional for now):

-10	total physical memory
-11	used physical memory
-12	free physical memory

Optional-set #1:

-13	shared memory size
-14	buffers
-15	cached
-16	total virtual memory size
-17	used virtual memory
-18	free virtual memory

Optional-set #2:

-40	battery voltage * 1000
-41	battery percent
-42	critical voltage value (*1000)
-43	warning voltage value (*1000)

The optional values will returns 0 if are not supported.

RTE [*info* [, ...]] [Command]

Creates a Run-Time-Error. The parameters will be displayed on error-line.

TICKS [Function]

Returns the system-ticks. The tick value is depended on operating system.

TICKSPERSEC [Function]

Returns the number of ticks per second

TIMER [Function]

Returns the number of seconds from midnight

TIME [Function]

Returns the current time as string "HH:MM:SS"

TIMEHMS *hms | timer, BYREF h, BYREF m, BYREF s* [Command]

Converts a time-value to hours, minutes and seconds integer values

DATE [Function]
Returns the current day as string "DD/MM/YYYY"

JULIAN (*dmy* | (*d,m,y*)) [Function]
Returns the Julian date. (dates must be greater than 1/1/100 AD)

Example:

```
PRINT Julian(DATE)
PRINT Julian(31, 12, 2001)
```

DATEDMY *dmy* | *julian_date*, *BYREF d*, *BYREF m*, *BYREF y* [Command]
Returns the day, month and the year as integers.

WEEKDAY (*dmy* | (*d,m,y*) | *julian_date*) [Function]
Returns the day of the week (0 = Sunday)

```
PRINT WeekDay(DATE)
PRINT WeekDay(Julian(31, 12, 2001))
PRINT WeekDay(31, 12, 2001)
```

DATEFMT (*format*, *dmy* | (*d,m,y*) | *julian_date*) [Function]
Returns formated date string

Format:

D	one or two digits of Day
DD	2-digit day
DDD	3-char day name
DDDD	full day name
M	1 or 2 digits of month
MM	2-digit month
MMM	3-char month name
MMMM	full month name
YY	2-digit year (2K)
YYYY	4-digit year

```
PRINT DATEFMT("ddd dd, mm/yy", "23/11/2001")
REM prints "Fri 23, 11/01"
```

DELAY *ms* [Command]
Delay for a specified amount of milliseconds. This 'delay' is also depended to system clock.

SORT *array* [*USE cmpfunc*] [Command]
Sorts an array.

The cmpfunc (if its specified) it takes 2 vars to compare. cmpfunc must returns

-1 if x < y, +1 if x > y, 0 if x = y

```
FUNC qscmp(x,y)
IF x=y
    qscmp=0
ELIF x>y
    qscmp=1
```

```
ELSE
    qscmp=-1
ENDIF
END
...
DIM A(5)
FOR i=0 TO 5
    A(i)=RND
NEXT
SORT A USE qscmp(x,y)
```

SEARCH *A, key, BYREF ridx* [*USE cmpfunc*] [Command]

Scans an array for the key. If key is not found the SEARCH command returns (in ridx) the value (LBOUND(A)-1). In default-base arrays that means -1.

The cmpfunc (if its specified) it takes 2 vars to compare. It must return 0 if x = y; non-zero if x <> y

```
FUNC cmp(x,y)
  cmp=!(x=y)
END
...
DIM A(5)
FOR i=0 TO 5
    A(i)=5-i
NEXT
SEARCH A, 4, r USE cmp(x,y)
PRINT r:REM prints 1
PRINT A(r): REM prints 4
```

CHAIN *file* [Command]

Transfers control to another SmallBASIC program.

file - A string expression that follows OS file naming conventions; The file must be a SmallBASIC source code file.

```
CHAIN "PROG2.BAS"
```

EXEC *file* [Command]

Transfers control to another program

This routine works like CHAIN with the exception the file can be any executable file.

EXEC never returns

ENVIRON "*expr*" [Command]
ENV "*expr*" [Command]

Adds a variable to or deletes a variable from the current environment variable-table.

expr A string expression of the form "name=parameter"

If name already exists in the environment table, its current setting is replaced with the new setting. If name does not exist, the new variable is added.

PalmOS *SB emulates environment variables.*

ENV ("*var*") [Function]
ENVIRON ("*var*") [Function]
Returns the value of a specified entry in the current environment table. If the parameter is empty ("") then returns an array of the envirment variables (in var=value form)

var A string expression of the form "var"

PalmOS *SB emulates environment variables.*

RUN *cmdstr* [Command]
Loads a secondary copy of system's shell and, executes an program, or an shell command.

cmdstr Shell's specific command string

After the specified shell command or program terminates, control is returned to the line following the RUN command.

PalmOS *The 'cmdstr' is the Creator-ID.*
PalmOS *The RUN never returns.*

RUN ("*command*") [Function]
RUN() is the function version of the RUN command. The difference is that, the RUN() returns a string with the output of the 'command' as an array of strings (each text-line is one element).

PalmOS *The RUN() does not supported.*
Windows *The stdout and stderr are separated! First is the stdout output and following the stderr.*

TRON [Command]
TROFF [Command]
TRACE ON/OFF. When trace mechanism is ON, the SB displays each line number as the program is executed

LOGPRINT *...* [Command]
PRINT to SB's logfile. The syntax is the same with the PRINT command.

MALLOC (*size*) [Function]
BALLOC (*size*) [Function]
Allocates a memory block.

***** *The variable can be freed by using ERASE.*

VADR (*var*) [Function]
Returns the memory address of the variable's data.

PEEK[**{16|32}**] (*addr*) [Function]
Returns the byte, word or dword at a specified memory address.

POKE[**{16|32}**] *addr, value* [Command]

Writes a specified byte, word or dword at a specified memory address.

USRCALL *addr* [Command]

Transfers control to an assembly language subroutine.

The USRCALL is equal to:

```
void (*f)(void);
f = (void (*)(void)) addr;
f();
```

BCOPY *src_addr, dst_addr, length* [Command]

Copies a memory block from 'src_addr' to 'dst_addr'

BLOAD *filename*[, *address*] [Command]

Loads a specified memory image file into memory.

BSAVE *filename, address, length* [Command]

Copies a specified portion of memory to a specified file.

STKDUMP [Command]

Displays the SB's internal executor's stack

***** *For debug purposes; it is not supported on "limited" OSes.*

6 Graphics & Sound

The SB's Graphics commands are working only with integers. (Of course, 2D algebra commands are working with reals) That is different of QB, but its much faster.

6.1 The colors

Monochrome
: 0 = black, 15 = white

2bit (4 colors)
: 0 = black, 15 = white, 1-6, 8 = dark-gray, 7, 9-14 = light-gray

4bit (16 colors)
: 16 Standard VGA colors, 16 colors of gray (on PalmOS)

8bit (256 paletted colors)
: 16 Standard VGA colors. The rest colors are ignored.

15bit (32K colors), 16bit (64K colors) and 24bit (1.7M colors)
: Color 0..15 is the standard VGA colors, full 24-bit RGB colors can be passed by using negative number.

6.2 The points

Any point can be specified by an array of 2 elements or by 2 parameters

Example:

```
LINE x1, y1, x2, y2
or
LINE [x1, y1], [x2, y2]
```

Also, the polylines can work with the same way.

```
DIM poly(10)
...
poly[0] = [x, y]
```

6.3 The STEP keyword

The STEP keyword calculates the next x,y parameters relative to current position. That position can be returned by using the POINT(0) and POINT(1) functions.

6.4 The 'aspect' parameter

The x/y factor.

6.5 The FILLED keyword

The FILLED keyword fills the result of the command with the drawing color.

6.6 Graphics Commands

ARC [*STEP*] *x,y,r,astart,aend* [,*aspect* [,*color*]] [*COLOR color*] [Command]
Draws an arc. astart,aend = first,last angle in radians.

