Chipmunk Basic Man Page

```
BASIC(1)           Chipmunk Basic v3.6.5                 BASIC(1)
```

Chipmunk BASIC - 'BASIC' language interpreter

SYNOPSIS   (UNIX)

```
basic [ filename ]
```

DESCRIPTION

Chipmunk basic is an interpreter for the BASIC language. If a filename parameter is given, then the named program file is loaded into memory and run.

Basic commands and statements can be entered and interpreted in immediate mode or executed as program statements when the Basic program is run. A built-in line number based editor allows program input from the console keyboard. See below for the commands and statements which the interpreter recognizes.

FLAGS

none

COMMANDS

Standard mumbleSoft-like Basic Commands:

load STRINGEXPR

> Load a program into memory from the named file. The program previously in memory is erased. All files are closed and all variables are cleared. Lines beginning with the '#' character will be treated as comments. All other lines in the file must begin with a line number. Duplicate line numbers are not allowed.

```
save STRINGEXPR

        Save the current program to the named file.

new

        Erase the program in memory.  All files are closed and
        all variables are cleared.

clear

        All  variables are cleared.  All arrays and string
        variables are deallocated.

run { LINENUM }
run { STRINGEXPR { , LINENUM } }

        Begin execution of the program at the first line, or at
        the specified line.  All variables are cleared.  If a
        STRINGEXPR is given then the BASIC program with that
        name file is loaded into memory first.  Program lines
        are executed in line number order.

cont

        CONTinue execution of the program on the next statement
        after the statement on which the program stopped
        execution due to a STOP command or an error.  See BUGS
        section.

LINENUM { TEXT }

        Enters a program line.  If a program line with
        line number LINENUM exists, then it is replaced.
        If no TEXT is given, the the program line with
        line number LINENUM is deleted.

list
        List the whole program.
        Line numbers above 999999999 will not list.

list 1-3
        List lines 1 to 2

list -2
        List lines up to 1
```

```
list 1
        List line 1

list 2-
        List lines from 2 on

merge STRINGEXPR

        Loads a program into memory.  The previous program
        remains in memory; variables are not cleared.  If a
        line exists in both programs, the new merged line
        overwrites the old one.

renum STRINGEXPR VAL { , VAL { , VAL { , VAL} } }

        Renumber program lines.  By default, the new sequence is
        10,20,30,... The first argument is a new initial line
        number; the second argument is the increment between
        line numbers. The third and fourth arguments, if
        present, specify a limiting range of old line numbers
        to renumber.  RENUM can be used to move non-overlapping
        blocks of code.

edit LINENUM

        Edit a single line. If the exit from the edit is via a
        cntrl-c then the program line is not modified.
                i       insert till  key
                x       delete one char
                A       append to end of line

del LINENUM [ - LINENUM ]

        Delete a line or specified range of lines. If not found
        then no lines will be deleted.

exit
bye
quit

        Terminates the basic interpreter, ending program
        execution and closing all files.
```

STATEMENTS

```
{ let } VAR = EXPR

        Assign a value to a variable.  Variable names can be up
        to 31 significant characters, consisting of letters,
        digits, underscores, and an ending dollar sign.
        Variable names are case insensitive.  Variables can
        hold real numbers (IEEE double) or strings of up to 254
        characters.  If the variable name ends with a "$" it
        holds strings, otherwise it holds numbers.  If a
        statement starts with a variable name then an implied
        LET is assumed.

end

        Terminates program execution and returns to the command
        prompt.  Not required.

stop

        Stops the execution of the program and returns to
        the command prompt.  Prints a "Break..." message.

if EXPR then STATEMENT { : STATEMENT } { { : } else STATEMENT }
if EXPR then LINENUM
if ( EXPR ) { then }

        The IF statement.  If the condition is true then the
        STATEMENTS after the THEN are executed and the
        statements after the ELSE are skipped.  If the
        condition is false then the statements after the "else"
        are executed instead.  If the item after "then" is a
        line number then a goto is executed.

        Multi-line block syntax:
        IF (EXPR) { THEN }
        ...
        { ELSE }
        ...
        ENDIF

        If there are no statements on the same line as
        the IF statement, and the condition is true, then
        statements are executed until a line with an ELSE
        or ENDIF is found.  If an ELSE is found first the
        following statements are skipped until an ENDIF is
```

found. If the condition is false, then statements are skipped until and ELSE is found. IF .. ENDIF blocks may be nested.

for VAR = EXPR to EXPR { step EXPR }

Beginning of a FOR-NEXT loop. It takes a starting value, a limit and an optional step argument. If the step value is negative, the variable counts down. The body of the loop is not executed if the end condition is true initially.

