DESCRIPTION

Rabbit BASIC is an interpreter for a dialect of the BASIC programming language loosely modelled around the historic Dartmouth BASIC. The interpreter operates either in interactive mode, which permits issuing individual commands or editing the current program, or in run mode, executing the program loaded into the main memory.

At startup, the interpreter attempts to load and execute a program called *autoexec.bas* in the current directory on UNIX machines, or from the root directory of the on-board Flash drive when running on the ULX2S FPGA board. If the startup program can not be found or its execution terminates, an interactive line editor is invoked which accepts and processes all subsequent input from the controlling terminal.

INTERACTIVE MODE

All lines prepended with a numeric line identifier are assumed to be a part of a program and will be stored in the main memory at a position corresponding to the specified line number, and will not be further evaluated or interpreted until program execution begins. Conversely, lines entered without a line number will be interpreted and executed immediately. The command line interpreter features a history buffer which permits repeating and editing previously issued commands. For proper operation the line editor requires a terminal emulator capable of correctly handling a small subset of VT-100 control sequences as well as encoding navigation keys using VT-100 control codes.

The following control keys may be used for navigation and line editing:

<ctrl-C>
: Interrupts editing of the current line, discarding all input.

<ctrl-R>
: Redraws the current line. Handy when the terminal emulator does not correctly interpret all VT-100 control sequences.

<ctrl-L>
: Clears the entire screen, then redraws the current line.

<ctrl-P> or **<cursor-up>**
: Discards the current line and replaces it with the previous line from history buffer.

<ctrl-N> or **<cursor-down>**
: Discards the current line and replaces it with the next line from history buffer.

<ctrl-B> or **<cursor-left>**
: Moves cursor one position to the left.

<ctrl-F> or **<cursor-left>**
: Moves cursor one position to the right.

<ctrl-A> or **<Home>**
: Moves cursor to the first character.

<ctrl-E> or **<End>**
: Moves cursor to the end of the line.

<ctrl-H> or **<Backspace>**
: Deletes the character before current cursor position.

<ctrl-D> or **<Delete>**
: Deletes the character at current cursor position.

<ctrl-O> or **<Insert>**
: Toggles between insert and overwrite mode.

<ctrl-I> or **<Tab>**
: Inserts space characters so that the new cursor position becomes aligned to multiples of four.

COMMANDS

REM or **or** ’ { text }>

An arbitrary comment, ignored by the interpreter.

LET var = exp

Evaluate the expression *exp* and assign its value to variable *var*. The LET keyword may be omitted for brevity.

PRINT or **? { arg,** ... **}**

PRINT #f { arg, ... }

Prints a variable number of arguments to the terminal. The arguments may be separated either by commas or semi-colons. A comma indicates that the cursor will be advanced to the next tab stop (typically 8 characters). The output may be directed to a file if a file descriptor preceeded by a hash sign is provided as the first argument.

INPUT { #f, } var1 { , ... }

INPUT "txt"; var1 { , ... }

Input data from a terminal or from a file and store it into named variable(s). An optional message string *txt* can be displayed before processing input from the controlling terminal.

LINPUT { #f, } var

LINPUT "txt"; var

Input a whole line (ignoring separators) into a single string variable *var*.

GOTO lnum

Unconditionally jump to the program line specified by *lnum*.

GOSUB lnum

Jump to a subrutine at line *lnum*, saving the current line number on the return stack.

ON expr GOTO lnum1 { , lnum2 ... }

Jump to the line number selected from the list following the GOTO statement and indexed by the value of *expr*. If *expr* is less than or equal to zero or exceeds the number of elements in the list then proceed to the next instruction.

ON expr GOSUB lnum1 { , lnum2 ... }

Jump to a subroutine at the line number selected from the list following the GOTOSUB statement indexed by the value of *expr*, saving the current line number on the return stack. If *expr* is less than or equal to zero or exceeds the number of elements in the list then proceed to the next instruction.

RETURN

Return from a subroutine, resuming execution at the line following the last GOSUB command.

IF expr THEN commands { ELSE commands }

Evaluate the expression *expr* and if the result is non-zero execute command(s) following the THEN keyword. Optionally, an ELSE keyword may be included followed by command(s) to be executed if the expression *expr* evaluates to zero. A line line number may be specified instead of a command following either or both THEN and ELSE keywords, which implies a GOTO to the requested line.