CHART *LINECHART|BARCHART, array*() [, *type* [, *x1, y1, x2, y2*]] [Command]
Draws a chart of array values in the rectangular area x1,y1,x2,y2

Where 'type':

0 simple
1 with marks
2 with ruler
3 with marks & ruler

PLOT *xmin, xmax USE f*(*x*) [Command]
Graph of f(x)

Example:

```
PLOT 0, 2*PI USE SIN(x)
```

CIRCLE [*STEP*] *x,y,r* [,*aspect* [, *color*]] [*COLOR color*] [*FILLED*] [Command]

x
y the circle's center

r the radius

Draws a circle (or an ellipse if the aspect is specified).

COLOR *foreground-color* [, *background-color*] [Command]
Specifies the foreground and background colors

DRAWPOLY *array* [,*x-origin,y-origin* [, *scalef* [, *color*]]] [*COLOR color*] [Command]
[*FILLED*]
Draws a polyline

If the array does not uses points as element arrays, then even elements for x (starting from 0), odd elements for y

DRAW *string* [Command]
Draws an object according to instructions specified as a string.

string - A string expression containing commands in the BASIC graphics definition language.

Graphics Definition Language

In the movement instructions below, n specifies a distance to move. The number of pixels moved is equal to n multiplied by the current scaling factor, which is set by the S command.

Un	Move up.
Dn	Move down.
Ln	Move left.
Rn	Move right.

En	Move diagonally up and right.
Fn	Move diagonally down and right.
Gn	Move diagonally down and left.
Hn	Move diagonally up and left.
Mx,y	Move to coordinate x,y. If x is preceded by a + or -, the movement is relative to the last point referenced.
B	A prefix command. Next movement command moves but doesn't plot.
N	A prefix command. Next movement command moves, but returns immediately to previous point.

***** *This command it is had not tested - please report any bug or incompatibility.*

LINE [*STEP*] *x,y* [,|*STEP x2,y2*] [, *color* | *COLOR color*] [Command]
Draws a line

PSET [*STEP*] *x,y* [, *color* | *COLOR color*] [Command]
Draw a pixel

RECT [*STEP*] *x,y* [,|*STEP x2,y2*] [, *color* | *COLOR color*] [*FILLED*] [Command]
Draws a rectangular parallelogram

TXTW (*s*) [Function]
TEXTWIDTH (*s*) [Function]
Returns the text width of string s in pixels

TXTH (*s*) [Function]
TEXTHEIGHT (*s*) [Function]
Returns the text height of string s in pixels

XPOS [Function]
YPOS [Function]
Returns the current position of the cursor in "characters".

POINT (*x* [, *y*]) [Function]
Returns the color of the pixel at x,y

if y does not specified x contains the info-code
0 = returns the current X graphics position
1 = returns the current Y graphics position

PAINT [*STEP*] *x, y* [,*color* [,*border*]] [Command]
Fills an enclosed area on the graphics screen with a specific color.

x *y*	Screen coordinate (column, row) within the area that is to be filled.
color	The fill-color
border	The boundary-color

if the border-color is specified then the PAINT will fill all the area which is specified by the border-color. (fill-until, color!=point(x,y)

if the border-color is NOT specified then the PAINT will fill all the are with the same color as the pixel at x,y. (fill-while, color=point(x,y))

VIEW [*x1,y1,x2,y2* [,*color* [,*border-color*]]] [Command]
Defines a viewport.

x1
y1
x2
y2 Corner coordinates of the viewport.

color If included, BASIC fills the viewport with the specified color.

border-color
If included, BASIC draws a border, in a specified color, around the defined viewport.

The viewport defined by VIEW is disabled by a VIEW command with no parameters.

WINDOW [*x1,y1,x2,y2*] [Command]
Specifies "world" coordinates for the screen.

x1
y1
x2
y2 The corner coordinates of the world space.

The WINDOW command allows you to redefine the corners of the display screen as a pair of "world" coordinates.

The world space defined by WINDOW is disabled by a WINDOW command with no parameters.

RGB (*r, g, b*) [Function]
The RGB functions returns the RGB color codes for the specified values The RGB() takes values 0..255 for each of the color.

The return value is a negative 24bit value to by used by drawing functions.

RGBF (*r, g, b*) [Function]
The RGBF functions returns the RGB color codes for the specified values The RGBF() takes values 0..1 for each of the color.

The return value is a negative 24bit value to by used by drawing functions.

BEEP [Command]
Generates a beep sound

PLAY *string* [Command]
Play musical notes

`A-G[-|+|#][nnn][.]`
Play note A..G, +|# is sharp, - is flat, . is multiplier 1.5

`On`	Octave 0..6, `<` moves down one octave, `>` moves up one octave
`Nnn`	Play note 0..84 (0 = pause)
`Pnnn`	Pause 1..64
`Lnnn`	Length of note 1..64 (1/nnn)
`Tnnn`	Tempo 32..255. Number of 1/4 notes per minute.
`MS`	Staccato (1/2)
`MN`	Normal (3/4)
`ML`	Legato
`Vnnn`	Volume 0..100
`MF`	Play on foreground
`MB`	Play on background
`Q`	Clear sound queue

SOUND *freq, dur_ms* [, *vol*] [*BG*] [Command]

Plays a sound

freq	The frequency
dur_ms	The duration in milliseconds
vol	The volume in 1/100 units
BG	Play it in background

NOSOUND [Command]

Stops background sound. Also, clears the sound queue.

7 Miscellaneous

RANDOMIZE [*int*] [Command]
Seeds the random number generator

RND [Function]
Returns a random number from the range 0 to 1

UBOUND (*array* [, *dim*]) [Function]
Returns the upper bound of the 'array'

LBOUND (*array* [, *dim*]) [Function]
Returns the lower bound of the 'array'

The parameter 'dim' is the array dimension whose bound is returned

```
DIM v1(-4 TO 7)
DIM v2(1 TO 2, 3 TO 4)
...
PRINT LBOUND(v1)   : REM -4
PRINT UBOUND(v1)   : REM 7
...
PRINT LBOUND(v2)   : REM 1
PRINT LBOUND(v2,2) : REM 3
```

CINT (*x*) [Function]
Converts x to 32b integer Meaningless. Used for compatibility.

CREAL (*x*) [Function]
Convert x to 64b real number. Meaningless. Used for compatibility.

CDBL (*x*) [Function]
Convert x to 64b real number. Meaningless. Used for compatibility.

PEN *ON|OFF* [Command]
Enables/Disables the PEN/MOUSE mechanism.

PEN (*0..14*) [Function]
Returns the PEN/MOUSE data.

Values:

0	true (non zero) if there is a new pen or mouse event
1	PEN: last pen down x; MOUSE: last mouse button down x
2	Same as 1 for y
3	true if the PEN is down; MOUSE: mouse left button is pressed
4	PEN: last/current x, MOUSE: the current x position only if the left mouse button is pressed (like PEN is down)
5	Same as PEN(4) for y

Mouse specific (non PalmOS):

10	current mouse x pos
11	current mouse y pos
12	true if the left mouse button is pressed
13	true if the right mouse button is pressed
14	true if the middle mouse button is pressed

***** *The driver must be enabled before use this function (see Pen command)*

PAUSE [*secs*] [Command]
Pauses the execution for a specified length of time, or until user hit the keyboard.

SWAP *a, b* [Command]
Exchanges the values of two variables. The parameters may be variables of any type.

8 File system

8.1 Special Device Names

`"COM1:[speed]"`
: Serial port 1

`"COM2:[speed]"`
: Serial port 2

`"PDOC:filename"`
: Compressed PDOC files for PalmOS or PDB/PDOC files on other systems. PDOCFS opens and uncompress the file on OPEN; and compress the file on CLOSE. So, it will use a lot of memory and time (its depended on size of the data).