Example:
```
        for i=1 to 10 : print i, : next i
        rem prints the numbers from 1 through 10
```

next { VAR }

End of a FOR-NEXT loop. If the termination conditions are met then execution falls through to the following statement, otherwise execution returns to the statement following the FOR statement with the corresponding index variable. If there no index variable parameter, the innermost FOR loop is used.

exit for

Exits the current FOR-NEXT loop.

while { EXPR }

Start of a WHILE loop. The loop is repeated until EXPR is false. If EXPR is false at loop entry, then the loop is not executed . A WHILE loop must be terminated by a balancing WEND statement.

wend { EXPR }

Terminating statement of a WHILE loop. If EXPR is true then exit the loop. Only one WEND is allowed for each WHILE. A WHILE-WEND loop without a condition will loop forever.

exit while

        Exits the current WHILE-WEND loop.

gosub LINENUM

        Transfer command to a line number. Save return address
        so that the program can resume execution at the
        statement after the "gosub" command. The recursion
        depth is limited only by available memory.

return

        Returns from the most recently activated subroutine
        call (which must have been called by GOSUB).

goto LINENUM

        This statement will transfer control to the line number
        specified. If the program is not running, then this
        command will begin execution at the specified line
        without clearing the variables. An "Undefined line"
        error will occur if LINENUM doesn't exist in the
        program.

```
on EXPR   goto  LINENUM { , LINENUM ... }
on EXPR   gosub LINENUM { , LINENUM ... }
```

        This command will execute either a goto or a gosub to
        the specified line number indexed by the value of EXPR.
        If EXPR is larger than the number of LINENUMs, then
        control passes to the next statement.

on error goto LINENUM

        If the error form is used, only one linenumber is
        allowed. LINENUM is the line to which control is
        transferred if an error occurs. A GOTO or CONT
        statement can be used to resume execution. An error
        inside a named SUB subroutine cannot be resumed from
        or CONTinued.

sub NAME (VAR { , VAR ... } }

        Subroutine entry. May be called by a CALL statement or
        by NAME. A SUB subroutine must be exited by a RETURN or
        END SUB statement. There should be only one RETURN or
        END SUB statement per SUB definition. The variables in
        the VAR list become local variables. String and numeric

        arguments are passed by value; array arguments must be
        pre-dimensioned and are passed by reference.

        Example:
```
                110  x = foo (7, j)  : rem Pass 7 and j by value.
                ...
                2000 sub foo (x,y,z) : rem z is a local variable
                2010   print x       : rem prints 7
                ...
                2080   foo = y+1     : rem return value
                2090 end sub
```

        Subroutine definitions may not be nested.

local (VAR { , VAR ... } }

        Declares a list of variables to be local to the
        enclosing sub subroutine. The scope of those variable's
        ends after any RETURN or END SUB statement.

select case EXPR

        Multi-way branch. Executes the statements after
        the CASE statement which matches the SELECT CASE
        expression, then skips to the END SELECT statement.
        If there is no match, and a CASE ELSE statement is
        present, then execution defaults to the statements
        following the CASE ELSE.

        Example:
```
                200 select case x
                210   case 2
                ...
                230   case 3, 4
                ...
                270   case else
                ...
                290 end select
```

dim VAR(d { , d { , d } }) { , VAR(d { , d { , d } }) ... }

        Dimension an array or list of arrays (string or numeric).
        A maximum of 4 dimensions can be used. The maximum

dimension size is limited by available memory. Legal array subscripts are from 0 up and including the dimension specified; d+1 elements are allocated. All arrays must be dimensioned before use.

Example:
```
10 dim a(10)  : rem  An 11 element array.
20 for i=0 to 10
30   a(i) = i^2
40 next i
50 print a(5) : rem  Should print 25
```

data ITEM { , ITEM }

DATA statements contain the data used in the READ statements. Items must be separated by commas. The items may be either numeric or string expressions, corresponding to the type of variable being read. Reading the wrong kind of object produces a "Type mismatch" error.