FOR var = expr1 TO expr2 { STEP expr3 }

Beginning of a FOR loop. The value of variable *var* will be initialized to the value of *expr1* and will be increased on each subsequent loop iteration either by one or by the value of *expr3* if the optional STEP keyword is specified. Looping will continue as long as the value of *var* is less then or equal to *expr2*. The loop is always executed at least once.

NEXT { var, ... }

Increase the value of variable *var* either by one or by the alternative STEP specified in the corresponding FOR statement. If the value of the variable exceeds the limit specified in the FOR

statement, advance to the next program line, otherwise jump to the line immediately following the FOR statement. Multiple variable names may be specified following the NEXT keyword, in which case each one is processed only after the completion of the inner loop. Alternatively, the NEXT keyword may be used with no arguments, in which case it implies closure of the first unterminated FOR loop.

WHILE expr

Beginning of a WHILE - WEND loop. The body of the loop is executed as long as the value of the expression *expr* evaluates to a non-zero value. The expression *expr* is evaluated *before* the body of the loop gets executed.

WEND

Terminate the body of a WHILE loop.

REPEAT

Begining of a REPEAT - UNTIL loop.

UNTIL expr

Terminating statement of a REPEAT - UNTIL loop. The body of the loop is executed as long as the value of the expression *expr* evaluates to a non-zero value. The expression *expr* is evaluated *after* the body of the loop gets executed, so the commands inside the loop are guaranteed to be executed at least once.

DATA constant { , constant ... }

Declare comma separated numerical or string constants to be used by READ statements. DATA statements are not permitted inside IF - THEN - ELSE constructs.

READ var { , var ... }

Read string or numeric constant(s) from DATA statements embedded in the program and assign them to variable(s) provided as arguments.

RESTORE { lnum }

Restore the pointer for reading DATA constants to the start of the program, so that the constants can be READ again. If an optional line number *lnum* is provided then the restore occurs from the start of that line. If no DATA statements are found then the RESTORE command searches from the start of the program.

DIM var(d1 {, d2} {, d3}) {, ... }

Declare and allocate memory for a list of arrays (string or arithmetic). A maximum of three subscripts can be used. All arrays must be declared via DIM before use.

BASE 0 | 1

Specify the starting index for arrays, which may be either zero or one.

OPEN stringexp { FOR INPUT | OUTPUT | APPEND | TERMINAL } AS exp

Open a file named *stringexp* to be used with file descriptor *exp*. Output mode is implied, hence the 'FOR OUTPUT' option may be omitted for brevity.

CLOSE exp

Close a file with file descriptor *exp*. Releases the file descriptor and flushes out all buffered data.

ON ERROR GOTO lnum

Register an error handler routine at line *lnum*.

RESUME { lnum }

Return from an error handler. Optionally, do not return to the instruction which triggered the error, but to the line *lnum*.

DEF FNname(var {,var }) = exp

Define a function *FNname* as a single-line expression *exp*.

DEF FNname(var {,var })
Define a function *FNname* as a subroutine which may include multiple lines terminated by a *FNEND* statement. The result is returned by simply assigning *FNname* a value before reaching the terminating *FNEND* statement.

DEFPROC name(var {,var })
Define a procedure *name* as a subroutine which may include multiple lines terminated by a *FNEND* statement. Unlike functions, procedures do not return values.

FNEND
Terminating statement of a *DEF FNname* or a *DEFPROC* block.

LOCAL var1 {, var2 .. }
Declare variables with a scope local to functions and procedures. The *LOCAL* statement should be placed immediately following *DEF FNname* or *DEFPROC* statements.

MID$(stringval, start {, len }) = stringexp
Assign *stringexp* to *stringval* starting from character *start* and either replacing next *len* characters or the remainder of the string.

CLS Clear the terminal screen.

POKE addr, byte
Write a *byte* into a memory location at *addr*.

RANDOM
Reseed the random number generator.

END Terminate program execution and return to the interactive mode command prompt.

STOP Terminate program execution and return to the interactive mode command prompt. Unlike the END command, the STOP command prints a message, and permits the execution to be resumed via the CONT command.

CONT Continue execution of a program which has been halted by a STOP command or via <*ctrl-C*>.

CLEAR
Clear all variables.

NEW Close all files, clear all variables, and clear program memory.

RUN { l }
Execute the currently loaded program. An optional numeric argument can be provided indicating a line number from which the program execution will begin. All variables are cleared and all currently open files are closed prior to starting program execution.