`"MEMO:memo-title"`
: MemoDB of PalmOS or regular file on other systems. Memo records (virtual files) are limited to 3935 bytes

`"SOCL:server:port"`
: Socket client. Actually a telnet client.

`"MMC:filename"`
: eBookMan only. Opens an MMC file.

Example: OPEN "COM1:" AS #1
OPEN "COM2:38400" AS #2

8.2 File System Commands

FREEFILE [Function]
Returns an unused file handle

OPEN *file* [*FOR INPUT*|*OUTPUT*|*APPEND*] *AS #fileN* [Command]
Makes a file or device available for sequential input, sequential output.

file A string expression that follows OS file naming conventions.

fileN A file-handle (integer 1 to 256).

FOR -

INPUT	Sequential input
OUTPUT	Sequential output
APPEND	Sequential output, beginning at current EOF

The files are always opened as shared.

CLOSE *#fileN* [Command]
Close a file or device

TLOAD *file, BYREF var* [, *type*] [Command]

Loads a text file into array variable. Each text-line is an array element.

file A string expression that follows OS file naming conventions.

var Any variable

type 0 = load into array (default), 1 = load into string

TSAVE *file, var* [Command]

Writes an array to a text file. Each array element is a text-line.

file A string expression that follows OS file naming conventions.

var An array variable or a string variable. Expressions are not allowed for memory reasons.

EXIST (*file*) [Function]

Returns true if the file exists

file A string expression that follows OS file naming conventions.

ACCESS (*file*) [Function]

Returns the access rights of the file.

file A string expression that follows OS file naming conventions.

The return-value is the permissions of the file as them as specified on GNU's manual (chmod() and stat() system calls)

The bits (in octal):

04000	set user ID on execution
02000	set group ID on execution
01000	sticky bit
00400	read by owner
00200	write by owner
00100	execute/search by owner
00040	read by group
00020	write by group
00010	execute/search by group
00004	read by others
00002	write by others
00001	execute/search by others

PalmOS *The return value is always 0777.*
DOS *The return value is depended on DJGPP's stat() function. Possible Unix compatible.*
Windows *The return value is depended on Cygnus's stat() function. Possible Unix compatible.*

```
IF ACCESS("/bin/sh") AND 0o4 THEN
    PRINT "I can read it!"
ENDIF
```

ISFILE (*file*) [Function]
Returns true if the *file* is a regular file.

ISDIR (*file*) [Function]
Returns true if the *file* is a directory.

ISLINK (*file*) [Function]
Returns true if the *file* is a link.

CHMOD *file, mode* [Command]
Change permissions of a file

file A string expression that follows OS file naming conventions.

mode The mode is compatible with the chmod()'s 'mode' parameter as its described on GNU's manual. See ACCESS() for more information.

```
' Make myfile available to anyone (read/write)
CHMOD "myfile.bas", 0o666
...
' Make myfile available to anyone (execute/read/write)
CHMOD "myfile.bas", 0o777
```

EOF (*fileN*) [Function]
Returns true if the file pointer is at end of the file. For COMx and SOCL VFS it returns true if the connection is broken.

PRINT# *fileN*, [*USING...*] ... [Command]
Write string to a file. The syntax is the same with the PRINT command.

* *We can use 'USG' instead of 'USING'.*

LINPUT# [*fileN*{,|;}] *var* [Command]
LINEINPUT# [*#fileN*{,|;}] *var* [Command]
LINE INPUT# [*fileN*{,|;}] *var* [Command]
Reads a whole text line from file or console.

INPUT (*len* [, *fileN*]) [Function]
This function is similar to INPUT. Reads 'len' bytes from file or console (if fileN is omitted). This function is a low-level function. That means does not convert the data, and does not remove the spaces.

INPUT# *fileN; var1* [,*delim*] [, *var2* [,*delim*]] ... [Command]
Reads data from file

BGETC (*fileN*) [Function]
(Binary mode) Reads and returns a byte from file or device.

BPUTC# *fileN; byte* [Command]
(Binary mode) Writes a byte on file or device

SEEK# *fileN; pos* [Command]
Sets file position for the next read/write

SEEK (*fileN*) [Function]
Returns the current file position

LOF (*fileN*) [Function]
Returns the length of file in bytes. For other devices, it returns the number of available data.

KILL "*file*" [Command]
Deletes the specified file

WRITE# *fileN; var1* [, ...] [Command]
READ# *fileN; var1* [, ...] [Command]
The READ/WRITE command set is used to store variables to a file as binary data.

The common problem with INPUT/PRINT set is there are many conflicts with data.

```
PRINT #1; "Hello, world"
```

You have wrote only one string and you want read it in one variable, but this is impossible for INPUT command to understand it, because INPUT finds the separator comma, so it thinks there are two variables not one.

So, now, you can store arrays, strings etc and what is you write is what you will read the next time.

BTW its faster too.

***** *The parameters can be variables ONLY.*
***** *Its very bad idea to mixed READ/WRITE commands with INPUT/PRINT commands in the same file.*

COPY "*file*", "*newfile*" [Command]
Makes a copy of specified file to the 'newfile'

RENAME "*file*", "*newname*" [Command]
Renames the specified file

MKDIR *dir* [Command]
Create a directory. This does not working on PalmOS.

CHDIR *dir* [Command]
Changes the current working directory. This does not working on PalmOS.

RMDIR *dir* [Command]
Removes a directory. This does not working on PalmOS.

DIRWALK *directory* [, *wildcards*] [*USE ...*] [Command]
Walk through the directories. The user-defined function must returns zero to stop the process.

```
FUNC PRNF(x)
    ? x
    PRNF=TRUE
END
```

```
...
DIRWALK "." USE PRNF(x)
```

PalmOS *Not supported.*

FILES (*wildcards*) [Function]

Returns an array with the filenames. If there is no files returns an empty array.

```
? FILES("*")
```

PalmOS *Returns only the user-files.*
***** *To use file on MEMO or PDOC or any other virtual file system you must use FILES("VFSx:*")*

```
PRINT FILES("MEMO:*")
```

9 Mathematics

All angles are in radians.

ABS (*x*) [Function]
Returns the absolute value of *x*.

MAX (...) [Function]
ABSMAX (...) [Function]
MIN (...) [Function]
ABSMIN (...) [Function]
Maximum/Minimum value of parameters. Parameters can be anything (arrays, ints, reals, strings). ABSMIN/ABSMAX returns the absolute min/max value.

```
? MAX(3,4,8)
? MIN(array(),2,3)
? MAX("abc","def")
```

SEQ (*xmin, xmax, count*) [Function]
Returns an array with 'count' elements. Each element had the x value of its position.

```
? SEQ(0,1,11)
```

EXPRSEQ *BYREF array, xmin, xmax, count USE expression* [Command]
Returns an array with 'count' elements. Each element had the 'y' value of its position as it is returned by the expression.

```
REM same as v=SEQ(0,1,11)
EXPRSEQ v, 0, 1, 11 USE x
```

POW (*x, y*) [Function]
x raised to power of y

SQR (*x*) [Function]
Square root of x

SGN (*x*) [Function]
Sign of x (+1 for positive, -1 for negative and 0 for zero)

9.1 Unit convertion

DEG (*x*) [Function]
Radians to degrees

RAD (*x*) [Function]
Degrees to radians

9.2 Round

INT (*x*) [Function]
Rounds x downwards to the nearest integer

FIX (*x*) [Function]
Rounds x upwards to the nearest integer

FLOOR (*x*) [Function]
Largest integer value not greater than x

CEIL (*x*) [Function]
Smallest integral value not less than x

FRAC (*x*) [Function]
Fractional part of x

ROUND (*x* [, *decs*]) [Function]
Rounds the x to the nearest integer or number with 'decs' decimal digits.