* IMPORTANT DIFFERENCE from other Basic versions *

String data must be encapsulated with quote marks.

read VAR { , VAR }

Read data from the DATA statements contained in the program. List items can be either string or numeric variables. Reading past the end the last DATA statement generates an error.

restore { LINENUM }

The RESTORE statement causes the next READ to use the first DATA statement in the program. If a LINENUM is given then the DATA statement on or after that particular line is used next.

rem or "`"

A remark or comment statement. Ignored by the program during execution, however a REM statement can be the target of a GOTO or GOSUB.

```
def fnNAME ( VAR { , VAR } ) = EXPR

        Define a user definable function.  Obsolete.

        Example:
                10 def fnplus(x,y) = x+y
                20 print fnplus(3,5) : rem prints 8
```

FILE AND INPUT/OUTPUT STATEMENTS

```
open STRINGEXPR for { input|output|append } as # FNUM

        Open a file. The { input|output|append } parameter
        specifies whether the file is to be read, written or
        appended.  If STRINGEXPR is "stdin" for input or
        "stdout" for output then the console will be used
        instead of a file.  A "file not found" error will
        occur if a non-existant file is specified in an OPEN
        for input statement.  FNUM must be an integer value
        between 0 and 8.

open STRINGEXPR for random as # FNUM len = VAL

        Opens a random access file.  Only GET and PUT statement
        are allowed to read and write random access files.

open ... else goto LINENUM

        See OPEN command.
        LINENUM is the line to which control is transferred if
        an error in opening a file occurs.  The variable ERL is
        set to the line number on which the file open error
        occured.

close # FNUM

        Close a file. Releases the file descriptor and flushes
        out all stored data.

print  VAL | STRINGVAL { [ , | ; ] VAL ... } { ; }
?      VAL | STRINGVAL { [ , | ; ] VAL ... } { ; }
print # FNUM, VAL ...

        This command will print its parameters tab delimited.
        If a semi-colon is used between parameters then no tab
        is inserted between the parameters.  The print
        output is terminated with a carriage return unless the
        parameter list ends with a semi-colon.  If a file
        descriptor is given then output is redirected to the
        given file.  If a

                tab(VAL);

        is found in a print statement, then print output will
        skip to the horizontal position specified by VAL.

print { # FNUM, } using STRINGVAL ; VAR { [ , | ; ] VAR ... }

        Prints formatted numbers.  Legal characters for the
        format string STRINGVAL are: + * $ # . E+ and trailing
        spaces.

        Examples:

                print using "**$###.##"; 1.23  :' ****$1.23
                print using "###.##"; 2.12345  :'   2.12
                print using "#.##E+##"; 2345.6 :'   2.35E+03

input  STRINGVAR | VAR  { , VAR }
input  "prompt ", { STRINGVAR | VAR  { , VAR } }
input  { # FNUM , } { STRINGVAR | VAR { , VAR } }

        Input from the console or from the file specified by
        FNUM. If the input is from the console then a prompt
        string can optionally be printed before input.

        *** NOTE ***

        All input to string variables is "line input"; a whole
        input line will be read into one string variable.  The
        number of comma seperated numeric values in the input
        data must be less than or equal to the number of
        numeric variables in the INPUT statement.  This INPUT
        usage is different from other versions Basic.

get STRINGVAR

        Gets one character from the console keyboard.
        Blocking (waits for a character).

fputbyte VAL, # FNUM
```

```
        Writes one byte to the file specified by FNUM.

fseek # FNUM, NUM

        Seek to file position NUM in file specified by FNUM.


get # FNUM, VAL, TYPED-VAR

        Reads one record from a random access file into VAR.

put # FNUM, VAL, TYPED-VAR

        Write one record to a random access file from VAR.

dim STRINGVAR as dbm$( STRINGEXPR )

        Open a sdbm database file using the filename contained
        in STRINGEXPR.  Creates a database if one doesn't exits.
        This database can be accessed by using or storing to the
        array string variable named by STRINGVAR ( STRINGKEY ).

close STRINGVAR

        Close a sdbm database file if one using that variable
        name is open.
```

MATRIX and OBJECT TYPE STATEMENTS

```
option base { 0 | 1 }
mat origin  { 0 | 1 }

        Sets the matrix index origin to either 0 or 1 for all
        MAT statements, including fill.  Defaults to 1.

mat ARRAY-VAR = EXPR

        Fills a 1 or 2 dimensional array with a constant
        value given by EXPR.  Lower bound = mat origin

mat ARRAY-VAR = idn ( { EXPR } )

        Fills a 2 dimensional array with an identity matrix.
        The array must have been previously dimensioned.
```

```
mat ARRAY-VAR = ARRAY-VAR
```

Copys a 2 dimensional array. The dimensions must match.

```
mat ARRAY-VAR = ARRAY-VAR { + | * } { EXPR | ARRAY-VAR }
```