LIST { start } { - end }
Display the content of the program memory to the controlling terminal. Optionally a range of line numbers to display may be specified.

EDIT lnum
Edit an existing line of the program text.

AUTO { start {, step } }
Perform auto line numbering so that a program can be typed in without entering line numbers. An optional *start* line number and an increment *step* may also be specified.

DELETE start - end
Delete a range of lines between *start* and *end* inclusively.

BYE Terminate the execution of the interpreter, closing all files.

SAVE stringexp
Save the current program to a named file.

LOAD stringexp
Close all files and clear all variables, then load a program from file *stringexp*.

MERGE stringexp
Read a program from file *stringexp* end merge it with the current program stored in main memory. Program lines in current program which have the same line numbers as the lines from the file *stringexp* will be silently overwritten.

CHAIN stringexp
Load a program from file *stringexp*, and execute it immediately. Numeric variables are preserved but all arrays and strings are cleared.

ERROR exp
Execute the given error sequence, which may be useful for debugging of error handler routines.

DIR { path }
List directory contents of the current directory, or of the target *path* if provided.

CD path
Change current directory to *path*.

PWD Print the current directory.

KILL path
Remove a file or directory pointed to by *path*. Directories must be empty for the requestt to succeed.

MKDIR path
Create a directory at *path*.

COPY src_path, dst_path
Copy a file from *src_path* to *dst_path*. If the destination file already exists it will be silently overwritten.

RENAME from_path, to_path
Rename a file named *from_path* to *to_path*.

EXEC path
Load a binary program (MIPS executable) from file at *path* into SRAM and execute it, displacing the BASIC interpreter.

MORE path
Display an ASCII file pointed to by *path* to the controlling terminal line by line, pausing each page (24 lines) for terminal input. Pressing <*space*> displays another page, whereas pressing <*enter*> or <*j*> displays a single new line. The pager may be interrupted by pressing <*q*> or <*ctrl+C*>.

BAUDS expr
Change the serial console baud rate. The FT232R USB to UART bridge on the ULX2S FPGA board should work well with most standard baud rates ranging from 300 to 3000000 bauds. The default speed is 115200 bauds.

SLEEP expr
Pause program execution for *expr* seconds. Fractional values are permitted for specifying delays with sub-second resolution.

VIDMODE expr {, scaling {, onroot}}
Choose one of four possible video output modes, identified by integer values in range from 0 to 3. Mode 0 uses a fixed 8-bit colour pallete, whereas mode 1 uses a fixed 16-bit pallete for each of 512 (W) x 288 (H) pixels in a fixed-size video display matrix. Mode 2 displays a static test image, while mode 3 completely turns off the video output. Modes 2 and 3 do not consume any memory bandwidth, hence permit the CPU to operate at full speed, whereas activating the framebuffer (modes 0 and 1) may have a noticeable impact on program execution performance. By default the

video framebuffer is turned off (mode 3). An optional integer *scaling* factor ranging from 1 to 4 may be specified when displaying the graphical output on an X11 screen. Additionally, graphic output may be directed to the root window by setting *onroot* parameter to 1. Scaling factor and onroot parameters are silently ignored when BASIC is running on the ULX2S FPGA board. Note that each invocation of *VIDMODE* command implicitly clears all currently defined sprites (see below).

DRAWABLE expr

Sets drawable framebuffer to the value of *expr*, which may be either 0 or 1. Framebuffer 0 is the default, and is automatically allocated each time video mode gets changed via the *VIDMODE* command, whereas framebuffer 1 will be automatically allocated the first time it gets referenced using the *DRAWABLE* command.

VISIBLE expr

Sets visible framebuffer to the value of *expr*, which may be either 0 or 1. Framebuffer 0 is the default, whereas framebuffer 1 must be first allocated using the *DRAWABLE* command.

INK color

Select a color to be used in subsequent graphics operations. Colors may be specified in three different formats. If the argument provided is a string and the first character of the argument is "#", then next six characters are interpreted as hexadecimal values in form of *RRGGBB*, corresponding to 8-bit values of red, green and blue components. Alternatively, if the argument is a string and its first character is not "#", then the color key is searched for in the following pallete: *black*, *gray*, *gray25*, *gray50*, *gray75*, *white*, *red*, *green*, *navy*, *blue*, *teal*, *lime*, *cyan*, *indigo*, *maroon*, *purple*, *olive*, *brown*, *violet*, *khaki*, *magenta*, *orange*, *pink*, *yellow*. Finally, a color may be specified as a numeric value, which will be interpreted differently depending on the pallete in use (8-bit or 16-bit).

