Yabasic

Table of Contents

I

if — evaluate a condition and execute statements or not, depending on the result

import — import a library

inkey$ — wait, until a key is pressed

input — read input from the user (or from a file) and assign it to a variable

instr() — searches its second argument within the first; returns its position if found

int() — return the integer part of its single numeric argument

L

label — mark a specific location within your program for `goto`, `gosub` or `restore`

left$() — return (*or change*) left end of a string

len() — return the length of a string

line — draw a line

line input — read in a whole line of text and assign it to a variable

local — mark a variable as local to a subroutine

log() — compute the natural logarithm

loop — marks the end of an infinite loop

lower$() — convert a string to lower case

ltrim$() — trim spaces at the left end of a string

M

max() — return the larger of its two arguments

mid$() — return (*or change*) characters from within a string

min() — return the smaller of its two arguments

mod — compute the remainder of a division

mouseb — extract the state of the mousebuttons from a string returned by `inkey$`

mousemod — return the state of the modifier keys during a mouseclick

mousex — return the x-position of a mouseclick

mousey — return the y-position of a mouseclick

N

new curve — start a new curve, that will be drawn with the `line`-command

next — mark the end of a for loop

not — negate a logical expression; can be written as `!`

numparams — return the number of parameters, that have been passed to a subroutine

Chapter 1. Introduction

About this document

This document describes yabasic. You will find information about the yabasic interpreter (the program **`yabasic`** under Unix or **`yabasic.exe`** under Windows) as well as the language (which is, of course, a sort of basic) itself.

This document applies to version **2.90** of yabasic

However, it does *not* contain the latest news about yabasic or a FAQ. As such information tends to change rapidly, it is presented online only at www.yabasic.de.

Although basic has its reputation as a language for beginning programmers, this is not an introduction to programming at large. Rather this text assumes, that the reader has some (moderate) experience with writing and starting computer programs.

About yabasic

yabasic is a traditional basic interpreter. It understands most of the typical basic-constructs, like `goto`, `gosub`, line numbers, `read`, `data` or string-variables with a trailing '`$`'. But on the other hand, yabasic implements some more advanced programming-constructs like subroutines or libraries (but *not* objects). yabasic works much the same under Unix and Windows.

yabasic puts emphasis on giving results quickly and easily; therefore simple commands are provided to open a graphic window, print the graphics or control the console screen and get keyboard or mouse information. The example below opens a window, draws a circle and prints the graphic:

```
open window 100,100
open printer
circle 50,50,40
text 10,50,"Press any key to get a printout"
clear screen
inkey$
close printer
close window
```

This example has fewer lines, than it would have in many other programming languages. In the end however yabasic lacks behind more advanced and modern programming languages like C++ or Java. But as far as it goes it tends to give you results more quickly and easily.

Chapter 2. The yabasic-program under Windows

Starting yabasic

Once, yabasic has been set up correctly, there are three ways to start it:

1. *Right click on your desktop:* The desktop menu appears with a submenu named *new*. From this submenu choose yabasic. This will create a new icon on your desktop. If you right click on this icon, its context menu will appear; choose **Execute** to execute the program.

2. As a variant of the way described above, you may simply *create a file with the ending* `.yab` (e.g. with your favorite editor). Everything else then works as described above.

3. *From the start-menu:* Choose yabasic from your start-menu. A console-window will open and you will be asked to type in your program. Once you are finished, you need to type `return` twice, and yabasic will parse and execute your program.

> **Note**
>
> This is *not* the preferred way of starting yabasic ! Simply because the program, that you have typed, *can not be saved* and will be lost inevitably ! There is no such thing as a `save`-command and therefore no way to conserve the program, that you have typed. This mode is only intended for quick hacks, and short programs.

Options

Under Windows yabasic will mostly be invoked by double-clicking on an appropriate icon; this way you do not have a chance to specify any of the command line options below. However, advanced users may change the librarypath in the registry, which has the same effect as specifying it as an option on the command line.

See the chapter on options for a complete list of all options, either on Unix or Windows.

The context Menu

Like every other icon under Windows, the icon of every yabasic-program has a *context menu* offering the most frequent operations, that may be applied to a yabasic-program.

Execute

This will invoke yabasic to execute your program. The same happens, if you *double click* on the icon.

Edit

notepad will be invoked, allowing you to edit your program.

View docu

This will present the embedded documentation of your program. Embedded documentation is created with the special comment `doc`.

Chapter 3. The yabasic-program under Unix

Starting yabasic

If your system administrator (vulgo *root*) has installed yabasic correctly, there are three ways to start it:

1. You may use your favorite editor (emacs, vi ?) to put your program into a file (e.g. `foo`). Make sure that the very first line starts with the characters '`#!`' followed by the full pathname of yabasic (e.g. '`#!/usr/local/bin/yabasic`'). This *she-bang*-line ensures, that your Unix will invoke yabasic to execute your program (see also the entry for the hash-character). Moreover, you will need to change the permissions of your yabasic-program `foo`, e.g. `chmod u+x foo`. After that you may invoke yabasic to invoke your program by simply typing `foo` (without even mentioning yabasic). However, if your `PATH`-variable does not contain a single dot ('.') you will have to type the full pathname of your program: e.g. `/home/ihm/foo` (or at least `./foo`).

2. Save your program into a file (e.g. `foo`) and type `yabasic foo`. This assumes, that the directory, where yabasic resides, is contained within your `PATH`-variable.

3. Finally your may simply type **`yabasic`** (maybe it will be necessary to include its full pathname). This will make yabasic come up and you will be asked to type in your program. Once you are finished, you need to type `return` twice, and yabasic will parse and execute your program.

> **Note**
> This is *not* the preferred way of starting yabasic ! Simply because the program, that you have typed, *can not be saved* and will be lost inevitably ! There is no such thing as a `save`-command and therefore no way to conserve the program, that you have typed. This mode is only intended for quick hacks, and short programs, i.e. for using yabasic as some sort of fancy desktop calculator.

Options

yabasic accepts a number of options on the command line.

See chapter on options for a complete list of all options, either on Unix or Windows.

Setting defaults

If you want to set some options *once for all*, you may put them into your X-Windows resource file. This is usually the file `.Xresources` or some such within your home directory (type **`man X`** for details).

Here is a sample section, which may appear within this file:

```
yabasic*foreground: blue
yabasic*background: gold
yabasic*geometry: +10+10
yabasic*font: 9x15
```

This will set the foreground color of the graphic-window to *blue* and the background color to *gold*. The window will appear at position *10,10* and the text font will be *9x15*.

Chapter 4. Command line options of yabasic

Here are the options, that yabasic accepts on the command line (both under Unix and Windows).

All the options below may be abbreviated (and one hyphen may be dropped), as long as the abbreviation does not become ambiguous. For example, you may write `-e` instead of `--execute`.

`--help` or `-?`

Prints a short help message, which itself describes two further help-options.

`--version`

Prints the version of yabasic.

`--infolevel` *`INFOLEVEL`*

Change the *infolevel* of yabasic, where *`INFOLEVEL`* can be one of `debug`, `note`, `warning`, `error`, `fatal` and `bison` (the default is `warning`). This option changes the amount of debugging-information yabasic produces. However, normally only the author of yabasic (*me* !) would want to change this.

`--execute` *`A-PROGRAM-AS-A-SINGLE-STRING`*

With this option you may specify some yabasic-code to be executed *right away*. This is useful for very short programs, which you do not want to save to a file. If this option is given, yabasic will not read any code from a file. E.g.

```
yabasic -e 'for a=1 to 10:print a*a:next a'
```

prints the square numbers from 1 to 10.

--bind *NAME-OF-STANDALONE-PROGRAM*

Create a standalone program (whose name is specified by *NAME-OF-STANDALONE-PROGRAM*) from the yabasic-program, that is specified on the command line. See the section about creating a *standalone*-program for details.

--geometry *+X-POSITION+Y-POSITION*

Sets the position of the graphic window, that is opened by `open window` (the *size* of this window, of course, is specified within the `open window`-command). An example would be `-geometry +20+10`, which would place the graphic window 10 pixels below the upper border and 20 pixels right of the left border of the screen. This value cannot be changed, once yabasic has been started.

-fg *FOREGROUND-COLOR* or --foreground *FOREGROUND-COLOR*

Unix only. Define the foreground color for the graphics-window (that will be opened with `open window`). The usual X11 color names, like *red*, *green*, ... are accepted. This value cannot be changed, once yabasic has been started.

-bg *BACKGROUND-COLOR* or --background *BACKGROUND-COLOR*

Unix only. Define the background color for the graphics-window. The usual X11 color names are accepted. This value cannot be changed, once yabasic has been started.

--display *X11-DISPLAY-SPECIFICATION*

Unix only. Specify the *display*, where the graphics window of yabasic should appear. Normally this value will be already present within the environment variable `DISPLAY`.

--font *NAME-OF-FONT*

Under Unix. Name of the font, which will be used for text within the graphics window.

--font *NAME-OF-FONT*

Under Windows. Name of the font, which will be used for graphic-text; can be any of `decorative, dontcare, modern, roman, script, swiss`. You may append a fontsize (measured in pixels) to any of those fontnames; for example `-font swiss30` chooses a swiss-type font with a size of 30 pixels.

--docu *NAME-OF-A-PROGRAM*

Print the *embedded documentation* of the named program. The embedded documentation of a program consists of all the comments within the program, which start with the special keyword `doc`. This documentation can also be seen by choosing

the corresponding entry from the context-menu of any yabasic-program.