Adds or multiplies a 2 dimensional array by either an expression or another array. The dimensions must be appropriate for matrix addition or matrix multiplication.

```
mat ARRAY-VAR = ( EXPR ) { + | - | * } ARRAY-VAR
```

Adds, subtracts or multiplies a 2 dimensional array by an expression. Note that the parenthesis around the expression are required.

```
mat ARRAY-VAR = transpose ARRAY-VAR
mat ARRAY-VAR = trn ARRAY-VAR
```

Transposes a 2 dimensional array. The dimensions of the first array must correspond to the transpose of the dimensions of the second array.

```
mat ARRAY-VAR = invert ARRAY-VAR
mat ARRAY-VAR = inv ARRAY-VAR
mat ARRAY-VAR = invert ARRAY-VAR else LINENUM
```

Inverts a 2 dimensional square array. If LINENUM is specified, control is transferred to LINENUM if the matrix is singular. Note that the default array origin is (1,1).

```
mat print ARRAY-VAR
mat print ARRAY-VAR ;
mat print #n, ARRAY-VAR
mat print using STRINGVAL ; ARRAY-VAR
```

Prints a 1 or 2 dimensional array. A trailing semicolon will try to pack the data without tabs. The format STRINGVAL in MAT PRINT USING applies to each element seperately.

```
fn fft1( 1, ARRAY_ELEMENT, ARRAY_ELEMENT, SIZE_VAL )
```

In-place 1d Discrete Fourier Transform of real and imaginary arrays of at least size SIZE_VAL, starting at the referenced array indexes. SIZE_VAL must be a power of 2. Uses FFT algorithm.

Example:
```
        fn fft1 ( 1, x(0), y(0), 256 )
```

type CLASSNAME

Creates a structure definition type. Each field requires a separate line. Legal types are string, integer, longint and double. The definition must conclude with an END TYPE statement. Use the DIM AS NEW statement to create records containing the structure specified by a TYPE statement.

Example:
```
        300 type person
        310   name as string * 32 : rem = 31 chars length
        320   age as integer      : rem  2 byte integers
        330   weight as double    : rem  8 byte doubles
        340 end type
        400 dim friend1 as new person
        410 friend1.name = "Mark" : friend1.age = 13
        420 print friend1.name, friend1.age
```

Created typed structure elements can only be used in expressions and assignment statements.

class CLASSNAME { extends SUPERCLASSNAME }

Creates a class definition. Class definitions can then be used to create objects with member functions (also called methods.) Classes inherit members from superclasses (single inheritance.)

Example:
```
CLASS bar
  y AS integer
  z AS PRIVATE double   ' private data
  s AS PUBLIC string    ' public keyword optional
  SUB blah(v)           ' public member function
    this.y = v + 7
```

```
          END SUB
        END CLASS

        DIM b AS NEW bar        ' create object b
        CALL b.blah(1)          ' send message "blah(1)" to b
```

CLASS and TYPE definitions are global, and cannot be nested inside other class definitions or subroutines.

dim VAR { (INT) } as new CLASSNAME

Create a record (TYPED-VAR) or object using a previously defined structure definition type created by TYPE...END TYPE or CLASS..END CLASS. Optionally creates an array of records or objects.

MISC COMMANDS

option degrees

Changes the trigonometric functions to use degrees instead of radians for all parameters and return results.

randomize EXPR

Seeds the random number generator with the integer EXPR. The pseudo-random number generator should return the same sequence when seeded with the same start value. The actual sequence may be system dependant.

erase VAR

Un-dimensions a dimensioned array. Frees memory.

{ let } mid$(STRINGVAR, EXPR1, EXPR2) = STRINGEXPR

Replace the sub-string in STRINGVAR, starting at character position EXPR1, with character length EXPR2, with the (EXPR2 in length) string STRINGEXPR.

{ let } field$(STRINGVAR, VAL { ,STRINGVAL }) = STRINGEXPR

Replace the N-th field of STRINGVAR with STRINGEXPR.

push VAR { , VAR ... }

Pushes one or more expressions or variables onto an internal stack. Expressions can be returned using the POP function; variables can be returned by using the POP statement.

pop VAL

POP statement (see also POP function). Pops VAL variables off the internal stack, restoring the value of those variables to their pushed values.

exec(STRINGEXPR)

Executes STRINGEXPR as a statement or command.
e.g. exec("print " + "x") will print the value of x.

cls

Clear the terminals screen. Leaves the cursor in the upper left corner. For Applesoft BASIC fans, the "home" command will also do this.

poke ADDR_EXPR, DATA_EXPR { , SIZE_VAL }

Poke a byte into a memory location. Unreasonable addresses can cause bus or segmentation errors.
Use a optional SIZE_VAL of 2 or 4 to poke short or long integers to properly aligned addresses.