9.3 Trigonometry

COS (*x*) [Function]
Cosine

SIN (*x*) [Function]
Sine

TAN (*x*) [Function]
Tangent

ACOS (*x*) [Function]
Inverse cosine

ASIN (*x*) [Function]
Inverse sine

ATAN (*x*) [Function]
ATN (*x*) [Function]
Inverse tangent

ATAN2 (*x*, *y*) [Function]
Inverse tangent (x,y)

COSH (*x*) [Function]

SINH (*x*) [Function]

TANH (*x*) [Function]

ACOSH (*x*) [Function]

ASINH (*x*) [Function]

ATANH (*x*) [Function]

SEC (*x*) [Function]
Secant

CSC (*x*) [Function]
Cosecant

COT (*x*) [Function]
Cotangent

ASEC (*x*) [Function]
Inverse secant

ACSC (*x*) [Function]
Inverse cosecant

ACOT (*x*) [Function]
Inverse cotangent

SECH (*x*) [Function]

CSCH (*x*) [Function]

COTH (*x*) [Function]

ASECH (*x*) [Function]

ACSCH (*x*) [Function]

ACOTH (*x*) [Function]

9.4 Logarithms

EXP (*x*) [Function]
Returns the value of e raised to the power of *x*.

LOG (*x*) [Function]
Returns the natural logarithm of *x*.

LOG10 (*x*) [Function]
Returns the base-10 logarithm of *x*.

9.5 Statistics

Sample standard deviation: SQR(STATSPREADS(array)) Population standard deviation: SQR(STATSPREADP(array))

SUM (...) [Function]
Sum of value

SUMSQ (...) [Function]
Sum of square value

STATMEAN (...) [Function]
Arithmetical mean

STATMEANDEV (...) [Function]
Mean deviation

STATSPREADS (...) [Function]
Sample spread

STATSPREADP (...) [Function]
Population spread

9.6 Equations

LINEQN (*a*, *b* [, *toler*]) [Function]
Returns an array with the values of the unknowns. This function solves equations by using the Gauss-Jordan method.

b	equations
b	results
toler	tolerance number. (the absolute value of the lowest acceptable number) default = 0 = none... |x| <= toler : x = 0

***** *The result is a matrix Nx1. For the SB that array is two-dimension array.*

INVERSE (*A*) [Function]
returns the inverse matrix of *A*.

DETERM (*A*[, *toler*]) [Function]
Determinant of A

toler = tolerance number (the absolute value of the lowest acceptable number) default = 0 = none

|x| <= toler : x = 0

ROOT *low*, *high*, *segs*, *maxerr*, *BYREF result*, *BYREF errcode USE expr* [Command]
Roots of F(x)

low	the lower limit
high	the upper limit
segs	the number of segments (spaces)
maxerr	tolerance (IF ABS(F(x)) < maxerr THEN OK)

errcode 0 for success; otherwise calculation error

result the result

```
FUNC F(x)
 F = SIN(x)
END
...
ROOT 1, 5, 500, 0.00001, result, errcode USE F(x)
```

DERIV *x, maxtries, maxerr, BYREF result, BYREF errcode USE expr* [Command]

Calculation of derivative

x value of x

maxtries maximum number of retries

maxerr tolerance

errcode 0 for success; otherwise calculation error

result the result

DIFFEQN *x0, y0, xf, maxseg, maxerr, BYREF yf, BYREF errcode USE expr* [Command]

Differential equation - Runge-Kutta method

x0
y0 initial x,y

xf x final

maxseg maximum number of segments on x

maxerr tolerance (acceptable error between the last 2 times)

errcode 0 for success; otherwise calculation error

yf the result

10 2D Algebra

SEGCOS (*Ax,Ay,Bx,By,Cx,Cy,Dx,Dy*) [Function]
SEGSIN (*Ax,Ay,Bx,By,Cx,Cy,Dx,Dy*) [Function]
Sinus or cosine of 2 line segments (A->B, C->D).

PTDISTSEG (*Bx,By,Cx,Cy,Ax,Ay*) [Function]
Distance of point A from line segment B-C

PTDISTLN (*Bx,By,Cx,Cy,Ax,Ay*) [Function]
Distance of point A from line B, C

PTSIGN (*Ax,Ay,Bx,By,Qx,Qy*) [Function]
The sign of point Q from line segment A->B

SEGLEN (*Ax,Ay,Bx,By*) [Function]
Length of line segment

POLYAREA (*poly*) [Function]
Returns the area of the polyline *poly*.

POLYEXT *poly*(), *BYREF xmin, BYREF ymin, BYREF xmax, BYREF ymax* [Command]
Returns the polyline's extents

INTERSECT *Ax, Ay, Bx, By, Cx, Cy, Dx, Dy, BYREF type, BYREF Rx, BYREF Ry* [Command]
Calculates the intersection of the two line segments A-B and C-D

Returns: Rx,Ry = cross

type = cross-type

0	No cross (R = external cross)
1	One cross
2	Parallel
3	Parallel (many crosses)
4	The cross is one of the line segments edges.

10.1 2D & 3D graphics transformations

2D & 3D graphics transformations can represented as matrices.
(c=cos, s=sin)

M3IDENT *BYREF m3x3* [Command]
Resets matrix (Identity)

```
|  1  0  0 |
|  0  1  0 |
|  0  0  1 |
```

M3ROTATE *BYREF m3x3, angle* [*, x, y*] [Command]

Rotate by angle with center x,y

```
|  c  s  0 |
| -s  c  0 |
|  *  *  1 |
```

M3SCALE *BYREF m3x3, x, y, Sx, Sy* [Command]

Scaling

```
| Sx  0  0 |
|  0 Sy  0 |
|  *  *  1 |
```

M3TRANS *BYREF m3x3, Tx, Ty* [Command]

Translation

```
|  1  0  0 |
|  0  1  0 |
| Tx Ty  1 |
```

M3APPLY *m3x3, BYREF poly* [Command]

Apply matrice to poly-line

Additional information:

```
|  1  0  0 |
|  0 -1  0 | = reflection on x
|  0  0  1 |

| -1  0  0 |
|  0  1  0 | = reflection on y
|  0  0  1 |
```

3D-Graphics Matrices:

```
|  1  0  0 Tx |
|  0  1  0 Ty | = translation
|  0  0  1 Tz |
|  0  0  0  1 |

| Sx  0  0  0 |
|  0 Sy  0  0 | = scaling
|  0  0 Sz  0 |
|  0  0  0  1 |

|  1  0  0  0 |
|  0  c -s  0 | = rotation on x
|  0  s  c  0 |
|  0  0  0  1 |

|  c  0  s  0 |
|  0  1  0  0 | = rotation on y
```

```
| -s  0  c  0 |
|  0  0  0  1 |

|  c -s  0  0 |
|  s  c  0  0 | = rotation on z
|  0  0  1  0 |
|  0  0  0  1 |
```

Any change to matrix will combined with its previous value.

```
DIM poly(24)
DIM M(2,2)
...
M3IDENT M
M3ROTATE M, pi/2, 0, 0
M3SCALE M, 0, 0, 1.24, 1.24
...
' Draw the original polyline
DRAWPOLY poly
...
' Draw the polyline
' rotated by pi/2 from 0,0 and scaled by 1.24
M3APPLY M, poly
DRAWPOLY poly
```

11 Strings

SPC (*n*) [Function]
SPACE (*n*) [Function]
returns a string of 'n' spaces

BIN (*x*) [Function]
Returns the binary value of x as string.

OCT (*x*) [Function]
Returns the octal value of x as string.

HEX (*x*) [Function]
Returns the hexadecimal value of x as string.

VAL (*s*) [Function]
Returns the numeric value of string *s*.

STR (*x*) [Function]
Returns the string value of *x*.

CBS (*s*) [Function]
BCS (*s*) [Function]
CBS() - converts (C)-style strings to (B)ASIC-style (S)trings

BCS() - converts (B)ASIC-style strings to (C)-style (S)trings

C-Style string means strings with \ codes

***** *On CBS() we cannot use the "" character but we can replace it with "x22 or "042.*

ASC (*s*) [Function]
Returns the ASCII code of first character of the string *s*.