--check

Check for possible compatibility problems within your yabasic-program. E.g. this option reports, if you are using a function, that has recently changed.

--librarypath *DIRECTORY-WITH-LIBRARIES*

Change the directory, wherein libraries will be searched and imported (with the `import`-command). See also `import` for more information about the way, libraries are searched.

--

Do not try to parse any further options; rather pass the subsequent words from the commandline to yabasic.

Chapter 5. All commands and functions of yabasic listed by topic

Numbers with base 2 or 16

In addition to the usual decimal notation (e.g. `1234`), yabasic also supports numeric literals with base 2 or 16; examples are `0b10011` (the number 19, written with base 2) or `0x34AF` (the number 13487, written with base 16) respectively. Please note that these numbers (apart from the way you write them into your program) are no different from

“ordinary” numbers and can be used in any place, where a normal number with base 10 would fit. E.g. you may compute the sine `sin(0b110)`; so the base 2 (or 16) is just a different way of *representation*.

See `bin$`, `hex$` or `dec` for related functions.

Number processing and conversion

abs()
: returns the absolute value of its numeric argument

acos()
: returns the arcus cosine of its numeric argument

and()
: the bitwise arithmetic and

asin()
: returns the arcus sine of its numeric argument

atan()
: returns the arctangent of its numeric argument

bin$()
: converts a number into a sequence of binary digits

cos()
: return the cosine of its single argument

dec()
: convert a base 2 or base 16 number into decimal form

eor()
: compute the bitwise exclusive or of its two arguments

euler
: another name for the constant 2.71828182864

exp()
: compute the exponential function of its single argument

frac()
: return the fractional part of its numeric argument

int()
: return the integer part of its single numeric argument

ceil()
: compute the ceiling for its (float) argument

floor()
: compute the floor for its (float) argument

round()
: round its argument to the nearest integer

log()
: compute the natural logarithm

max()
: return the larger of its two arguments

min()
: return the smaller of its two arguments

mod
compute the remainder of a division

bitnot()
the bitwise arithmetic not

or()
arithmetic or, used for bit-operations

pi
a constant with the value 3.14159

ran()
return a random number

shl()
shift its argument bitwise to the left

shr()
shift its argument bitwise to the right

sig()
return the sign of its argument

sin()
return the sine of its single argument

sqr()
compute the square of its argument

sqrt()
compute the square root of its argument

tan()
return the tangent of its argument

xor()
compute the exclusive or

**** or ^**
raise its first argument to the power of its second

< <= > >= = == <> !=
Compare numbers or strings

Conditions and control structures

and
logical and, used in conditions

break
breaks out of one or more loops or switch statements

case
mark the different cases within a switch-statement

continue
start the next iteration of a for-, do-, repeat- or while-loop

default
mark the default-branch within a switch-statement

do
start a (conditionless) do-loop

else

mark an alternative within an if-statement

elsif
starts an alternate condition within an if-statement

end
terminate your program

endif
ends an if-statement

false
a constant with the value of 0

fi
another name for endif

for
starts a for-loop

gosub
continue execution at another point within your program (and return later)

goto
continue execution at another point within your program (and never come back)

if
evaluate a condition and execute statements or not, depending on the result

label
mark a specific location within your program for goto, gosub or restore

loop
marks the end of an infinite loop

next
mark the end of a for loop

not
negate a logical expression; can be written as !

on gosub
jump to one of multiple gosub-targets

on goto
jump to one of many goto-targets

on interrupt
change reaction on keyboard interrupts

logical or
logical or, used in conditions

bitwise or
arithmetic or, used for bit-operations

pause
pause, sleep, wait for the specified number of seconds

repeat
start a repeat-loop

return

return from a subroutine or a gosub

sleep
pause, sleep, wait for the specified number of seconds

step
specifies the increment step in a for-loop

switch
select one of many alternatives depending on a value

then
tell the long from the short form of the if-statement

true
a constant with the value of 1

until
end a repeat-loop

wait
pause, sleep, wait for the specified number of seconds

wend
end a while-loop

while
start a while-loop

:
separate commands from each other

Data keeping and processing

arraydim()
returns the dimension of the array, which is passed as an array reference

arraysize()
returns the size of a dimension of an array

data
introduces a list of data-items

dim
create an array prior to its first use

read
read data from data-statements

redim
create an array prior to its first use. A synonym for dim

restore
reposition the data-pointer

String processing

asc()
accepts a string and returns the position of its first character within the ascii charset

chomp$()
remove a single trailing newline from its string-argument; if the

string does not end in a newline, the string is returned unchanged

chr$()
accepts a number and returns the character at this position within the ascii charset

glob()
check if a string matches a simple pattern

hex$()
convert a number into hexadecimal

instr()
searches its second argument within the first; returns its position if found

rinstr()
find the rightmost occurrence of one string within the other

left$()
return (or change) left end of a string

len()
return the length of a string

lower$()
convert a string to lower case

ltrim$()
trim spaces at the left end of a string

mid$()
return (or change) characters from within a string

right$()
return (or change) the right end of a string

split()
split a string into many strings

str$()
convert a number into a string

token()
split a string into multiple strings

trim$()
remove leading and trailing spaces from its argument

rtrim$()
trim spaces at the right end of a string

upper$()
convert a string to upper case

val()
converts a string to a number

File operations and printing

at()
can be used in the print-command to place the output at a specified position

beep

ring the bell within your computer; a synonym for bell

bell
ring the bell within your computer (just as beep)

clear screen
erases the text window

close
close a file, which has been opened before

close printer
stops printing of graphics

print color
print with color

print colour
see print color

eof
check, if an open file contains data

getscreen$()
returns a string representing a rectangular section of the text terminal

inkey$
wait, until a key is pressed

input
read input from the user (or from a file) and assign it to a variable

line input
read in a whole line of text and assign it to a variable

open
open a file

open printer
open printer for printing graphics

print
Write to terminal or file

putscreen
draw a rectangle of characters into the text terminal

reverse
print reverse (background and foreground colors exchanged)

screen
as clear screen clears the text window

seek()
change the position within an open file

tell
get the current position within an open file

using
Specify the format for printing a number

#
either a comment or a marker for a file-number

at

can be used in the print-command to place the output at a specified position

@
synonymous to at

;
suppress the implicit newline after a print-statement

Subroutines

end sub
ends a subroutine definition

local
mark a variable as local to a subroutine

numparams
return the number of parameters, that have been passed to a subroutine

return
return from a subroutine or a gosub

static
preserves the value of a variable between calls to a subroutine

sub
declare a user defined subroutine

Libraries

export
mark a function as globally visible

import
import a library

Invoking other program from within yabasic

system()
hand the name of an external command over to your operating system and return its exitcode

system$()
hand the name of an external command over to your operating system and return its output

Adding new code to a running program

See also adding code during execution.

compile
compile a string with yabasic-code on the fly

eval()
compile and execute a single numeric expression

eval$()

compile and execute a single string-expression

execute()
execute a user defined subroutine, which must return a number

execute$()
execute a user defined subroutine, which must return a string

Commands and functions related with time

date$
returns a string with various components of the current date

pause
pause, sleep, wait for the specified number of seconds

sleep
pause, sleep, wait for the specified number of seconds

time$
return a string containing the current time

Other commands

bind()
binds a yabasic-program and the yabasic-interpreter together into a standalone program

doc
special comment, which might be retrieved by the program itself

docu$
special array, containing the contents of all docu-statement within the program

error
raise an error and terminate your program

exit
terminate your program

peek
retrieve various internal information

peek$
retrieve various internal string-information

poke
change selected internals of yabasic

rem
start a comment

to
this keyword appears as part of other statements

wait
pause, sleep, wait for the specified number of seconds

//
starts a comment

:
separate commands from each other

Graphics and printing

backcolor
change color for background of graphic window

box
draw a rectangle. A synonym for rectangle

circle
draws a circle in the graphic-window

clear
erase circles, rectangles or triangles

clear window
clear the graphic window and begin a new page, if printing is under way

close curve
close a curve, that has been drawn by the line-command

close window
close the graphics-window

color
change color for any subsequent drawing-command

dot
draw a dot in the graphic-window

fill
draw a filled circles, rectangles or triangles

getbit$()
return a string representing the bit pattern of a rectangle within the graphic window

line
draw a line

mouseb
extract the state of the mousebuttons from a string returned by inkey$

mousemod
return the state of the modifier keys during a mouseclick

mousex
return the x-position of a mouseclick

mousey
return the y-position of a mouseclick

new curve
start a new curve, that will be drawn with the line-command

open window
open a graphic window

putbit
draw a rectangle of pixels encoded within a string into the graphics window

rectangle
draw a rectangle

The foreign function interface

Chapter 6. Some features and general concepts of yabasic

This chapter presents some general concepts and terms, which deserve a description on their own, but are not associated with a single command or function in yabasic. Most of these topics do not lend themselves to be read alone, rather they might be read (or

skimmed) as background material if an entry from the alphabetical list of commands refers to them.