NUMERIC FUNCTIONS

sgn(VAL)

Returns the sign of the parameter value. Returns 1 if the value is greater than zero , zero if equal to zero. -1 if negative.

abs(x)

Returns the absolute value of x.

int(x)
int(x,0)

Returns the integer value of x. Truncates toward

```
        minus infinity.  The absolute value of x must be
        less than 2^31-1.  The usage int(x, 0) truncates
        towards zero instead of minus infinity.


floor(x)

        Returns the integer value of x.
        Truncates toward negative infinity.

sqr(x)
sqrt(x)

        Returns the square root of x.

log(x)

        Returns the natural logarithm of x.

log10(x)

        Returns the logarithm base 10 of x.

exp(x)

        Returns e^x. e=2.7182818...

sin(x)
cos(x)
atn(x)

        Trigonometric functions: sin, cosine and arctangent.

atn(y,x)
atan(y,x)

        4 quadrant arctangent

pi

        Returns pi, 3.141592653589793...

rnd ( EXPR )

        Returns an integer pseudo-random number between 0 and
        int(EXPR)-1 inclusive. If EXPR is 1, then returns a
```

rational number between 0 (inclusive) and 1. If EXPR is negative then EXPR seeds the random number generator.

len(STRINGEXPR)

Returns the length of the string STRINGEXPR.

len(TYPED-VAR)

Returns the length, in bytes, of a typed record (one created by DIM AS).

val(STRINGEXPR | EXPR)

Value of the expression contained in a STRINGEXPR or EXPR. STRINGEXPR may be a string literal, variable, function, or string formula.

asc(STRINGEXPR)

Returns the ascii code for the first character of STRINGEXPR. A null string returns zero.

instr(a$, b$ { , VAL })

Returns the position of the substring b$ in the string a$ or returns a zero if b$ is not a substring. VAL is an optional starting position in a$

det (ARRAY-VAR)

Returns the determinant of a 2-dimensional square array.

fn dot (ARRAY-VAR, ARRAY-VAR)

Returns the dot product of two 1-dimensional arrays.

MISC FUNCTIONS

isarray(VAR)

If VAR is a dimensioned array, returns the number of dimensions, otherwise returns 0.

ubound(VAR [, EXPR])

If VAR is a dimensioned array, returns the maximum legal subscript of the first dimension of that array, else returns 0. Use EXPR to return other dimensions.

pop

POP function (see also POP statement). Pops one variable value off the stack and returns that value (string or numeric).

(POP can be used as either a statement (with a parameter) or a function (no parameter). Note that the POP function, unlike the POP statement, does not restore the value of the variable pushed, but only returns the pushed value. This use of the POP statement is different from the Applesoft usage.)

varptr(VAR | STRINGVAR)

Returns the memory address of a variable.

fn eval(STRINGEXPR)

Value of a formula string contained in a STRINGEXPR.

For example, fn eval("1 + sqr(4)") yields 3.

fn bswap16(VAL)

Returns a byte swapped 16-bit value.

fn bswap32(VAL)

Returns a byte swapped 32-bit value.

fn version$()

Returns a string including version number, "BE/LE" for endianess, OS, and graphics availability

fn lnumexists()

Returns 1 if a line number matching VAL exists in
the current program.

fn bigendian()

Returns 1 if words are stored in memory in bigendian
byte order, 0 if in little endian byte order

erl

Returns the line number of the last error. Zero if the
error was in immediate mode. The variable errorstatus$
gives the error type.

timer

Returns a numeric value of elapsed of seconds from the
computers internal clock.

timer()

Returns a elapsed time in OS dependant format, perhaps
milliseconds or ticks.

peek(ADDR { , SIZE_VAL })

Returns the value of the byte in memory at address ADDR.
If SIZE_VAL is 2 or 4, returns the value of the 16-bit
or 32-bit word respectively (if correctly aligned).
If SIZE_VAL is 8, returns the value of the numeric
variable located at ADDR. (peek(varptr(x),8) == x)

STRING FUNCTIONS

x$ + y$

String concatenation.