CHR (*x*) [Function]
Returns one-char string of character with ASCII code x.

LOWER (*s*) [Function]
LCASE (*s*) [Function]
UPPER (*s*) [Function]
UCASE (*s*) [Function]
Converts the string s to lower/upper case

```
? LOWER("Hi"):REM hi
? UPPER("Hi"):REM HI
```

LTRIM (*s*) [Function]
Removes leading white-spaces from string s

```
? LEN(LTRIM("  Hi")):REM 2
```

RTRIM (*s*) [Function]
Removes trailing white-spaces from string s

TRIM (*s*) [Function]

Removes leading and trailing white-spaces from string *s*. `TRIM` is equal to `LTRIM(RTRIM(s))`

SQUEEZE (*s*) [Function]

Removes the leading/trailing and duplicated white-spaces

```
? "["; SQUEEZE(" Hi  there "); "]"
' Result: [Hi there]
```

ENCLOSE (*str*[, *pair*]) [Function]

Encloses a string. The default pair is ""

```
? enclose("abc", "()")
' Result: (abc)
```

DISCLOSE (*str*[, *pairs* [, *ignore-pairs*]]) [Function]

Discloses a string.

Default pairs and ignore pairs

```
First
non white-space
character           Check   Ignore
-------------------------------------------
"                   ""      ''
'                   ''      ""
(                   ()      ""''
[                   []      ""''
{                   {}      ""''
<                   <>      ""''


Otherwise:
"                   ""      ''
```

```
s = "abc (abc)"
? s; tab(26); disclose(s, "()")
' prints abc
s = "abc (a(bc))"
? s; tab(26); disclose(s, "()"); tab(40); disclose(disclose(s, "()"), "()")
' prints a(bc), bc
s = "abc (a='(bc)')"
? s; tab(26); disclose(s, "()", "''"); tab(40); &
    disclose(disclose(s, "()", "''"), "()", "''")
' prints a='(bc)', nothing
```

LEFT (*s* [,*n*]) [Function]
RIGHT (*s*[,*n*]) [Function]

Returns the n number of leftmost/rightmost chars of string s If n is not specified, the SB uses 1

LEFTOF (*s1, s2*) [Function]
RIGHTOF (*s1, s2*) [Function]

Returns the left/right part of s1 at the position of the first occurrence of the string s2 into string s1

***** *s2 does not included on new string.*

LEFTOFLAST (*s1, s2*) [Function]
RIGHTOFLAST (*s1, s2*) [Function]

Returns the left/right part of s1 at the position of the last occurrence of the string s2 into string s1

***** *s2 does not included on new string.*

MID (*s, start* [,*length*]) [Function]

Returns the part (length) of the string s starting from 'start' position

If the 'length' parameter is omitted, MID returns the whole string from the position 'start'.

INSTR ([*start,*] *s1, s2*) [Function]

Returns the position of the first occurrence of the string s2 into string s1 (starting from the position 'start')

If there is no match, INSTR returns 0

RINSTR ([*start,*] *s1, s2*) [Function]

Returns the position of the last occurrence of the string s2 into string s1 (starting from the position 'start')

If there is no match, RINSTR returns 0

REPLACE (*source, pos, str* [, *len*]) [Function]

Writes the 'str' into 'pos' of 'source' and returns the new string.

This function replaces only 'len' characters. The default value of 'len' is the length of 'str'.

```
s="123456"
...
' Cut
? replace(s,3,"",len(s))
...
' Replace
? replace(s,2,"bcd")
...
' Insert
? replace(s,3,"cde",0)
...
' Replace & insert
? replace(s,2,"RRI",2)
```

TRANSLATE (*source, what* [*, with*]) [Function]

Translates all occurrences of the string 'what' found in the 'source' with the string 'with' and returns the new string.

```
? Translate("Hello world", "o", "0")
' displays: Hell0 w0rld
```

CHOP (*source*) [Function]

Chops off the last character of the string 'source' and returns the result.

STRING (*len,* {*ascii*|*str*}) [Function]

Returns a string containing 'len' times of string 'str' or the character 'ascii'.

FORMAT (*format, val*) [Function]

Returns a formated string.

Numbers:

`#`	Digit or space
`0`	Digit or zero
`^`	Stores a number in exponential format. Unlike QB's USING format this is a place-holder like the #.
`.`	The position of the decimal point.
`,`	Separator.
`-`	Stores minus if the number is negative.
`+`	Stores the sign of the number.

Strings:

`&`	Stores a string expression without reformatting it.
`!`	Stores only the first character of a string expression.
`\ \`	Stores only the first n + 2 characters of a string expression, where n is the number of spaces between the two backslashes. Unlike QB, there can be literals inside the \ \. These literals are inserted in the final string.

```
? FORMAT("#,##0", 1920.6) : REM prints 1,921
? FORMAT("\  - \", "abcde") : REM prints "abc-de"
```

SPRINT *var;* [*USING...;*] ... [Command]

Create formated string and storing it to var The syntax is the same with the PRINT command.

```
SPRINT s; 12.34; TAB(12); 11.23;
```

***** *You can use 'USG' instead of 'USING'.*

SINPUT *src; var* [*, delim*] [*,var* [*, delim*]] ... [Command]

Splits the string 'src' into variables which are separated by delimiters.

```
SINPUT "if x>1 then y"; vif, " ", vcond, "then", vdo
? vcond, vdo
' result in monitor
' x>1   y
```

SPLIT *string, delimiters, words*() [, *pairs*] [*USE expr*] [Command]

Returns the words of the specified string into array 'words'

Example:

```
s="/etc/temp/filename.ext"
SPLIT s, "/.", v()
FOR i=0 TO UBOUND(v)
  PRINT i;" [";v(i);"]"
NEXT
'
displays:
0 []
1 [etc]
2 [temp]
3 [filename]
4 [ext]
```

JOIN *words*(), *delimiters, string* [Command]

Returns the words of the specified string into array 'words'

Example:

```
s="/etc/temp/filename.ext"
SPLIT s, "/.", v()
JOIN v(), "/", s
PRINT "[";s;"]"
'
displays:
[/etc/temp/filename/ext]
```

12 Console

12.1 Supported console codes

* *“e = CHR(27)*

\t	tab (32 pixels)
\a	beep
\r\n	new line (cr/lf)
\xC	clear screen
\e[K	clear to EOL
\e[nG	moves cursor to specified column
\e[0m	reset all attributes to their defaults
\e[1m	set bold on
\e[4m	set underline on
\e[7m	reverse video
\e[21m	set bold off
\e[24m	set underline off
\e[27m	set reverse off
\e[3nm	set foreground color. where n: 0 black 1 red 2 green 3 brown 4 blue 5 magenta 6 cyan 7 white
\e[4nm	set background color. (see set foreground)

PalmOS only:

\e[8nm	(n=0..7) select system font
\e[9nm	(n=0..3) select buildin font

eBookMan only:

\e[50m	select 9pt font
\e[51m	select 12pt font
\e[52m	select 16pt font
\e[nT	move to n/80th screen character position

12.2 Console Commands

PRINT [*USING* [*format*;]] [*expr*|*str* [{,|;} [*expr*|*str*]] ... [Command]

Displays a text or the value of an expression.

PRINT SEPARATORS

TAB(n)	Moves cursor position to the nth column.
SPC(n)	Prints a number of spaces specified by n.
;	Carriage return/line feed suppressed after printing.
,	Carriage return/line feed suppressed after printing. A TAB character is placed.

The PRINT USING

Print USING, is using the FORMAT() to display numbers and strings. Unlike the FORMAT, this one can include literals, too.

_	Print next character as a literal. The combination _#, for example, allows you to include a number sign as a literal in your numeric format.
[other]	Characters other than the foregoing may be included as literals in the format string.