PAPER color

Select a color to be used as a background when drawing text. The same syntax and rules as for the *INK* command apply. Additionally, transparent background may be selected by specifying -1 as the color value.

PLOT x0, y0 {, x1, y1 ... }

Draw a single pixel at coordinates *(x0,y0)*. If additional coordinates are provided then continue drawing lines to coordinates corresponding to further argument pairs.

LINETO x0, y0 {, x1, y1 ... }

Draw a line from the last cursor position to a pixel at coordinates *(x0,y0)*. If additional coordinates are provided then continue drawing lines to coordinates corresponding to further argument pairs.

RECTANGLE x0, y0, x1, y1 {, fill}

Draw a border of a rectangle defined by the provided coordinates. If an optional argument *fill* is provided and its value is non-zero, then the entire region encompassed by the rectangle is filled with current color.

CIRCLE x, y, r {, fill}

Draw a circle at *x*, *y* with radius *r*. If an optional argument *fill* is provided and its value is non-zero, then the entire region encompassed by the circle is filled with current color.

TEXT x, y, stringexpr {, scale_x {, scale_y} }

Draw text *stringexpr* at *x*, *y*. Optional arguments *scale_x* and *scale_y* may be specified to increase the size of the font.

FILL x, y

Flood the area at coordinates *x* and *y* of the drawable framebuffer with the current ink color.

LOADJPG path

Load a JPEG image from a file pointed to by *path* directly to the drawable framebuffer.

SPRGRAB spr_id, x0, y0, x1, y1
Create a sprite uniquely identified by a positive integer *spr_id* and fill it with data from the drawable framebuffer enclosed in a rectangular area defined by coordinates *x0*, *y0*, *x1* and *y1*.

SPRLOAD spr_id, path {, downscaling_factor}
Create a sprite uniquely identified by a positive integer *spr_id* and populate it with JPEG image loaded from a file at *path*. An optional integer parameter *downscaling_factor* in range between 0 and 3 may be specified to reduce the size of the sprite. Note that creating sprites bigger than the framebuffer area (512 * 288) is permitted, though should be used with care in order to avoid memory exhaustion problems, especially on constrained platforms such as the ULX2S board.

SPRTRANS spr_id, color
Declare *color* as transparent for existing sprite *spr_id*.

SPRPUT spr_id, x, y
Place sprite *spr_id* on the drawable framebuffer at coordinates *x* and *y*.

SPRFREE {spr_id}
Destroy all defined sprites and return the occupied memory to the free memory pool. If an optional *spr_id* argument is provided, only the selected sprite is freed.

FUNCTIONS

MIN(x, ...)
Returns the minimum value among all of the provided arguments.

MAX(x, ...)
Returns the max value among all of the provided arguments.

ABS(x)
Returns the absolute value of x.

SGN(x)
Returns the sign of the argument x, which can be -1, 0 or 1.

INT(x) Return the integer part of x.

SQRT(x)
Returns the square root of x.

LOG(x)
Returns the natural logarithm of x.

LOG10(x)
Returns the logarithm in base 10 of x.

EXP(x)
Returns eˆx. e=2.7182818..

SIN(x) COS(x) TAN(x) ASIN(x) ACOS(x) ATAN(x)
Trignometric functions.

SINH(x) COSH(x) TANH(x) ASINH(x) ACOSH(x) ATANH(x)
Hyperbolic functions.

RND Returns an integer random number between 1 and 32767.

RND(x)
If x is zero returns a random number between 0 and 1 otherwise returns an integer random number between 1 and INT(x).

PEEK(x)
Returns the value of a byte from memory at address x.

MID$(a$, start {, len })
Returns a substring of *a$* between character *start* and the end of the string. If optional argument *len* is provided, the substring will be restricted to *len* characters.

RIGHT$(a$,j)
Returns the right j characters of a$.

LEFT$(a$,j)
Returns the left j characters of a$.

STRING$(a$,j)
Returns a$ repeated j times.

ERMSG$(j)
Returns the j'th error message.

CHR$(j)
Returns the ascii character corresponding to the value of j.