String concatenation (and the MID$, LEN and INSTR
functions) can handle strings of up to 32766 characters
in length (if the memory available to the program
permits).

chr$(VAL)

```
        Returns the ascii character corresponding to the value
        of VAL.

str$( VAL { , EXPR } )

        Returns a string representation corresponding to VAL.
        If EXPR is present then the string is padded to that
        length.

format$( VAL , STREXPR )

        Returns the string representation of VAL formatted
        according to the format string STREXPR. The format
        string STREXPR uses the same formatting syntax as the
        PRINT USING statement.

mid$( a$, i { , j } )

        Returns a substring of a$ starting at the i'th
        positions and j characters in length. If the second
        parameter is not specified then the substring is
        taken from the start position to the end of a$.

right$(a$, EXPR )

        Returns the right EXPR characters of a$.

left$(a$, EXPR )

        Returns the left EXPR characters of a$.

field$( STRINGVAL, VAL { , STRINGVAL } )

        Returns the N-th field of the first string.  If the
        optional string is present then use the first character
        of that string as the field separator.  The default
        separator is a space.  Similar to UNIX 'awk' fields.

        e.g.  field$("1 22 333 4", 3)  returns  "333"

        If VAL is -1 then returns a string with a length
        equal to the number of seperators in the first string.

hex$( VAL { , EXPR } )
bin$( VAL { , EXPR } )
```

Returns the hexadecimal or binary string representation corresponding to VAL. If EXPR is present then the string is padded with zeros to make it that length.

lcase$(STRINGVAL)

Returns STRINGVAL in all lower case characters.

errorstatus$

Returns the error message for the last error.

FILE FUNCTIONS

inkey$

Return one character from the keyboard if input is available. Returns a zero length string { "" } if no keyboard input is available. Non-blocking. Can be used for keyboard polling.

input$(EXPR { , FILENUM })

Returns EXPR characters from file FILENUM. If f is not present then get input from the console keyboard.

fgetbyte(FILENUM)

Reads one byte from the open file specified by FILENUM and returns an unsigned numeric value [0..255].

eof(FILENUM)

Returns true if the the last INPUT statement, INPUT$ or FGETBYTE function call which referenced the text file specified by FILENUM tried to read past the end of file. (Note that reading the last line of a file will not read past the eof mark. A subsequent read is needed to set the EOF flag to true. Reading past the end-of-file will not report an error.)

fn eol(FILENUM)

Returns line termination character of previous line input statement (usually CR, LF or 0).

OPERATORS

The following math operators are available:

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

logical operators: (any non-zero value is true)

not	logical not

bitwise operators:

and	bitwise and
or	bitwise or
xor	bitwise exclusive-or

comparison operators:

<=	less than or equal
<>	not equal to
>=	greater than or equal
=	equal
>	greater than
<	less than

x$=y$, x$y$, x$<=y$, x$>=y$, x$<>y$

String comparisons; result is 1 if true, 0 if false.

Operator precedence (highest to lowest):

```
( )
functions()
^
-{unary_minus}
* / mod
+ -
= < > <= >= <>
not
```

```
            and
            or xor


UNIX/linux (and MacOSX Terminal) functions:

    argv$
            Returns the UNIX shell command line arguments.

    fn sysfork()

            Forks process.  Returns 0 to the child process and
            the pid of new process to the parent process.

    exit( EXPR )

            Exits Basic interpreter with status value of EXPR.


UNIX/linux (and MacOSX Terminal) commands:

    #cbas#run_only

            When used in a Basic program file, upon load sets
            the  Chipmunk  Basic  interpreter to run-only mode.
            Interactive mode disabled.  Thus the Stop and  End
            commands and the use of Ctrl-C will quit the
            interpreter.


UNIX/linux/MacOSX commands:

    open "pipe:" + STRINGEXPR for input  as # FNUM
    open "pipe:" + STRINGEXPR for output as # FNUM

            Opens an input or output pipe with STRINGEXPR as
            the process command string.

            example: open "pipe:/bin/ls -l" for input as #1

    open "socket:" + STRINGEXPR , EXPR for input as # FNUM
    open "socket:" + STRINGEXPR , EXPR for output as # FNUM

            Open a socket to the server and port specified by
            STRINGEXPR and EXPR.  The input port must be opened
            before the output port.
```