***** *When a PRINT USING command is executed the format will remains on the memory until a new format is passed. Calling a PRINT USING without a new format specified the PRINT will use the format of previous call.*

Examples:

```
PRINT USING "##: #,###,##0.00";
FOR i=0 TO 20
    PRINT USING; i+1, A(i)
NEXT
....
PRINT USING "Total ###,##0 of \ \"; number, "bytes"
```

***** *The symbol ? can be used instead of keyword PRINT You can use 'USG' instead of 'USING'.*

CAT (*x*) [Function]

Returns a console codes

0	reset
1	bold on
-1	bold off
2	underline on
-2	underline off
3	reverse on
-3	reverse off

PalmOS only:

80..87	select system font
90..93	select custom font

Example:

```
? cat(1);"Bold";cat(0)
```

INPUT [*prompt* {,|;}] *var*[, *var* [, ...]] [Command]
Reads from "keyboard" a text and store it to variable.

LINPUT *var* [Command]
LINEINPUT *var* [Command]
LINE INPUT *var* [Command]
Reads a whole text line from console.

INKEY [Function]
This function returns the last key-code in keyboard buffer, or an empty string if there are no keys.

Special key-codes like the function-keys (PC) or the hardware-buttons (PalmOS) are returned as 2-byte string.

Example:

```
k=INKEY
IF LEN(k)
  IF LEN(k)=2
    ? "H/W #"+ASC(RIGHT(k,1))
  ELSE
    ? k; " "; ASC(k)
  FI
ELSE
  ? "keyboard buffer is empty"
FI
```

CLS [Command]
Clears the screen.

AT *x, y* [Command]
Moves the console cursor to the specified position. x,y are in pixels

LOCATE *y, x* [Command]
Moves the console cursor to the specified position. x,y are in character cells.

Appendix A Interactive Mode

Like a shell, SB can run interactively. The *Interactive Mode* offers an old-style coding taste. Also, it is offers a quick editing/testing tool for console mode versions of SB.

The *Interactive Mode* can be used as a normal command-line *shell*. It executes shell commands as a normal shell, but also, it can store/edit and run SB programs. However we *suggest to use an editor*.

- We can use the [TAB] for autocompletion (re-edit program lines or filename completition).
- We can use [ARROWS] for history.
- There is no need to type line numbers, there will be inserted automagically if you use '+' in the beginning of the line.
- There is no need to type line numbers, use `NUM`.
- Line numbers are not labels, are used only for editing. Use keyword `LABEL` to define a label.
- Line numbers are not saved in files.

A.1 Interactive Mode Commands

HELP [*sb-keyword*] [Command]
Interactive mode help screen. The symbol '?' does the same.

BYE [Command]
QUIT [Command]
EXIT [Command]
The BYE command ends SmallBASIC and returns the control to the Operating System.

NEW [Command]
The NEW command clears the memory and screen and prepares the computer for a new program. Be sure to save the program that you have been working on before you enter NEW as it is unrecoverable by any means once NEW has been entered.

RUN [*filename*] [Command]
The RUN command, which can also be used as a statement, starts program execution.

CLS [Command]
Clears the screen.

LIST `{` [*start-line*] - [*end-line*] `}` [Command]
The LIST command allows you to display program lines. If LIST is entered with no numbers following it, the entire program in memory is listed. If a number follows the LIST, the line with that number is listed. If a number followed by hyphen follows LIST, that line and all lines following it are listed. If a number preceeded by a hyphen follows LIST, all lines preceeding it and that line are listed. If two numbers separated by a hyphen follow LIST, the indicated lines and all lines between them are listed.

RENUM { [*initial-line*] [,] [*increment*] } [Command]
The RENUM command allows you to reassign line numbers.

ERA { [*start-line*] - [*end-line*] } [Command]
The ERA command allows you to erase program lines. If ERA is entered with no numbers following it, the entire program in memory is erased. If a number follows the ERA, the line with that number is erased. If a number followed by hyphen follows ERA, that line and all lines following it are erased. If a number preceeded by a hyphen follows ERA, all lines preceeding it and that line are erased. If two numbers separated by a hyphen follow ERA, the indicated lines and all lines between them are erased.

NUM [*initial-line* [, *increment*]] [Command]
The NUM command sets the values for the autonumbering. If the 'initial-line' and 'increment' are not specified, the line numbers start at 10 and increase in increments of 10.

SAVE *program-name* [Command]
The SAVE command allows you to copy the program in memory to a file. By using the LOAD command, you can later recall the program into memory.

LOAD *program-name* [Command]
The LOAD command loads 'program-name' file into memory. The program must first have been put on file using the SAVE command. LOAD removes the program currently in memory before loading 'program-name'.

MERGE *program-name, line-number* [Command]
The MERGE command merges lines in 'program-name' file into the program lines already in the computer's memory. Use 'line-number' to specify the position where the lines will be inserted.

CD [*path*] [Command]
Changed the current directory. Without arguments, displays the current directory.

DIR [*regexp*] [Command]
DIRE [*regexp*] [Command]
DIRD [*regexp*] [Command]
DIRB [*regexp*] [Command]
Displays the list of files. You can use DIRE for executables only or DIRD for directories only, or DIRB for BASIC sources.

TYPE *filename* [Command]
Displays the contents of the file.

Appendix B MySQL Module

MYSQL.CONNECT (*host, database, user, [password]*) [Function]
Connects/reconnects to the server

MYSQL.QUERY (*handle, sqlstr*) [Function]
Send command to mysql server

MYSQL.DBS (*handle*) [Function]
Get a list of the databases

MYSQL.TABLES (*handle*) [Function]
Get a list of the tables

MYSQL.FIELDS (*handle, table*) [Function]
Get a list of the fields of a table

MYSQL.DISCONNECT *handle* [Command]
Disconnects

MYSQL.USE *handle, database* [Command]
Changes the current database

Example:

```
import mysql

h = mysql.connect("localhost", "mydatabase", "user", "password")
? "Handle = "; h
? "DBS    = "; mysql.dbs(h)
? "TABLES = "; mysql.tables(h)
? "Query  = "; mysql.query(h, "SELECT * FROM sbx_counters")
mysql.disconnect h
```

Appendix C GDBM Module

Example:

```
import gdbm

const GDBM_WRCREAT = 2 ' A writer.  Create the db if needed.

' TEST
h = gdbm.open("dbtest.db", 512, GDBM_WRCREAT, 0o666)
? "Handle = "; h
? "Store returns = "; gdbm.store(h, "key1", "data1....")
? "Store returns = "; gdbm.store(h, "key2", "data2....")
? "Fetch returns = "; gdbm.fetch(h, "key1")
gdbm.close h
```

Appendix D Limits

D.1 Typical 32bit system

Bytecode size	4 GB
Length of text lines	4095 characters
User-defined keyword length	128 characters
Maximum number of parameters	256
Numeric value range	64 bit FPN (-/+ 1E+308)
Maximum string size	2 GB
Number of file handles	256
Number of array-dimensions	6
Number of colors	24 bit (0-15=VGA, <0=RGB)
Background sound queue size	256 notes
INPUT (console)	1023 characters per call, up to 16 variables
COMMAND$	1023 bytes

System events are checked every 50ms

D.2 PalmOS (Typical 16bit system)

Length of text lines	511 characters
Maximum number of parameters	32
User-defined keyword length	32 characters
Number of array-dimensions	3
Maximum string size	<32 KB
Number of file handles	16
Number of elements/array	2970 (that means 64KB of memory)
Bytecode size	<64 KB (by using CHAIN you can run progs > 64KB)
INPUT (console)	255 characters per call, up to 16 variables
COMMAND$	127 bytes

Appendix E Writting Modules

* Modules are working only at Linux for now *

Modules are dynamic-linked libraries. The modules are "connected" with the SmallBASIC with a two-way style. That means, the module can execute functions of SB's library.