Logical shortcuts

Logical shortcuts are no special language construct and there is no keyword for them; they are just a way to evaluate *logical expressions*. Logical expressions (i.e. a series of conditions or comparisons joined by `and` or `or`) are only evaluated until the final result of the expression can be determined. An example:

```
if (a<>0 and b/a>2) print "b is at least twice as big as a"
```

The logical expression `a<>0 and b/a>2` consists of two comparisons, both of which must be true, if the `print` statement should be executed. Now, if the first comparison (`a<>0`) is `false`, the whole logical expression can never be `true` and the second comparison (`b/a>2`) need not be evaluated.

This is exactly, how yabasic behaves: The evaluation of a composed logical expressions is terminated immediately, as soon as the final result can be deduced from the already evaluated parts.

In practice, this has the following consequences:

- If two or more comparisons are joined with `and` and one comparison results in `false`, the logical expression is evaluated no further and the overall result is `false`.
- If two or more comparisons are joined with `or` and one comparison results in `true`, the logical expression is evaluated no further and the result is `true`.

“Nice, but whats this good for ?”, I hear you say. Well, just have another look at the example, especially the second comparison (`b/a>2`); dividing `b` by `a` is potentially hazardous: If `a` equals zero, the expression will cause an error and your program will terminate. To avoid this, the first part of the comparison (`a<>0`) checks, if the second one can be evaluated without risk. This pre-checking is the most common usage and primary motivation for *logical shortcuts* (and the reason why most programming languages implement them).

Conditions and expressions

Well, bottomline there is no difference or distinction between *conditions* and *expressions*, at least as yabasic is concerned. So you may assign the result of comparisons to variables or use an arithmetic expression or a simple variable within a condition (e.g. within an `if`-statement). So the constructs shown in the example below are all totally valid:

```
input "Please enter a number between 1 and 10: " a
```

```
rem   Assigning the result of a comparison to a variable
okay=a>=1 and a<=10

rem   Use a variable within an if-statement
if (not okay) error "Wrong, wrong !"
```

So conditions and expressions are really the same thing (at least as long as yabasic is concerned). Therefore the terms *conditions* and *expression* can really be used interchangeably, at least in theory. In reality the term *condition* is used in connection with `if` or `while` whereas the term *expression* tends to be used more often within arithmetic context.

Comparing strings or numbers

Yabasic, of course, allows to compare strings with strings and numbers with numbers; `<`, `<=`, `>` and `>=` compare their left-hand side to their right-hand side as usual; nothing new here and examples can be found throughout this manual.

More interesting, the *equality*-operator (for numbers as well as for strings) can be written *in two different ways*: either as `=` (traditional) or as `==` (more modern). The second form has the advantage of beeing visually distinct from the *assignment*-operator, which is the single `=`. One may argue therefore, that using `==` results in code, that is easier to understand and read; This manual however sticks to tradition and mostly uses the single `=` for equality-check.

Finally, *inequality* can be checked with `<>` or `!=`; both operators behave identically and so it is only a matter of taste, which one to use.

References on arrays

References on arrays are the only way to refer to an array *as a whole* and to pass it to subroutines or functions like `arraydim` or `arraysize`.

While (for example) `a(2)` designates the second element of the array `a`, `a()` (with empty braces) refers to the array `a` itself. `a()` is called an *array reference*. A nice example is the bultin function `split`, that accepts an array-reference and modifies the content of this array.

You may also pass to and use array reference within your own subroutines; these subroutines will then be able to modify the array you have passed in often this is intended.

Passing an array reference does not create a copy of the array; this has some interesting consequences:

- *Speed and space:* Creating a copy of an array would be a time and memory consuming operation; passing just a reference is cheap and fast.
- Returning many values: A subroutine, that wants to give back more than one value, may require an array reference among

its arguments and then store its many return values within this array. This is the only way to return more than one value from a subroutine.

An example

The following program creates two subroutines (`print_words` and `upcase_words`), that operate on an array of words (`words$()` below):

```
dim words$(4)
for i=1 to 4
  read words$(i)
next i

print_words(words$())
upcase_words(words$())
print_words(words$())

sub print_words(w$())
  local i
  for i=1 to arraysize(w$(),1)
    print w$(i)," ";
  next i
  print
end sub

sub upcase_words(w$())
  local i
  for i=1 to arraysize(w$(),1)
    w$(i) = upper$(w$(i))
  next i
end sub

data "case","does","not","matter"
```

If you run this program, you will get this output:

```
case does not matter
CASE DOES NOT MATTER
```

Specifying Filenames under Windows

As you probably know, windows uses the character '`\`' to separate the directories within a pathname; an example would be `C:\yabasic\yabasic.exe` (the usual location of the yabasic executable). However, the very same character '`\`' is used to construct escape sequences, not only in yabasic but in most other programming languages.

Therefore the string `"C:\t.dat"` does *not* specify the file `t.dat` within the directory `C:`; this is because the sequence '`\t`' is translated into the tab-character. To specify this filename, you need to use the string `"C:\\t.dat"` (note the double slash '`\\`').

Escape-sequences

Escape-sequences are the preferred way of specifying 'special' characters. They are introduced by the '\'-character and followed by one of a few regular letters, e.g. '\n' or '\r' (see the table below).

Escape-sequences may occur within any string at any position; they are replaced at *parsetime* (opposed to *runtime*), i.e. as soon as yabasic discovers the string, with their corresponding *special* character. As a consequence of this `len("\a")` returns 1, because yabasic replaces `"\a"` with the matching special character just before the program executes.

Table 6.1. Escape sequences

Escape Sequence	Matching special character
`\n`	*newline*
`\t`	*tabulator*
`\v`	*vertical tabulator*
`\b`	*backspace*
`\r`	*carriage return*
`\f`	*formfeed*
`\a`	*alert* (i.e. a beeping sound)
`\\`	*backslash*
`\'`	*single quote*
`\"`	*double quote*
`\x`*HEX*	`chr$(`*HEX*`)` (see below)

Note, that an escape sequences of the form \xHEX allows one to encode arbitrary characters as long as you know their position (as a hex-number) within the ascii-charset: For example \x012 is transformed into the character `chr$(18)` (or `chr$(dec("12",16))`. Note that \x requires a hexa-decimal number (and the hexadecimal string "12" corresponds to the decimal number 18).

Subroutines: Sharing code within one program

Purpose

Nobody wants to repeat oneself and therefore yabasic allows to collect arbitrary code into *subroutines*, so that you may call it from *multiple* locations within you program. To this end, two conditions must be fulfilled:

1. The subroutine neeeds to know details about what to do; that's why subroutines have *parameters*. E.g. in the overly simple subroutine `sub add(a,b)` (see the example below) the parameters

would be `a` and `b`, specifying, which numbers to add.

Remark: In certain cases a subroutine may want to find out, how many parameters it has been called with, by querying the special variable `numparams`.

2. The subroutine needs to run without messing up the state of the program, at the point where it has been called. That's why many subroutines use `local` variables, which are different and isolated from all other variables in your program, even if they happen to have the same name. *parameters* (as described above) are, in addition to their primary function, also local variables.

   Remark: If a subroutine wants to remember some information between invocations, it may declare some of its variables as `static` instead of `local` function

To see these concepts explained in more detail (complete with examples), follow the links at the end of this section.

Remark: You may notice, that other programming language may use other terms than subroutine for the same concept: `function` or `procedure` have been popular for pieces of code, that either return a value or not, and in other languages `def` is used to name both. And the term `method` is used in object-oriented languages. However yabasic is not object oriented, and regardless, if a piece of code produces a value or not, it can be encapsulated in a subroutine, so yabasic uses the function `sub` throughout.

A simple example

The short program below does nothing more than to add two numbers; for this purpose, it even defines a subroutine. This admittedly is more overhead, than you would normally take.

```
print "About to add two numbers."
input "Please enter first number: " x
input "Please enter second number: " y
print "Their sum is: ", add(x,y)

sub add(a,b)
  return a+b
end sub
```

If you run it, you would see:

```
About to add two numbers.
Please enter first number: 2
Please enter second number: 3
Their sum is: 5
```

Again, see the link at the end of this section for more explanations

and examples (e.g. on `local` or `static`, which have only been mentioned but not shown at work so far).

See also

All commands for subroutines, where you will find links to the individual keywords related.

Libraries: Sharing code between many programs

Purpose

Libraries build upon `subroutines` and take the concept of code-reuse one step further: They allow code to be shared between *different* programs (as compared to subroutines, which on their own allow code-reuse within a single program only). Moreover, it is possible and in fact common, that the author of a library and the author of a program using that library, are different persons, each writing their respective code on their own.

A simple example

Here is a program, that asks the user for two numbers and then uses a library `adder` to add those:

```
import adder

print "About to add two numbers."
input "Please enter first number: " x
input "Please enter second number: " y
print "Their sum is: ", adder.add(x,y)
```

The statement `import adder` pulls in code from a very simple *library* `adder.yab`:

```
sub add(a,b)
  return a+b
end sub
```

If you run it, you might see:

```
About to add two numbers.
Please enter first number: 3
Please enter second number: 4
Their sum is: 7
```

Compared with the very similar example for subroutines, there are two differences:

1. The code of the subroutine `add` has been moved to its own file `adder.yab`.

2. The subroutine `add` needs to be called as `adder.add`, which consists of filename (`adder.yab` but without the ending `.yab`) and the name of the subroutine (`add`) within that file.

This is an example of *namespaces*.

Namespaces

When the executable code is devided between a *main program file* and (multiple) *libraries* it is important to keep their subroutines and variables seperate. To this end yabasic internally prefixes the subroutines and variables defined in a library with the shortened name of the library. E.g. in the example above, the subroutine `add` from the library `adder.yab` is prefixed by this library-name and ends up as beeing defined as `adder.add`.