UNIX/linux/MacOSX functions:

```
    sys( STRINGVAL )

            UNIX system call.  The string parameter is given to
            the shell as a command.  Returns exit status.

    system$( STRINGVAL )

            UNIX shell command.  The string parameter is given to
            the shell as a command.  Returns first line of shell
            standard output.

    getenv$( STRINGVAL )

            Returns value for environment name STRINGVAL.

    fn pid()
    fn ppid()
    fn uid()

            Returns system process and user info.

    fn sleep( VAL)

            Sleeps process for VAL seconds.

    fn shmem( key, mode, length )

            Returns pointer to shared memory segment or 0 for
            fail.  If mode > 0 then creates shared memory area
            (see unix shmget), otherwise tries to find existing
            one matching key.  If length = 0 then releases
            shared memory area.  Use of only one shared memory
            area at a time supported.  Length is in bytes.

    fn kill( STRINGVAL )

            Deletes the file named by STRINGVAL.  Returns 0 if
            successful.
```

Macintosh GUI & Terminal commands:

```
    morse STRINGVAL { , VAL, VAL, VAL, VAL }

            Plays morse code through the speaker.
            The parameters are: word-speed-wpm, volume{0..100},
            dot-speed-wpm, frequency{in Hz or cps}

    sound VAL, VAL, VAL

            The parameters are:
            frequency{in Hz}, seconds_duration, volume{0..100}

    say STRINGVAL

            Speaks STRINGVAL if the Speech Manager Extension is
            resident.  Try "say a$,200,46,1" for faster speech.
            The last parameter is the voice selector.


Unix X11 GUI commands:

    graphics 0

            Opens a graphics window if X11 library is available.

    graphics moveto VAL, VAL

            Sets the (x,y) location of the graphics pen.

    graphics lineto VAL, VAL

            Draws a line from the current pen location to location
            (x,y) in the graphics window.


Macintosh GUI commands:

    graphics 0

            Opens a graphics window.

    moveto VAL, VAL

            Sets the (x,y) location of the graphics pen.

    lineto VAL, VAL
```

```
            Draws a line from the current pen location to location
            (x,y) in the graphics window.


Macintosh GUI commands:

    *** NOTE ***

            Many MacOS specific functions and commands are only
            documented in the Chipmunk Basic quick reference file.

    gotoxy VAL, VAL

            Set the horizontal and vertical location of the
            text output cursor.  (0,0) is the upper left corner.

    window x, y, char_cols, char_lines

            Change the text console window position and size.

    open "SFGetFile" for input  as #FNUM
    open "SFPutFile" for output as #FNUM

            Puts up a standard file dialog for the file name.

    files { STRINGVAL }

            Displays a listing of files in the named or current
            directory.


Macintosh GUI functions:

    fre
            Returns the amount of memory left for program use.


Macintosh functions:

    date$
            Returns a string corresponding to the current date.

    time$
            Returns a string corresponding to the current time.

    pos(VAL)
```

Returns the horizontal position of the text cursor.
If VAL is negative returns the vertical position.

errorstatus$

Also returns the full path name of the program and files opened by SFGetFile and SFPutFile (Only under System 7 and only if the path name fits in a string variable)

Macintosh menu items:

Open or O	will put up a dialog to allow selection of a program file to load. Basic Program file names must end with a ".bas" suffix.
Copy	will allow copying picts from the graphics window.
.	Command-period will stop program execution.

RESERVED WORDS AND SYMBOLS

```
+ - * / ^ mod  and or xor not  > < >= <= <> = ()
sqr log exp sin cos tan atn  pi
abs sgn int rnd peek val asc len
mid$ right$ left$ str$ chr$  lcase$ ucase$
goto  if then else endif  gosub return
for to step next  while wend  select case
rem  let  dim erase  data read restore   field$
input print open for output append as close# load save
random lof loc get put
inkey$  input$ eof  files  fgetbyte# fseek# fputbyte
run stop end exit quit cont  renum  new clear
date$ time$ timer  sound morse say  doevents
home cls gotoxy htab vtab pos
graphics sprite pset moveto lineto window scrn mouse
varptr peek poke fre push pop  isarray
sub call usr  def fn
type class extends  string integer single double
asin acos sinh cosh tanh log10 floor true false ubound

eqv imp  static  option degrees radians redim
```

```
    msgbox  do loop until break
    method private public local   menu dialog memstat()
    draw play  bload bsave  min max mat
    each  resume  function
    key is each set width swap dbm$
```

CONVENTIONS

```
    EXPR            an expression that evaluates to a numeric value.
    STRINGEXPR      a string expression.
    VAR             a numeric variable.
    STRINGVAR       a string variable. Name must end with a "$".
    INTEGERVAR      a 16-bit variable. Name must end with a "%".
```

All program lines must begin with a line number unless the whole program is loaded from a file.
Using spaces (indentation) between the line number and program statements is legal. Line numbers can be between 1 and 2147483647. Programs lines must be no longer than 254 characters in length.