Module programmers will need to use variable's API to process parameters, and return values. Also, the device's API must be used because SB can run in different environments, of course module authors can use other C or other-lib functions to do their jobs.

Every module must implements the following C functions.

int sblib_proc_count()
: Returns the number of procedures of the module.

int sblib_func_count()
: Returns the number of functions of the module.

int sblib_proc_getname(int index, char *name)
: Fills the 'name' variable with the name of the 'index'-th procedure. Returns 1 on success or 0 on error.

int sblib_func_getname(int index, char *name)
: Fills the 'name' variable with the name of the 'index'-th function. Returns 1 on success or 0 on error.

int sblib_proc_exec(int index, int param_count, slib_par_t *params, var_t *retval)
: Executes the 'index' procedure. Returns 1 on success or 0 on error.

int sblib_func_exec(int index, int param_count, slib_par_t *params, var_t *retval)
: Executes the 'index' function. Returns 1 on success or 0 on error.

The slib_par_t structure contains two fields. The *var_p* which is a var_t structure (a SB variable), and the *byref* which is true if the variable can be used as by-reference.

E.1 Variables API

Variables had 4 types. This type is described in .type field.

Values of .type

V_STR String. The value can be accessed at *.v.p.ptr*.

V_INT Integer. The value can be accessed at *.v.i*.

V_REAL Real-number. The value can be accessed at *.v.n*.

V_ARRAY
: Array. The .v.a.ptr is the data pointer (sizeof(var_t) * size). The .v.a.size is the number of elements. The .v.a.lbound[MAXDIM] is the lower bound values. The .v.a.ubound[MAXDIM] is the upper bound values. The .v.a.maxdim is the number of dimensions.

Example:

```
/*
*   Displays variable data.
*   If the variable is an array, then this function
*   runs recursive, and the 'level' parameter is used.
*/
static void print_variable(int level, var_t *variable)
{
    int i;

    /* if recursive; place tabs */
    for ( i = 0; i < level; i ++ )
        dev_printf("\t");

    /* print variable */
    switch ( variable->type )  {
    case V_STR:
        dev_printf("String =\"%s\"\n", variable->v.p.ptr);
        break;
    case V_INT:
        dev_printf("Integer = %ld\n", variable->v.i);
        break;
    case V_REAL:
        dev_printf("Real = %.2f\n",  variable->v.n);
        break;
    case V_ARRAY:
        dev_printf("Array of %d elements\n", variable->v.a.size);
        for ( i = 0; i < variable->v.a.size; i ++ ) {
            var_t *element_p;

            element_p = (var_t *) (variable->v.a.ptr + sizeof(var_t) * i);
            print_variable(level+1, element_p);
            }
        break;
        }
}
```

E.1.1 Gereric

void v_free(var_t *var)
: This function resets the variable to 0 integer.

void v_free(var_t *var)
: This function deletes the contents of varible *var*.

var_t* v_new()
: Creates a new variable and returns it. The returned variable it must be freed with both, v_free() and free() functions.

var_t *v_clone(const var_t *source)
Returns a new variable which is a clone of *source*. The returned variable it must be freed with both, v_free() and free() functions.

void v_set(var_t *dest, const var_t *src)
Copies the *src* to *dest*.

Example

```
void    myfunc()
{
        var_t   myvar;

        v_init(&myvar);
        ...
        v_free(&myvar);
}
```

E.1.2 Real Numbers

double v_getreal(var_t *variable)
Returns the floating-point value of a variable. if *variable* is string it will converted to double.

void v_setreal(var_t *var, double number)
Sets the *number* real-number value to *var* variable.

E.1.3 Integer Numbers

double v_igetnum(var_t *variable)
Returns the floating-point value of a variable. if *variable* is string it will converted to double.

void v_setint(var_t *var, int32 number)
Sets the *number* integer-number value to *var* variable.

E.1.4 Strings

void v_tostr(var_t *arg)
Converts variable *arg* to string.

char* v_getstr(var_t *var)
Returns the string-pointer of variable *var*. If the *var* is not a string, it must be converted to string with the v_tostr() function.

void v_zerostr(var_t *var)
Resets the variable *var* to a zero-length string.

void v_setstr(var_t *var, const char *string)
Sets the string value *string* to the variable *var*.

void v_strcat(var_t *var, const char *string)
Adds the string *string* to string-variable *var*.

void v_setstrf(var_t *var, const char *fmt, ...)
Sets a string value to variable *var* using printf() style. The buffer size is limited to 1KB for OS_LIMITED (PalmOS), otherwise 64kB.

E.1.5 Arrays

SB arrays are always one-dimension. The multiple dimensions positions are calculated at run-time. Each element of the arrays is a '`var_t`' object.

var_t* v_elem(var_t *array, int index)
Returns the variable pointer of the element *index* of the *array*. *index* is a zero-based, one dimention, index.

int v_asize(var_t *array)
Returns the number of the elements of the *array*.

void v_resize_array(var_t *array, int size)
Resizes the 1-dimention *array*.

void v_tomatrix(var_t *var, int r, int c)
Converts the variable *var* to an array of *r* rows and *c* columns.

void v_toarray1(var_t *var, int n)
Converts the variable *var* to an array of *n* elements.

void v_setintarray(var_t *var, int32 *itable, int count)
Makes variable *var* an integer array of *count* elements. The values are specified in *itable*.

void v_setrealarray(var_t *var, double *rtable, int count)
Makes variable *var* a real-number array of *count* elements. The values are specified in *rtable*.

void v_setstrarray(var_t *var, char **ctable, int count)
Makes variable *var* a string array of *count* elements. The values (which are copied) are specified in *ctable*.

Example

```
void    myfunc()
{
        int             c_array[] = { 10, 20, 30 };
        var_t   myvar;

        v_init(&myvar);
        v_setintarray(&myvar, c_array, 3);
        v_free(&myvar);
}
```

E.2 Typical Module Source

This is a typical example of a module with one Function and and one Command. The command "CMDA" displays its parameters, and the function "FUNCA" returns a string.

— mymod.c —

```
#include <extlib.h>

/*
*   Displays variable data.
*   If the variable is an array, then runs recursive, the
*   'level' parameter is the call level.
*/
static void print_variable(int level, var_t *param)
{
    int i;

    /* if recursive; place tabs */
    for ( i = 0; i < level; i ++ )
        dev_printf("\t");

    /* print variable */
    switch ( param->type )  {
    case V_STR:
        dev_printf("String =\"%s\"\n", param->v.p.ptr);
        break;
    case V_INT:
        dev_printf("Integer = %ld\n", param->v.i);
        break;
    case V_REAL:
        dev_printf("Real = %.2f\n",  param->v.n);
        break;
    case V_ARRAY:
        dev_printf("Array of %d elements\n", param->v.a.size);
        for ( i = 0; i < param->v.a.size; i ++ ) {
            var_t *element_p;

            element_p = (var_t *) (param->v.a.ptr + sizeof(var_t) * i);
            print_variable(level+1, element_p);
            }
        break;
        }
}

/* typical command */
void m_cmdA(int param_count, slib_par_t *params, var_t *retval)
{
```

```
    int i;

    for ( i = 0; i < param_count; i ++ ) {
        param = params[i].var_p;
        print_variable(0, param);
        }
}

/* typical function */
int m_funcA(int param_count, slib_par_t *params, var_t *retval)
{
    v_setstr(retval, "funcA() works!");
    return 1;        /* success */
}

/* the node-type of function/procedure tables */
typedef struct {
        char *name;  /* the name of the function */
        int  (*command)(slib_par_t *, int, var_t *);
        } mod_kw;

/* functions table */
static mod_kw func_names[] =
{
{ "FUNCA", m_funcA }, // function A
{ NULL, NULL }
};

/* commands table */
static mod_kw proc_names[] =
{
{ "CMDA", m_cmdA }, // command A
{ NULL, NULL }
};

/* returns the number of the procedures */
int sblib_proc_count(void)
{
    int i;

    for ( i = 0; proc_names[i].name; i ++ );
    return i;
}

/* returns the number of the functions */
int sblib_func_count(void)
{
```

```
    int i;

    for ( i = 0; func_names[i].name; i ++ );
    return i;
}

/* returns the 'index' procedure name */
int sblib_proc_getname(int index, char *proc_name)
{
    strcpy(proc_name, proc_names[index].name);
    return 1;
}

/* returns the 'index' function name */
int sblib_func_getname(int index, char *proc_name)
{
    strcpy(proc_name, func_names[index].name);
    return 1;
}

/* execute the 'index' procedure */
int sblib_proc_exec(int index, int param_count,
                                slib_par_t *params, var_t *retval)
{ return proc_names[index].command(params, param_count, retval); }

/* execute the 'index' function */
int sblib_func_exec(int index, int param_count,
                                slib_par_t *params, var_t *retval)
{ return func_names[index].command(params, param_count, retval); }
```