Subroutines and variables defined within the main program are prefixed with `main`; this prefix is fixed and not related to the actual filename of the main-program. Normally, however, there is no need to use this prefix explicitly; it only helps yabasic to keep everything apart.

See also

`import`, `export`, subroutines

Adding code to a running program

Purpose

Normally, you write programs in yabasic and specify all the necessary logic and calculations within your program. Once you are done, you invoke it, probably multiple times; and while it is running, it does not change.

However, there are some commands within yabasic, that allow to blur the line between writing and execution. Namely `eval`, `eval$`, `compile`, `execute` and `execute$` allow to create and execute new yabasic-code while your program is running. This comes in handy, if the code to be used comes from the user of your program and will only be known *after* your program has started. A simple example is the yabasic-code to calculate the maximum of a user-supplied expression within a given range; find it as an example for `eval`. In the same way, one may write a program to plot an arithmetic function, whose definition is entered by the user.

Note: Even if the commands listed above allow to change the yabasic-program, that is currently running, the *file* where the program is stored, does not change. Therefore, the changes to the running program are not permanent.

How the various functions and commands differ

The most simple functions are `eval` and `eval$`; they compile an expression (with a numeric or string result), e.g. and execute it right away. The compiled code is remembered, so that it need not be compiled again, when the sames expression is executed again; this caters efficiency. However these functions only accept a *single expression* and nothing else.

If you need more complex computation and logic, the process needs to be split: First create a new subroutine with the `compile`-command, then execute this subroutine (maybe multiple times) via `execute` or `execute$`. This allows to use the broad logic available in subroutines (e.g. conditions, loops, local variables or even other subroutines) and therefore much more complex calculations than with `eval`. If you want to use `compile` multiple times within your program (e.g. in a loop), you may want to enumerate the functions you create to avoid name-clashes (as shown in the examples of `compile`).

To invoke the subroutines created, you need to execute them with `execute` or `execute$`, which require the name of the function (a string) as their first argument.

Creating a *standalone* program from your yabasic-program

Sometimes you may want to give one of your yabasic-programs to other people. However, what if those other people do not have yabasic installed ? In that case you may create a *standalone*-program from your yabasic-program, i.e. an executable, that may be executed on its own, standalone, even (and especially !) on computers, that do not have yabasic installed. Having created a standalone program, you may pass it around like any other program (e.g. one written in C) and you can be sure that your program will execute right away.

Such a standalone-program is simply created by copying the full yabasic-interpreter and your yabasic-program (plus all the libraries that it may `import`) together into a single, new program, whose name might be chosen at will (under windows of course it should have the ending `.exe`). If you decide to create a standalone-program, there are three facilities in yabasic, that you may use:

- The `bind`-command, which does the actual job of creating the standalone program from the yabasic-interpreter and your program.
- The command-line Option `--bind` (see options), which does the same from the command-line.
- The special `peek("isbound")`, which may be used to check, if the yabasic-program containing this `peek` is bound to the interpreter as part of a standalone program.

With these bits you know enough to create a standalone-program. Actually there are two ways to do this: on the command line and from within your program.

Creating a standalone-program from the command line

Let's say you have the following *very* simple program within the file `foo.yab`:

```
print "Hello World !"
```

Normally you would start this yabasic-program by typing **`yabasic foo.yab`** and as a result the string `Hello World !` would appear on your screen. However, to create a *standalone*-program from `foo.yab` you would type:

`yabasic -bind foo.exe foo.yab`

This command does *not* execute your program `foo.yab` but rather create a *standalone*-program `foo.exe`. Note: under Unix you would probably name the standalone program `foo` or such, omitting the windows-specific ending `.exe`.

Yabasic will confirm by printing something like: `---Info: Successfully bound 'yabasic' and 'foo.yab' into 'foo.exe'.`

After that you will find a program `foo.exe` (which must be made *executable* with the **`chmod`**-command under Unix first). Now, executing this program `foo.exe` (or `foo` under Unix) will produce the output `Hello World !.`

This newly created program `foo.exe` might be passed around to anyone, even if he does not have yabasic installed.

Creating a standalone-program from within your program

It is possible to write a yabasic-program, that binds itself to the yabasic-interpreter. Here is an example:

```
if (!peek("isbound")) then
  bind "foo"
  print "Successfully created the standalone executable 'foo' !"
  exit
endif

print "Hello World !"
```

If you run this program (which may be saved in the file `foo.yab`) via **`yabasic foo.yab`**, the `peek("isbound")` in the first line will check, if the program is already part of a standalone-program. If *not* (i.e. if the yabasic-interpreter and the yabasic-program are separate files) the `bind`-command will create a standalone program `foo` containing both. As a result you would see the output `Successfully created the standalone executable 'foo' !`. Note: Under Windows you would probably choose

the filename `foo.exe`.

Now, if you run this standalone executable `foo` (or `foo.exe`), the very same yabasic-program that is shown above will be executed again. However, this time the `peek("isbound")` will return `TRUE` and therefore the condition of the `if`-statement is *false* and the three lines after `then` are *not* executed. Rather the last `print`-statement will run, and you will see the output `Hello World !`.

That way a yabasic-program may turn itself into a standalone-program.

Points to consider before creating a standalone program

- The new standalone program will be at least as big as the interpreter itself, which is typically a few hundred kilobytes.
- There is no easy way to extract your yabasic-program from within the standalone program: If you ever want to change it, you should keep it around as a separate file.
- If a new version of yabasic becomes available, you might want to recreate your standalone program to take advantage of bugfixes and improvements.

See also

The `bind`-command, the `peek`-function and the command line options.

Interaction with functions from a non-yabasic library or dll

> **Note**
>
> Under Unix, depending on the way yabasic has been built, this feature might have been disabled; the error message in this case will read like `this build of yabasic does not support calling foreign libraries`. To resolve this issue, you are invited to contact the maintainer.

> **Note**
>
> This is interesting, but somewhat advanced stuff. You will need a good understanding of various concepts of the C-language, especially *pointers* and *structures* as well as allocating and freeing blocks of memory. Please be aware, that mistakes or errors during calls to foreign functions or buffers may easily crash yabasic.

Yabasic allows to employ functionality from an external library; i.e. from a library, which is *not* written in yabasic, but rather in C; such a library is called a *foreign* library, as opposed to a library written in yabasic itself. Calling out to a foreign library can be useful, if such a library provides functionality, that can not be replicated in yabasic itself and for which a commandline-interface (which could be used via `system`) does not exist or is too cumbersome or slow. Examples would be libraries libVLC or libcurl which offer the functionality of vlc or curl to other programs, especially programs written in yabasic.

The foreign function interface of yabasic relies on the great libffi-library, a library making it easy to call other libraries dynamicaaly and the established standard for this task.

Some Background

Libraries

Libraries (e.g. libcurl) are meant to provide functionality to other programs (here: yabasic and your yabasic-program). Libraries and programs must be *linked* together; this can happen either *statically* at compile-time or *dynamically* during the excution of the program. For yabasic as the program, static linking happens at the time, yabasic itself is build, whereas dynamic linking happen during the execution *and under control* of your yabasic-program. So the foreign function interface deals with dynamic (or *runtime*) linking to external libraries. This linking is done by yabasic behind the scene, when you invoke `foreign_function_call`; this function, after loading the library, directly calls the specified function therein.

Which functions are available differs from library to library and you should already have this information before you try the library with yabasic.

Types

If you want to use a function from a foreign library, you will need to deal with the fact, that each function requires several parameters and returns exactly one. These parameters have a wide variety of types which need to be mapped to the two types (numbers and strings) known by yabasic. Here are the types available for foreign functions, grouped by the way, they are handled in yabasic:

uint8,int8,uint16,int16,uint32,int32,uint64,int64,float,double,char,short,int,long

In C all these types are used to represent numbers (integer or floating point) with various degrees of precision. When invoking foreign_function_call you need to pass strings, which specify the right type as well as the actual yabasic-value, which will then be converted accordingly. Which types a foreign function expects can be looked up e.g. from its manpage.

string

Strings for foreign functions directly map to strings of yabasic. So if you specify this type for a parameter of a foreign function, the matching value is simply a yabasic string. If you specify `string` as the return value of a foreign function you should use the variant of calling it, which returns a string, i.e. foreign_function_call$.

buffer

You should specify a *buffer* as the type of a parameter or the return type, if the foreign function expects a structure or a pointer to a memory area; see structures and buffers for details.

Three simple examples

Computing the cosine

This first example prints the cosine of 2, not by using yabasics own `cos`-function but by calling out to the standard C-library:

```
if peek$("os")="windows" then
  lib$ = "msvcrt.dll"
else
  lib$ = "libm.so.6"
endif

print "cos(2): ",foreign_function_call(lib$,"double","cos","double",2)
```

The first lines determine the name of the library, which is different under Unix and Windows. The call to foreign_function_call than just states the name of the library, the return type (`"double"`) of the function and then its name (`"cos"`), as well as type and value (`2`) of its argument. The final result `-0.416147` then is the same as from the the internal cos-function, which is no surprise, because yabasic is already statically linked to the standard C-library and uses its function to compute the cosine.