STR$(expr)
Evaluate numeric expression *expr* and conver the result to a string.

SPACE$(j)
Returns a string of j spaces.

DIR$(path$)
Returns the list of file names residing in a directory at path$.

LEN(a$)
Returns the length of string a$.

VAL(a$)
Returns the value of the number specified by the string.

ASC(a$)
Returns the ascii code for the first element of a$.

INSTR(a$, b$ {,c})
Return the position of first occurence of string *a$* inside string *b$*. If optional argument *c* is provided then the search begins from character *c*.

EOF(f) Returns true if the file specified by f has reached the end of the file.

POSN(f)
Returns the current printing position in the file. If f is zero then it is the printing position of the terminal.

EVAL(a$)
Evaluates the expression defined by the string a$. e.g. EVAL("12") returns the value 12.

PI Returns the value of pi. = 3.141592653589...

ERL Returns the line number of the last error. Zero if error was in immeadiate mode.

ERR Returns the error code of the last error.

TIM Returns a numeric value for the number of seconds since interpreter startup.

CURKEYS
Returns a bitmapped value corresponding to the current state of cursor buttons (see *LEDs, buttons and switches* below). When running in an X11 environment, space bar is mapped to the *btn_center* key.

MATHEMATICAL OPERATORS

^	exponentiation
*	multiplication
/	division
MOD	remainder
+	addition
-	subtraction

AND	bitwise and
OR	bitwise or
XOR	bitwise exclusive or
NOT	bitwise not
<=	less than or equal
<>	not equal to
>=	greater than or equal
=	equal
>	greater than
<	less than

EXPRESSION SYNTAX

```
stringexp  ::= string | string + stringexp
string     ::= qstring | stringvar | stringfunc
qstrings   ::= "any char" | ‘any char‘
stringvar  ::= numbvar$ | numbvar$[ dim1 { ,dim2 {, dim3 } } ]

val        ::= term | term sep val
             | not val | - val
term       ::= numb | valfunc | numbvr
             | stringexp csep stringexp
numb       ::= digit | digit digit+
             | digit* . digit*
             | digit* e {+ | -} digit+
             | digit* . digit* e {+ | -} digit+
digit      ::= 0 1 2 3 4 5 6 7 8 9
numbvr     ::= numbvar | subsc
numbvar    ::= lett | lett alpha+
subsc      ::= numbvar( val {, val { ,val } } )
sep        ::= + - * /  ^ and or xor | csep
csep       ::= <> > < >= <= =
usrfunc    ::=  fn/numbvar { (val { , val { , val } } ) }
```

ULX2S FPGA BOARD

ULX2S is a tiny FPGA prototyping board designed primarily as an affordable teaching aid to be used in basic digital design courses and to be easily embeddable in more complex digital systems. A pre-built FPGA bitstream with a system-on-a-chip configuration centered around a 32-bit RISC CPU core operating at 81.25 MHz also permits execution of various software packages, including a BASIC interpreter.

An on-board SPI Flash memory chip can be accessed from BASIC as disk drive *"C:"*. In standard configuration the SPI Flash drive also hosts an executable binary of the BASIC interpreter which is automatically loaded by the ROM bootloader when the board is powered up. Data can be read from the on-board Flash drive at rates of up to 10 MBytes/s, while writing speed is limited to around 185 KBytes/s. The SPI Flash chip does not provide any wear-leveling machinery, so write access should be moderately exercised to avoid exceeding the chip’s declared endurance of around 100.000 write cycles.

A MicroSD, MicroSDHC or MicroSDXC card formatted with the FAT32 file system and inserted in the MicroSD slot should be accessible as disk drive *"D:"*. Read speeds of up to 4.5 MBytes/s and write speeds of up to 2.5 MBytes/s may be sustained depending on card type, data layout and access patterns.

1 MByte of on-board static RAM is mapped by the SoC configuration to address 0x80000000, which is also the location where the BASIC interpreter is automatically loaded by the ROM bootloader. The linear video framebuffer, if enabled, occupies either 147456 or 294912 bytes of SRAM depending on the selected pallete (8- or 16-bit).

All I/O ports are memory-mapped to a region starting at 0xffff8000, which permits I/O ports to be addressed using small negative integers. The following ports may be safely accessed from BASIC using PEEK and POKE:

General-Purpose Input / Output (GPIO)

A total of 29 pins on DIL connectors J1 and J2 can be controlled via GPIO ports. GPIO data ports can be both read and written to, while bits in the corresponding control ports determine whether each pin is configured as input (control bit cleared) or as output (control bit set). By default all pins are configured as input.