Variable names starting with "fn" are reserved for the (obsolete) def fn functions.

Hexadecimal numbers can be entered by preceding them with a "0x" as in 0x02ae, or by "&h" as in &h0172.

Multiple statements may be given on one line, separated by colons:

```
        10 INPUT X : PRINT X : STOP
```

DIAGNOSTICS

Some errors can be caught by the user program using the "on error goto" command. If no error trapping routine has been supplied then program execution is terminated and a message is printed with the corresponding line number.

CHANGES

```
    v3.6.5(b3) - fixed mat multiply
               - fixed bug in format$() with leading zeros
```

```
              - fixed bug in exit while statement
              - fixed bugs with empty for and empty edit statements
              - fixed object string bug and object array bug
    v3.6.4(b7) - added local statement
    v3.6.4(b5) - add fn eval(), #ifndef _chipmunkbasic_
    v3.6.4(b1) - add det(), fn dot(), allow next j,i
              - bsave & mat inv size limits increased
    v3.6.4(b0) - added fn sleep(), dim as string
              - fixed rnd() scaling, format$()
    v3.6.3(b7) - fixed socket read eof, object print, fft
    v3.6.3(b4) - fixed exit for, sub return, mat a*b
    v3.6.3b1 - fixed randomize/rnd bug, get bug
    v3.6.3b0 - added sdbm database commands
    v3.6.2b9 - fixed bug in atn() with expression parameter
                    - added fn version$(), open pipe for output
    v3.6.1b2 - fixed bug in integer LET command
    v3.6.1   - changed precedence of unary minus (-) to below
                exponentiation (^) to match ANSI/ISO specification.
             - changed default matrix origin to 1
             - added mat print commmand.
    v3.6.0   - added unix exit status, fn kill(), fn bigendian()
    v3.5.9b4 - fixed def fn array parameters, added atn(y,x)
    v3.5.9b3 - fixed bugs with if():, instr(), mid$ & val("..")
    v3.5.9   - fixed bugs in format$() & intl. string comparisons
    v3.5.8b7 - fixed  bugs in array assignment, the read/data
               statement, fputbyte(), and ELSEIF nesting
             - added the MAT INVERT statement
    v3.5.8   - changed int() to round towards -infinity.
    v3.5.7b3 - added network socket i/o
    v3.5.7b2 - fixed unix/linux rnd() function
    v3.5.7b1 - fixed a problem with array indexes > 65k
    v3.5.3  - allow integer (i%) variables as for/next loop indices
    v3.4.7  - lower precedence of NOT operator
            - disabled ON GOTO range checking
    v3.4.6  - added MAT matrix statements
    v3.4.0  - OPEN ELSE added
    v3.3.4  - changed integer conversion to rounding
            - changed sub return values to: sub_name = x
            - added reserved words for: true false
    v3.3.3  - added acos, tanh, log10
    v3.2.8  - added class definitions

    Many others ...
```

BUGS

Many. Perhaps competitive with Central American rain forests.

FOR/NEXT loops with integer indices require a variable in the NEXT statement. Integer arrays can only have a dimension of one and will only work in assignment (LET) statements. All arithmetic on integer variables is done using floating point arithmetic. DIM AS DOUBLE and DIM AS INTEGER statements are ignored.

Many string functions (except +, MID$, LEN and INSTR) silently truncate their results to 254 characters (e.g. without warning). All string function may silently truncate strings longer than 32766 characters. Any operation on strings longer than 254 characters will cause the program to run slower.

Comments starting with ' sometimes can't be used after statements that can end with a string parameter. (: ' should always work.)

Any variables used as a CLASS, or TYPE, globally overide all local variables of the same names. Local TYPE'd variables must be declared globally as TYPE'd variables. Named SUBroutines are slower than GOSUBs. The combined length of a SUBroutine name and any local variables declared STATIC must be less than 29 characters.

Can't CONTinue from an error inside a named SUB subroutine.

The PRINT USING format string doesn't recognize comma's, underscores and many other common format characters.

Macintosh screen editing will only recognise the last line modified before a RETURN or ENTER key. The EDIT command and Mac screen editing are incompatible.

There are many undocumented graphics and sprite commands and keywords in the Macintosh port. See the accompanying README and Chipmunk Basic quick-reference file.

DISCLAIMER

There is no warranty that this document is accurate.

SEE ALSO

    The Chipmunk Basic Home Page:

            http://www.nicholson.com/rhn/basic

AUTHORS

    David Gillespie wrote basic.p 1.0 and the p2c lib.