Searching a string within another string

A second example:

```
if peek$("os")="windows" then
  lib$ = "msvcrt.dll"
else
  lib$ = "libm.so.6"
endif
```

```
print foreign_function_call$(lib$,"string","strstr","string","foobar","string"
```

This example calls the `strstr`-function from the standard C-library; this function accepts two string arguments and returns a string, which is the first (if any) appearance of the second string within the first one (remark: this function makes more sense in C than in yabasic). Please note the option `"copy_string_result"`, which advices yabasic to return a copy of the result of `strstr`; otherwise your program might crash, because `strstr` simply returns a pointer to a part of its first argument, a string that will later be freed by yabasic.

Showing a message box under Windows

This example is windows only; it shows a standard Windows message box with the given title and message:

```
message_box("Hello World !","Message from yabasic")

sub message_box(message$, title$)
  msgptr$ = foreign_buffer_alloc$(len(message$)+1)
  foreign_buffer_set msgptr$, 0, message$
  titleptr$ = foreign_buffer_alloc$(len(title$)+1)
  foreign_buffer_set titleptr$, 0, title$
  hwnd$ = foreign_function_call$("user32.dll", "buffer", "GetActiveWindow")
  ret = foreign_function_call("user32.dll", "int32", "MessageBoxA", "buffer", h
  foreign_buffer_free msgptr$
  foreign_buffer_free titleptr$
  return ret
end sub
```

The relevant Windows-function `MessageBoxA` is found within the library `user32.dll`; most of the example deals with properly allocating, handling and freeing buffers to hold the supplied text-snippets. Thanx to *Jean-Marc Duro* for this example.

Internal steps during a call to a foreign function

A remark on libffi: this is the library which allows yabasic to call functions from other libraries libffi is used by many other programming-languages for the same purpose; in yabasic it is linked statically (rather than dynamically) so that its functionality is available right from the start. Summing up: libffi itself need not be loaded but helps to call functions from other loaded libraries.

Here is the sequence of events during a foreign function call (e.g. `foreign_function_call`):

- Yabasic parses the type specifications and argument values provided and collects the necessary information for libffi.
- The named library is loaded with the appropriate call (which is different under Windows and Unix). This step might easily fail, e.g. if you misspelled the name of the library or your system cannot find the library.
- With the help of libffi the named function is invoked.

- If you specified the option `unload_library`, the library that has been loaded is unloaded again.
- The return value of the function is converted to a form suitable for yabasic and your program continues.

Errors are reported during every step.

Abbreviations for long names

Yabasics functions for dealing with foreign libraries start with `foreign_function` or `foreign_buffer` (e.g. `foreign_buffer_alloc`). To help in typing, these names can all be abbreviated by contracting `foreign_function` into `frnfn` and `foreign_buffer` into `frnbf`. In the examples below, both forms appear.

Structurs and buffers

The C-language provides a wide variety of simple datatypes (like numbers an strings) and allows to aggregate simple datatypes to *structures* such a structure contains a set of simple types arranged without overlap (but sometimes with gaps). Yabasic on itself does not know the internals of a structure but rather treats it as a uniform *buffer*. Structure and buffer are just flipsides of the same memory area viewed either from C or yabasic. For your yabasic-program a buffer is represented by a *handle*, which is just a simple printable string (containing the size and the memory adress).

The detailed knowledge about the simple types within a structure must be coded into your program, which uses the command (or function) `foreign_buffer_set` and `foreign_buffer_get`. Both functions require type and offset (which needs to be looked up in documentation of the foreign library) of the simple type within the structure and a handle to the buffer, which contains the structure.

Beeing essentially a memory area, a buffer is created with `foreign_buffer_alloc` and destroyed with `foreign_buffer_free` if needed no more.

Besides representing a structure, a buffer can also provide room to store raw areas of memory for use by the foreign library; example might be image- or sound-content.

Two more complex examples

Dealing with time

The example below deals with the time functions from the standard C-library; some of them deal with the *tm* structure for keeping the segmented time; to understand the example it is good to have the tm-structure at hand; see below. In addition it might be helpful to consult the manpages of the various C-functions (e.g. `localtime`)involved.

```
struct tm {
    int tm_sec;    /* Seconds (0-60) */
```

```
    int tm_min;    /* Minutes (0-59) */
    int tm_hour;   /* Hours (0-23) */
    int tm_mday;   /* Day of the month (1-31) */
    int tm_mon;    /* Month (0-11) */
    int tm_year;   /* Year - 1900 */
    int tm_wday;   /* Day of the week (0-6, Sunday = 0) */
    int tm_yday;   /* Day in the year (0-365, 1 Jan = 0) */
    int tm_isdst;  /* Daylight saving time */
};
```

The example plays with the two forms of keeping the time, either as unix-time (number of seconds since epoch) or as a segmented time (sec, min, etc.). The six steps are each introduced by comments, please see below.

```
# First: Determine the correct library depending on OS
#
if peek$("os")="windows" then
  lib$ = "msvcrt.dll"
else
  lib$ = "libm.so.6"
endif

# Second: Get the unix-time
#
# time() has a pointer argument to store the result (in addition to returning i
# we pass NULL, so only the return value is relevant
#
null$ = foreign_buffer_alloc$(-1)
now = foreign_function_call(lib$,"int","time","buffer",null$)
print "Seconds since the epoch: ",now

# Third: Convert the unix-time to a segmented time
#
# localtime() does not accept the time-value as an argument, but rather require
# to the time-value, so we construct a buffer for one int and put in our value
now$ = foreign_buffer_alloc$(foreign_function_size("int"))
foreign_buffer_set now$,0,"int",now
# Dump the buffer for educational purpose
print "Dump of buffer:          ",foreign_buffer_dump$(now$)
# localtime() returns a structure with the componentes (year, day, sec, etc.) a
local$ = foreign_function_call$(lib$,"buffer","localtime","buffer",now$)

# Fourth: Get the current year from the resulting buffer
#
# assuming, that year is the sixth element of the structure
# so offset is 5
offset = 5 * foreign_function_size("int")
year = foreign_buffer_get(local$,offset,"int")
print "Current year:            ", year + 1900


# Fifth: manipulate the segmented time
#
# set year to something else
foreign_buffer_set local$,offset,"int",year-50


# Sixth: convert time-structure from localtime into ascii
#
print "50 years back:           ", foreign_function_call$(lib$,"string","asctir
```

On my computer this program produces the following output:

```
Seconds since the epoch: 1559014899
```

```
Dump of buffer:          F3ADEC5C
Current year:            2019
50 years back:           Tue May 28 05:41:39 1969
```

Getting the version of libcurl

This final example just invokes libcurl to report its version. This is somewhat involved, because the matching function `curl_version_info` (see its man-page) returns a structure, which contains a pointer to a string, as can be seen from the structures definition:

```
typedef struct {
  CURLversion age;          /* see description below */
  const char *version;      /* human readable string */
  unsigned int version_num; /* numeric representation */
  const char *host;         /* human readable string */
  int features;             /* bitmask, see below */
  char *ssl_version;        /* human readable string */
  long ssl_version_num;     /* not used, always zero */
  const char *libz_version; /* human readable string */
  const char * const *protocols; /* protocols */

  ...                       /* more lines omitted */

} curl_version_info_data;
```

Please note, that the yabasic-code below uses abbreviations (e.g. `frnfn_call` instead of `foreign_function_call`.

```
# Get structure with version info
info$ = frnfn_call$("libcurl.so.4","buffer","curl_version_info","int",1)
# dump it for reference
print frnbf_dump$(info$,32)
# assume, that the pointer to version string is at offset 8
sinfo$ = frnbf_get_buffer$(info$,8)
# print readable version
print frnbf_get$(sinfo$,0,10)
```

The printing of `frnbf_dump` gives a hint on the internal offsets within the structure and helps to determine that offset of 8 for the next call.

On my system this program produces `7.61.1` for the version of curl.

See also

foreign_function_call, foreign_function_call2, foreign_function_size, foreign_buffer_alloc, foreign_buffer_free, foreign_buffer_size, foreign_buffer_dump, foreign_buffer_set, foreign_buffer_set_buffer, foreign_buffer_get, foreign_buffer_get2, foreign_buffer_get_buffer, system

Chapter 7. All commands and functions of yabasic grouped alphabetically

A

H

hex$() — convert a number into hexadecimal

I

if — evaluate a condition and execute statements or not, depending on the result

import — import a library

inkey$ — wait, until a key is pressed

input — read input from the user (or from a file) and assign it to a variable

instr() — searches its second argument within the first; returns its position if found

int() — return the integer part of its single numeric argument

L

label — mark a specific location within your program for `goto`, `gosub` or `restore`

left$() — return (*or change*) left end of a string

len() — return the length of a string

line — draw a line

line input — read in a whole line of text and assign it to a variable

local — mark a variable as local to a subroutine

log() — compute the natural logarithm

loop — marks the end of an infinite loop

lower$() — convert a string to lower case

ltrim$() — trim spaces at the left end of a string

M

max() — return the larger of its two arguments

mid$() — return (*or change*) characters from within a string

min() — return the smaller of its two arguments

mod — compute the remainder of a division

mouseb — extract the state of the mousebuttons from a string returned by `inkey$`

mousemod — return the state of the modifier keys during a mouseclick

mousex — return the x-position of a mouseclick

mousey — return the y-position of a mouseclick

N

new curve — start a new curve, that will be drawn with the `line`-command

next — mark the end of a for loop

A

Name

abs() — returns the absolute value of its numeric argument

Synopsis

```
y=abs(x)
```

Description

If the argument of the `abs`-function is positive (e.g. 2) it is returned unchanged, if the argument is negative (e.g. -1) it is returned as a positive value (e.g. 1).