E.3 Typical Module Makefile

This is a typical Makefile. In our example the module name is 'mymod' and its source is the 'mymod.c'. Also, our module is requires the 'mysqlclient' library to be linked together.

The variables of Makefile

MODNAME
: The name of the module

MODLIBS
: The libraries that are required by the module.

MODIDIR
: The SB's module directory. There will be installed the module.

CINC
: 'Include' path. This must points to the SB source files.

CFLAGS
: Compilers flags.

— Makefile —

```
MODNAME=mymod
MODLIBS=-lmysqlclient
MODIDIR=/usr/lib/sbasic/modules
CINC=-I/opt/sbasic/source
CFLAGS=-Wall -fPIC $(CINC) -D_UnixOS -DLNX_EXTLIB

all: $(MODIDIR)/$(MODNAME).so

$(MODIDIR)/$(MODNAME).so: $(MODNAME).c
    -mkdir -p $(MODIDIR)
    gcc $(CFLAGS) -c $(MODNAME).c -o $(MODNAME).o
    gcc -shared -Wl,-soname,$(MODNAME).so -o $(MODNAME).so $(MODNAME).o $(MODLIBS)
    mv $(MODNAME).so $(MODIDIR)
    ldconfig -n $(MODIDIR)

clean:
    -rm -f *.so *.o $(MODIDIR)/$(MODNAME).so
```

Appendix F Glossary

What it could be good to know.

ANSI The American National Standards Institute. This organization produces many standards, among them the standards for the C and C++ programming languages. See also "ISO".

Program An program consists of a series of *commands*, *statements*, and *expressions*. The program executed by an interpreted language command by command until it ends.

Script Another name for an program. ...

Bit Short for "Binary Digit". All values in computer memory ultimately reduce to binary digits: values that are either zero or one.

Computers are often defined by how many bits they use to represent integer values. Typical systems are 32-bit systems, but 64-bit systems are becoming increasingly popular, and 16-bit systems are waning in popularity.

Character Set
The set of numeric codes used by a computer system to represent the characters (letters, numbers, punctuation, etc.) of a particular country or place. The most common character set in use today is ASCII (American Standard Code for Information Interchange). Many European countries use an extension of ASCII known as ISO-8859-1 (ISO Latin-1).

Compiler A program that translates human-readable source code into machine-executable object code. The object code is then executed directly by the computer or by a virtual-machine. See also "Interpreter".

Deadlock The situation in which two communicating processes are each waiting for the other to perform an action.

Environment Variables
A collection of strings, of the form *name*=*val*, that each program has available to it. Users generally place values into the environment in order to provide information to various programs. Typical examples are the environment variables `HOME` and `PATH`.

Escape Sequences
A special sequence of characters used for describing nonprinting characters, such as '`\n`' for newline or '`\033`' for the ASCII ESC (Escape) character.

Flag A variable whose truth value indicates the existence or nonexistence of some condition.

Free Software Foundation
FSF A nonprofit organization dedicated to the production and distribution of freely distributable software. It was founded by Richard M. Stallman.

GNU General Public License
GNU GPL
This document describes the terms under which binary library archives or shared objects, and their source code may be distributed.

With few words, GPL allows source code and binary forms to be used copied and modified freely.

GMT "Greenwich Mean Time". It is the time of day used as the epoch for Unix and POSIX systems.

GNU "GNU's not Unix". An on-going project of the Free Software Foundation to create a complete, freely distributable, POSIX-compliant computing environment.

GNU/Linux
A variant of the GNU system using the Linux kernel, instead of the Free Software Foundation's Hurd kernel. Linux is a stable, efficient, full-featured clone of Unix that has been ported to a variety of architectures. It is most popular on PC-class systems, but runs well on a variety of other systems too. The Linux kernel source code is available under the terms of the GNU General Public License, which is perhaps its most important aspect.

Hexadecimal
Base 16 notation, where the digits are `0`–`9` and `A`–`F`, with '`A`' representing 10, '`B`' representing 11, and so on, up to '`F`' for 15. Hexadecimal numbers are written in SB using a leading '`0x`' or '`&H`', to indicate their base. Thus, `0x12` is 18 (1 times 16 plus 2).

I/O Abbreviation for "Input/Output", the act of moving data into and/or out of a running program.

Interpreter
A program that reads and executes human-readable source code directly. It uses the instructions in it to process data and produce results.

ISO The International Standards Organization. This organization produces international standards for many things, including programming languages, such as C and C++.

Lesser General Public License
LGPL This document describes the terms under which binary library archives or shared objects, and their source code may be distributed.

Octal Base-eight notation, where the digits are `0`–`7`. Octal numbers are written in SB using a leading '`&o`', to indicate their base. Thus, `&o13` is 11 (one times 8 plus 3).

POSIX The name for a series of standards that specify a Portable Operating System interface. The "IX" denotes the Unix heritage of these standards.

Private Variables and/or functions that are meant for use exclusively by this level of functions and not for the main program. See `LOCAL`, "Nested Functions".

Recursion When a function calls itself, either directly or indirectly.

Redirection
Redirection means performing input from something other than the standard input stream, or performing output to something other than the standard output stream.

In Unices, you can redirect the output of the `print` statements to a file or a system command, using the '`>`', '`>>`', '`|`', and '`|&`' operators. You can redirect input to the `INPUT` statement using the '`<`', '`|`', and '`|&`' operators.

RegExp
Regulat Expression
Short for *regular expression*. A regexp is a pattern that denotes a set of strings, possibly an infinite set. For example, the regexp '`R.*xp`' matches any string starting with the letter '`R`' and ending with the letters '`xp`'.

Search Path
In SB, a list of directories to search for SB program files. In the shell, a list of directories to search for executable programs.

Seed The initial value, or starting point, for a sequence of random numbers.

Shell The command interpreter for Unix, POSIX-compliant systems, DOS and WinNT/2K/XP (CMD). The shell works both interactively, and as a programming language for batch files, or shell scripts.

Unix A computer operating system originally developed in the early 1970's at AT&T Bell Laboratories. It initially became popular in universities around the world and later moved into commercial environments as a software development system and network server system. There are many commercial versions of Unix, as well as several work-alike systems whose source code is freely available (such as GNU/Linux, NetBSD, FreeBSD, and OpenBSD).

Appendix G GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written 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 noncommercially. 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 that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

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. (For example, 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.

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 "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and 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 has been designed to thwart or discourage subsequent modification by readers is not Transparent. 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 designed for human modification. Opaque formats include PostScript, PDF, 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 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.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, 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 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 publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. 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 less than five).

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", and 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. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements 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 retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

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.

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 "Acknowledgements", 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, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they 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 quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around 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 provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense 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.

G.1 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 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

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.

Appendix H Command Index

P

Q

R

S

T

U

V

W

X

Y

Appendix I Variable Index