-256 (0xffffff00): GPIO data, byte 0 (input / output)

bit 0: pin j1_2
bit 1: pin j1_3
bit 2: pin j1_4
bit 3: pin j1_8
bit 4: pin j1_9
bit 5: pin j1_13
bit 6: pin j1_14
bit 7: pin j1_15

-255 (0xffffff01): GPIO data, byte 1 (input / output)

bit 0: pin j1_16
bit 1: pin j1_17
bit 2: pin j1_18
bit 3: pin j1_19
bit 4: pin j1_20
bit 5: pin j1_21
bit 6: pin j1_22
bit 7: pin j1_23

-254 (0xffffff02): GPIO data, byte 2 (input / output)

bit 0: pin j2_2
bit 1: pin j2_3
bit 2: pin j2_4
bit 3: pin j2_5
bit 4: pin j2_6
bit 5: pin j2_7
bit 6: pin j2_8
bit 7: pin j2_9

-253 (0xffffff03): GPIO data, byte 3 (input / output)

bit 0: pin j2_10
bit 1: pin j2_11
bit 2: pin j2_12
bit 3: pin j2_13
bit 4: pin j2_16
bits 5 to 7: not connected

-252 (0xffffff04): GPIO control, byte 0 (output only)

-251 (0xffffff05): GPIO control, byte 1 (output only)

-250 (0xffffff06): GPIO control, byte 2 (output only)

-249 (0xffffff07): GPIO control, byte 3 (output only)

LEDs, buttons and switches

-240 (0xffffff10): pushbuttons (inpuut)

bit 0: btn_right (input)
bit 1: bnt_left (input)
bit 2: btn_down (input)
bit 3: btn_up (input)
bit 4: btn_center (input)

-239 (0xffffff11): LEDs (output)

bits 0 to 7: led_0 to led_7 (output)

-238 (0xffffff12): DIL switches (input)

bits 0 to 3: sw_0 to sw_3 (input)
bits 4 to 7: not connected

DIAGNOSTICS

When the interpreter discovers an error it will call an error trapping routine. The errors can be caught by the user program using the on-error feature. If no error trapping routine has been supplied a message is printed with the corresponding line number.

EXAMPLES

Compute a sum of two numbers:

>**? 1 + 2**
3
Ready

Compute a sine function:

>**? sin(pi/4)**
0.707106781
Ready

Concatenate two strings:

> **a$ = "abc"**
Ready
> **b$ = a$ + "def"**
Ready
>**? b$**
abcdef
Ready

Iterate three times through a FOR loop:

```
> for i = 1 to 3 : print "iteration #"; i : next i
iteration # 1
iteration # 2
iteration # 3
Ready
```

Display random values on LEDs until a button is pressed on the ULX2S board or until <ctrl+C> is received on the controlling terminal:

```
> repeat : poke -240, rnd(255) : sleep 0.1 : until peek(-240) > 0
Ready
```

A short program for computing factorials:

```
>10 input "f:"; f
>20 r = 1 : for i = 1 to f : r = r * i : next i
>30 print f; "! ="; r
>40 goto 10
>list
  10 INPUT "f:"; f
  20 r = 1 : FOR i = 1 TO f : r = r * i : NEXT i
  30 PRINT f; "! ="; r
  40 GOTO 10
Ready
>save "factorial.bas"
Ready
>run
f:3
 3! = 6
f:8
 8! = 40320
f:100
 100! = 9.33262154e157
f: <ctrl+C>
breaking at line 10
Ready
```

BUGS

The RENUMBER command fails to properly track and update goto targets hidden inside IF .. THEN .. ELSE constructs.

REPEAT - UNTIL loops inside functions, procedures or nested inside other loops apparently do not work.

The MOD operator is implemented using *fmod(3)*, so the result may or may not include a fractional part.

DISCLAIMER

(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

AUTHORS

Phil Cockcroft created the Rabbit BASIC in early 1980's while he was at University College, London. He released the source code to the Public Domain in 1986 and continued to further improve and maintain it until mid-1990's. In 2013. Marko Zec added features specific to the ULX2S FPGA board, such as file management and framebuffer routines, and rewrote the line editor as well as the most of this manual.