Example

```
print abs(-2),abs(2)
```

This example will print `2 2`

See also

sig

Name

acos() — returns the arcus cosine of its numeric argument

Synopsis

```
x=acos(angle)
```

Description

`acos` is the arcus cosine-function, i.e. the inverse of the `cos`-function. Or, more elaborate: It Returns the angle (in radians, not degrees !), which, fed to the cosine-function will produce the argument passed to the `acos`-function.

Example

```
print acos(0.5),acos(cos(pi))
```

This example will print `1.0472 3.14159` which are π/3 and π respectively.

See also

`cos`, `asin`

Name

and — logical and, used in conditions

Synopsis

```
if a and b …
while a and b …
```

Description

Used in conditions (e.g within `if`, `while` or `until`) to join two expressions. Returns `true`, if and only if its left and right argument are both `true` and `false` otherwise.

Note, that *logical shortcuts* may take place.

Example

```
input "Please enter a number" a
if (a>=1 and a<=9) print "your input is between 1 and 9"
```

See also

or,not

Name

and() — the bitwise arithmetic `and`

Synopsis

```
x=and(a,b)
```

Description

Used to compute the bitwise `and` of both its argument. Both arguments are treated as binary numbers (i.e. a sequence of digits 0 and 1); a bit of the resulting value will then be 1, if both arguments have a 1 at this position in their binary representation.

Note, that both arguments are silently converted to integer values and that negative numbers have their own binary representation and may lead to unexpected results when passed to `and`.

Example

```
print and(6,3)
```

This will print `2`. This result is clear, if you note, that the binary representation of 6 and 3 are 110 and 011 respectively; this will yield 010 in binary representation or 2 as decimal.

See also

or, eor and bitnot

Name

arraydim() — returns the dimension of the array, which is passed as an array reference

Synopsis

```
a=arraydim(b())
```

Description

If you apply the `arraydim()`-function on a one-dimensional array (i.e. a vector) it will return `1`, on a two-dimensional array (i.e. a matrix) it will return 2, and so on.

This is mostly used within subroutines, which expect an array among their parameters. Such subroutines tend to use the `arraydim`-function to check, if the array which has been passed, has the right dimension. E.g. a subroutine to multiply two matrices may want to check, if it really is invoked with two 2-dimensional arrays.

Example

```
dim a(10,10),b(10)
print arraydim(a()),arraydim(b())
```

This will print `2 1`, which are the dimension of the arrays `a()` and `b()`. You may check out the function `arraysize` for a full example.

See also

`arraysize` and `dim`.

Name

arraysize() — returns the size of a dimension of an array

Synopsis

```
x=arraysize(a(),b)
```

Description

The `arraysize`-function computes the size of the specified dimension of a given array. Here, *size* stands for the maximum number, that may be used as an index for this array. The first argument to this function must be an reference to an array, the second one specifies, which of the multiple dimensions of the array should be taken to calculate the size. Please note, that `arraysize` returns the value that has been used in the actual `dim`-statement, the real (internal) size of the array is allocated *one larger* in each dimension to have a first element at index 0; however this is *not* reflected by the output of `arraysize`.

An Example involving subroutines: Let's say, an array has been declared as `dim a(10,20)` (that is a two-dimensional array or a matrix). If this array is passed as an array reference to a subroutine, this sub will not know, what sort of array has been passed. With the `arraydim`-function the sub will be able to find the dimension of the array, with the `arraysize`-function it will be able to find out the size of this array in its two dimensions, which will be 10 and 20 respectively.

Our sample array is two dimensional; if you envision it as a matrix this matrix has 10 lines and 20 columns (see the `dim`-statement above. To state it more formally: The first dimension (lines) has a size of 10, the second dimension (columns) has a size of 20; these numbers are those returned by `arraysize(a(),1)` and `arraysize(a(),2)` respectively. Refer to the example below for a typical usage.

Example

```
rem
rem  This program adds two matrices elementwise.
rem

dim a(10,20),b(10,20),c(10,20)

rem  initialization of the arrays a() and b()
for y=1 to 10:for x=1 to 20
   a(y,x)=int(ran(4)):b(y,x)=int(ran(4))
next x:next y

matadd(a(),b(),c())

print "Result:"
for x=1 to 20
   for y=10 to 1 step -1
      print c(y,x)," ";
   next y
   print
next x

sub matadd(m1(),m2(),r())

   rem  This sub will add the matrices m1() and m2()
   rem  elementwise and store the result within r()
   rem  This is not very useful but easy to implement.
   rem  However, this sub excels in checking its arguments
   rem  with arraydim() and arraysize()

   local x:local y

   if (arraydim(m1())<>2 or arraydim(m2())<>2 or arraydim(r())<>2) then
      error "Need two dimensional arrays as input"
   endif

   y=arraysize(m1(),1):x=arraysize(m1(),2)
   if (arraysize(m2(),1)<>y or arraysize(m2(),2)<>x) then
      error "The two matrices cannot be added elementwise"
   endif

   if (arraysize(r(),1)<>y or arraysize(r(),2)<>x) then
      error "The result cannot be stored in the third argument"
   endif

   local xx:local yy
   for xx=1 to x
      for yy=1 to y
         r(yy,xx)=m1(yy,xx)+m2(yy,xx)
      next yy
   next xx

 end sub
```

See also

arraydim and dim.

Name

asc() — accepts a string and returns the position of its first character within the ascii charset

Synopsis

```
a=asc(char$)
```

Description

The asc-function accepts a string, takes its first character and looks it up within the ascii-charset; this position will be returned. The asc-function is the opposite of the chr$-function. There are valid uses for asc, however, comparing strings (i.e. to bring them into alphabetical sequence) is *not* among them; in such many cases you might consider to compare strings directly with <, = and > (rather than converting a string to a number and comparing this number).

Example

```
input "Please enter a letter between 'a' and 'y': " a$
if (a$<"a" or a$>"y") print a$," is not in the proper range":end
print "The letter after ",a$," is ",chr$(asc(a$)+1)
```

See also

chr$

Name

asin() — returns the arcus sine of its numeric argument

Synopsis

```
angle=asin(x)
```

Description

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

the asin-function.

Example

```
print asin(0.5),asin(sin(pi))
```

This will print 0.523599 -2.06823e-13 which is π/6 and almost 0 respectively.

See also

sin, acos

Name

at() — can be used in the print-command to place the output at a specified position

Synopsis

```
clear screen
…
print at(a,b)
print @(a,b)
```

Description

The at-clause takes two numeric arguments (e.g. at(2,3)) and can be inserted after the print-keyword. at() can be used only if clear screen has been executed at least once within the program (otherwise you will get an error).

The two numeric arguments of the at-function may range from 0 to the width of your terminal minus 1, and from 0 to the height of your terminal minus 1; if any argument exceeds these values, it will be truncated accordingly. However, yabasic has no influence on the size of your terminal (80x25 is a common, but not mandatory), the size of your terminal and the maximum values acceptable within the at-clause may vary. To get the size of your terminal you may use the peek-function: peek("screenwidth") returns the width of your terminal and peek("screenheight") its height.

Example

```
clear screen
maxx=peek("screenwidth")-1:maxy=peek("screenheight")-1
for x=0 to maxx
  print at(x,maxy*(0.5+sin(2*pi*x/maxx)/2)) "*"
next x
```

This example plots a full period of the sine-function across the screen.

See also

print, clear screen, color

Name

atan() — returns the arctangent of its numeric argument

Synopsis

```
angle=atan(a,b)
angle=atan(a)
```

Description

`atan` is the arctangent-function, i.e. the inverse of the tan-function. Or, more elaborate: It Returns the angle (in radians, not degrees !), which, fed to the `tan`-function will produce the argument passed to the `atan`-function.

The `atan`-function has a second form, which accepts two arguments: `atan(a,b)` which is (mostly) equivalent to `atan(a/b)` *except* for the fact, that the two-argument-form returns an angle in the range -π to π, whereas the one-argument-form returns an angle in the range -π/2 to π/2. To understand this you have to be good at math.

Example

```
print atan(1),atan(tan(pi)),atan(-0,-1),atan(-0,1)
```

This will print `0.463648 2.06823e-13 -3.14159 3.14159` which is π/4, almost 0, -π and π respectively.

See also

tan, sin

B

Name

color — change color for background of graphic window

Synopsis

```
backcolour red,green,blue
backcolour "red,green,blue"
```

Description

Change the color, that becomes visible, if any portion of the window is erased, e.g. after `clear window` or `clear line`. Note however, that parts of the window, that show the old background color will not change.

As with the `color`-command, the new background color can either be specified as a triple of three numbers or as a single string, that contains those three numbers separated by commas.

Note, that the command `backcolor` can be written as `backcolour` too and vice versa.

Example

```
open window 255,255
for x=10 to 235 step 10:for y=10 to 235 step 10
        backcolour x,y,0
        clear window
        sleep 1
next y:next x
```

This changes the background colour of the graphic window repeatedly and clears it every time, so that it is filled with the new background colour.

See also

`open window`, `color`, `line`, `rectangle`, `triangle`, `circle`

Name

beep — ring the bell within your computer; a synonym for `bell`

Synopsis

```
beep
```

Description

The `bell`-command rings the bell within your computer once. This command is *not* a sound-interface, so you can neither vary the length or the height of the sound (technically, it just prints `\a`). `bell` is exactly the same as `beep`.

Example

```
beep:print "This is a problem ..."
```

See also

beep

Name

bell — ring the bell within your computer (just as beep)

Synopsis

```
bell
```

Description

The beep-command rings the bell within your computer once. beep is a synonym for bell.

Example

```
print "This is a problem ...":beep
```

See also

bell

Name

bin$() — converts a number into a sequence of binary digits

Synopsis

```
hexadecimal$=bin$(decimal)
```

Description

The bin$-function takes a single numeric argument an converts it into a string of binary digits (i.e. zeroes and ones). If you pass a negative number to bin$, the resulting string will be preceded by a '-'.

If you want to convert the other way around (i.e. from binary to decimal) you may use the dec-function.

Example

```
for a=1 to 100
  print bin$(a)
next a
```

This example prints the binary representation of all digits between 1 and 100.

See also

hex$, dec, numbers with base 2 or 16.

Name

bind() — binds a yabasic-program and the yabasic-interpreter together into a *standalone* program

Synopsis

```
bind("foo.exe")
```

Description

The bind-command combines your own yabasic-program (plus all the libraries it does import) and the interpreter by copying them into a new file, whose name is passed as an argument. This new program may then be executed on any computer, even if it does not have yabasic installed.

Please see the section about creating a *standalone*-program for details.

Example

```
if (!peek("isbound")) then
  bind "foo"
  print "Successfully created the standalone executable 'foo' !"
  exit
endif

print "Hello World !"
```

This example creates a standalone program foo from itself.

See also

The section about creating a *standalone*-program, the peek-function

and the command line options.

Name

bitnot() — the bitwise arithmetic `not`

Synopsis

```
x=bitnot(a)
```

Description

This function is used to compute the *bitwise not* of its single argument. The argument is treated as binary number (i.e. a sequence of digits 0 and 1); a bit of the resulting value will be 1, if the argument has a 0 at this position in its binary representation; if the bit in the argument is 1, the bit in the result will be 0.

Note, that its argument is silently converted to a positive integer value and that negative numbers have their own binary representation and may lead to unexpected results when passed to `bitnot`.

A note on naming: This one-argument-function is named `bitnot` to distinguish it from the one-argument-function `not`, which operates on *logical* expressions. For the similar functions `and` and `or` this distinction between *logical* and *bitwise* function is done implicitly through the number of arguments (1 and 2, respectively).

Example

```
print bin$(not(17))
```

This will print `11111111111111111111111111101110`. This result is clear, if you note, that the binary representation of 17 is 10001, which inverted will give the long binary number given before.

See also

`or`, `eor` and `and`

Name

box — draw a rectangle. A synonym for `rectangle`

Synopsis

See the rectangle-command.

Description

The box-command does exactly the same as the rectangle-command; it is just a *synonym*. Therefore you should refer to the entry for the rectangle-command for further information.

Name

break — breaks out of one or more loops or switch statements

Synopsis

```
break
break 2
```

Description

break transfers control immediately outside the enclosing loop or switch statement. This is the preferred way of leaving a such a statement (rather than goto, which is still possible in most cases). An optional digit allows one to break out of multiple levels, e.g. to leave a loop from within a switch statement. Please note, that only a literal (e.g. 2) is allowed at this location.

Example

```
for a=1 to 10
  break
  print "Hi"
next a

while 1
  break
  print "Hi"
wend

repeat
  break
  print "Hi"
until 0

switch 1
case 1:break
case 2:case 3:print "Hi"
end switch
```

This example prints nothing at all, because each of the loops (and the switch-statement) does an immediate break (before it could print any "Hi").

See also

for, while, repeat and switch.

C

Name

case — mark the different cases within a switch-statement

Synopsis

```
switch a
  case 1
  case 2
  …
end switch

…

switch a$
  case "a"
  case "b"
  …
end switch
```

Description

Please see the switch-statement.

Example

```
input a
switch(a)
  case 1:print "one":break
  case 2:print "two":break
  default:print "more"
end switch
```

Depending on your input (a number is expected) this code will print `one` or `two` or otherwise `more`.

See also

switch

Name

ceil() — compute the ceiling for its (float) argument

Synopsis

```
print ceil(x)
```

Description

The `ceil`-function returns the smallest integer number, that is larger or equal than its argument.

Example

```
print ceil(1.5),floor(1.5)
print ceil(2),floor(2)
```

Comparing functions `ceil` and `floor`, gives a first line of output (`1 2`), showing that `ceil` is less or equal than `floor`; but as the second line of output (`2 2`) shows, the two functions give equal results for integer arguments.

See also

`floor`, `int`, `frac`, `round`

Name

chomp$() — remove a single trailing newline from its string-argument; if the string does not end in a newline, the string is returned unchanged

Synopsis

```
print chomp$("Hallo !\n")
```

Description

The `chomp$`-function checks, if its string-argument ends in a newline and removes it eventually; for this purpose `chomp$` can replace an `if`-statement. This can be especially useful, when you deal with input from external sources like `system$`.

You may apply `chomp$` freely, as it only acts, if there is a newline to remove; note however, that user-input, that comes from the normal `input`-statement, does not need such a treatment, because it already comes without a newline.

Example

The following yabasic-program uses the unix-command **`whoami`** to get the username of the current user in order to greet him personally. This is done twice: First with the `chomp$`-function and then again with

with an equivalent `if`-statement:

```
print "Hello " + chomp$(system$("whoami")) + " !"

user$ = system$("whoami")
if (right$(user$,1)="\n") user$=left$(user$,len(user$)-1)
print "Hello again " + user$ + " !"
```

See also

`system$`

Name

chr$() — accepts a number and returns the character at this position within the ascii charset

Synopsis

```
character$=chr$(ascii)
```

Description

The `chr$`-function is the opposite of the `asc`-function. It looks up and returns the character at the given position within the ascii-charset. It's typical use is to construct *nonprintable* characters which do not occur on your keyboard.

Nevertheless you won't use `chr$` as often as you might think, because the most important nonprintable characters can be constructed using *escape-sequences* using the \-character (e.g. you might use \n instead of `chr$(10)` wherever you want to use the newline-character).

Example

```
print "a",chr$(10),"b"
```

This will print the letters 'a' and 'b' in different lines because of the intervening newline-character, which is returned by `chr$(10)`.

See also

`asc`

Name

circle — draws a circle in the graphic-window

Synopsis

```
circle x,y,r
clear circle x,y,r
fill circle x,y,r
clear fill circle x,y,r
```

Description

The `circle`-command accepts three parameters: The x- and y-coordinates of the center and the radius of the circle.

Some more observations related with the `circle`-command:

- The graphic-window must have been opened already.
- The circle may well extend over the boundaries of the window.
- If you have issued `open printer` before, the circle will finally appear in the printed hard copy of the window.
- `fill circle` will draw a filled (with black ink) circle.
- `clear circle` will erase (or clear) the outline of the circle.
- `clear fill circle` or `fill clear circle` will erase the full area of the circle.

Example

```
open window 200,200

for n=1 to 2000
  x=ran(200)
  y=ran(200)
  fill circle x,y,10
  clear fill circle x,y,8
next n
```

This code will open a window and draw 2000 overlapping circles within. Each circle is drawn in two steps: First it is filled with black ink (`fill circle x,y,10`), then most of this circle is erased again (`clear fill circle x,y,8`). As a result each circle is drawn with an opaque white interior and a 2-pixel outline (2-pixel, because the radii differ by two).

See also

`open window`, `open printer`, `line`, `rectangle`, `triangle`

Name

clear — erase circleS, rectangleS or triangleS

Synopsis

```
clear rectangle 10,10,90,90
clear fill circle 50,50,20
clear triangle 10,10,20,20,50,30
```

Description

May be used within the circle, rectangle or triangle command and causes these shapes to be erased (i.e. be drawn in the colour of the background).

fill can be used in conjunction with and wherever the fill-clause may appear. Used alone, clear will erase the outline (not the interior) of the shape (circle, rectangle or triangle); together with fill the whole shape (including its interior) is erased.

Example

```
open window 200,200
fill circle 100,100,50
clear fill rectangle 10,10,90,90
```

This opens a window and draws a pacman-like figure.

See also

clear, circle, rectangle, triangle

Name

clear screen — erases the text window

Synopsis

```
clear screen
```

Description

clear screen erases the text window (the window where the output of print appears).

It must be issued at least once, before some advanced screen-commands (e.g. print at or inkey$) may be called; this requirement is due to some limitations of the curses-library, which is used by yabasic under Unix for some commands.

Example

```
clear screen
print "Please press a key : ";
a$=inkey$
print a$
```

The `clear screen` command is essential here; if it would be omitted, yabasic would issue an error ("`need to call 'clear screen' first`") while trying to execute the `inkey$`-function.

See also

inkey$

Name

clear window — clear the graphic window and begin a new page, if printing is under way

Synopsis

```
clear window
```

Description

`clear window` clears the graphic window. If you have started printing the graphic via `open printer`, the `clear window`-command starts a new page as well.

Example

```
open window 200,200
open printer "t.ps"

for a=1 to 10
if (a>1) clear window
text 100,100,"Hallo "+str$(a)
next a

close printer
close window
```

This example prints 10 pages, with the text "Hello 1", "Hello 2", ... and so on. The `clear screen`-command clears the graphics window and starts a new page.

See also

open window, open printer

Name

close — close a file, which has been opened before

Synopsis

```
close filenum
close # filenum
```

Description

The `close`-command closes an open file. You should issue this command as soon as you are done with reading from or writing to a file.

Example

```
open "my.data" for reading as 1
input #1 a
print a
close 1
```

This program opens the file `"my.data"`, reads a number from it, prints this number and closes the file again.

See also

open

Name

close curve — close a curve, that has been drawn by the `line`-command

Synopsis

```
new curve
line to x1,y1
…
close curve
```

Description

The `close curve`-command closes a sequence of lines, that has been drawn by repeated `line to`-commands.

Example

```
open window 200,200
new curve
line to 100,50
line to 150,150
line to 50,150
close curve
```

This example draws a triangle: The three `line to`-commands draw two lines; the final line is however not drawn explicitly, but drawn by the `close curve`-command.

See also

`line`, `new curve`

Name

close printer — stops printing of graphics

Synopsis

```
close printer
```

Description

The `close printer`-command ends the printing graphics. Between `open printer` and `close printer` everything you draw (e.g. circles, lines …) is sent to your printer. `close printer` puts an end to printing and will make your printer eject the page.

Example

```
open window 200,200
open printer
circle 100,100,50
close printer
close window
```

As soon as `close printer` is executed, your printer will eject a page with a circle on it.

See also

`open printer`

Name

close window — close the graphics-window

Synopsis

```
close window
```

Description

The `close window`-command closes the graphics-window, i.e. it makes it disappear from your screen. It includes an implicit `close printer`, if a printer has been opened previously.

Example

```
open window 200,200
circle 100,100,50
close window
```

This example will open a window, draw a circle and close the window again; all this without any pause or delay, so the window will be closed before you may regard the circle..

See also

`open window`

Name

color — change color for any subsequent drawing-command

Synopsis

```
colour red,green,blue
colour "red,green,blue"
```

Description

Change the color, in which lines, dots, circles, rectangles or triangles are drawn. The `color`-command accepts three numbers in the range 0 ... 255 (as in the first line of the synopsis above). Those numbers specify the intensity for the primary colors red, green and blue respectively. As an example `255,0,0` is red and `255,255,0` is yellow.

Alternatively you may specify the color with a single string (as in the second line of the synopsis above); this string should contain three numbers, separated by commas. As an example `"255,0,255"` would be violet. Using this variant of the `colour`-command, you may use symbolic names for colours:

```
open window 100,100
yellow$="255,255,0"
```

```
color yellow$
text 50,50,"Hallo"
```

, which reads much clearer.

Example

```
open window 255,255
for x=10 to 235 step 10:for y=10 to 235 step 10
        colour x,y,0
        fill rectangle x,y,x+10,y+10
next y:next x
```

This fills the window with colored rectangles. However, none of the used colours contains any shade of blue, because the `color`-command has always 0 as a third argument.

Note, that the command `color` can be written as `colour` too and vice versa.

See also

`open window`, `backcolor`, `line`, `rectangle`, `triangle`, `circle`

Name

compile — compile a string with yabasic-code *on the fly*

Synopsis

```
compile(code$)
```

Description

This is an advanced command (closely related with the `execute`-command). It allows you to compile a string of yabasic-code (which is the only argument). Afterwards the compiled code is a normal part of your program.

Note, that there is no way to *remove* the compiled code.

Examples

```
compile("sub mysub(a):print a:end sub")
mysub(2)
```

This example creates a function named `mysub`, which simply prints its single argument.

Another Example

This next example combines the functions `compile` and `execute`:

```
count = 1
subname$ = "foo" + str$(count)
compile("sub "+ subname$ + "(a):print a:end sub")
execute(subname$,2)
```

This example creates and executes a function, whose name (`foo1`) is stored within the variable `subname$`; the newly created function simply prints its single argument. This example could be executed multiple times within a single yabasic-program, simply by incrementing the variable `count`; by doing that, multiple subroutines (`foo1`, `foo2`, ...) could be created and executed in succession.

See also

adding code during execution, `execute`, `execute$`, `eval`, `eval$`

Name

continue — start the next iteration of a `for`-, `do`-, `repeat`- or `while`-loop

Synopsis

```
continue
```

Description

You may use `continue` within any loop to start the next iteration immediately. Depending on the type of the loop, the loop-condition will or will not be checked. Especially: `for`- and `while`-loops will evaluate their respective conditions, `do`- and `repeat`-loops will not.

Remark: Another way to change the flow of execution within a loop, is the break-command.

Example

```
for a=1 to 100
  if mod(a,2)=0 continue
  print a
next a
```

This example will print all odd numbers between 1 and 100.

See also

for, do, repeat, while, break

Name

cos() — return the cosine of its single argument

Synopsis

```
x=cos(angle)
```

Description

The cos-function expects an angle (in radians) and returns its cosine.

Example

```
print cos(pi)
```

This example will print -1.

See also

acos, sin

D

Name

data — introduces a list of data-items

Synopsis

```
data 9,"world"
…
read b,a$
```

Description

The data-keyword introduces a list of comma-separated list of strings or numbers, which may be retrieved with the read-command.

The data-command itself does nothing; it just stores data. A single data-command may precede an arbitrarily long list of values, in which strings or numbers may be mixed at will.

yabasic internally uses a data-pointer to keep track of the current location within the data-list; this pointer may be reset with the restore-

command.

Example

```
do
  restore
  for a=1 to 4
    read num$,num
    print num$,"=",num
  next a
loop
data "eleven",11,"twelve",12,"thirteen",13,"fourteen",14
```

This example just prints a series of lines `eleven=11` up to `fourteen=14` and so on without end.

The `restore`-command ensures that the list of `data`-items is read from the start with every iteration.

See also

`read`, `restore`

Name

date$ — returns a string with various components of the current date

Synopsis

```
a$=date$
```

Description

The `date$`-function (which *must* be called without parentheses; i.e. `date$()` would be an error) returns a string containing various components of a date; an example would be `4-05-27-2004-Thu-May`. This string consists of various fields separated by hyphens (`"-"`):

- The day within the week as a number in the range 0 (=Sunday) to 6 (=Saturday) (in the example above: `4`, i.e. Thursday).
- The month as a number in the range 1 (=January) to 12 (=December) (in the example: `5` which stands for May).
- The day within the month as a number in the range 1 to 31 (in the example: `27`).
- The full, 4-digit year (in the example: `2004`, which reminds me that I should adjust the clock within my computer ...).
- The abbreviated name of the day within the week (`Mon` to `Sun`).

- The abbreviated name of the month (`Jan` to `Dec`).

Therefore the whole example above (`4-05-27-2004-Thu-May`) would read: day 4 in the week (counting from 0), May 27 in the year 2004, which is a Thursday in May.

Note, that all fields within the string returned by `date$` have a fixed with (numbers are padded with zeroes); therefore it is easy to extract the various fields of a date format with `mid$`.

Example

```
rem   Two ways to print the same ...

print mid$(date$,3,10)

dim fields$(6)
a=split(date$,fields$(),"-")
print fields$(2),"-",fields$(3),"-",fields$(4)
```

This example shows two different techniques to extract components from the value returned by `date$`. The `mid$`-function is the preferred way, but you could just as well `split` the return-value of `date$` at every "-" and store the result within an array of strings.

See also

`time$`

Name

dec() — convert a base 2 or base 16 number into decimal form

Synopsis

```
a=dec(number$)
a=dec(number$,base)
```

Description

The `dec`-function takes the string-representation of a base-2 or base-16 (which is the default) number and converts it into a decimal number. The optional second argument (`base`) might be used to specify a base other than 16. However, currently only base 2 or base 16 are supported. Please note, that for base 16 and 2 you may write literals in the usual way, by preceding them with `0x` or `0b` respectively, e.g. like

```
print 0xff + 0b11
```

; this may save you from applying the `dec` altogether.

Example

```
input "Please enter a binary number: " a$
print a$," is ",dec(a$)
```

See also

bin$, hex$, numbers with base 2 or 16

Name

default — mark the *default*-branch within a switch-statement

Synopsis

```
switch a+3
case 1
  …
case 2
  …
default
  …
end switch
```

Description

The default-clause is an optional part of the switch-statement (see there for more information). It introduces a series of statements, that should be executed, if none of the cases matches, that have been specified before (each with its own case-clause).

So default specifies a default to be executed, if none of the explicitly named cases matches; hence its name.

Example

```
print "Please enter a number between 0 and 6,"
print "specifying a day in the week."
input d
switch d
case 0:print "Monday":break
case 1:print "Tuesday":break
case 2:print "Wednesday":break
case 3:print "Thursday":break
case 4:print "Friday":break
case 5:print "Saturday":break
case 6:print "Sunday":break
default:print "Hey you entered something invalid !"
end switch
```

This program translates a number between 0 and 6 into the name of a weekday; the default-case is used to detect (and complain about)

invalid input.

See also

sub, case

Name

dim — create an array prior to its first use

Synopsis

```
dim array(x,y)
dim array$(x,y)
```

Description

The `dim`-command prepares one or more arrays (of either strings or numbers) for later use. This command can also be used to enlarges an existing array.

When an array is created with the `dim`-statement, memory is allocated and all elements are initialized with either 0 (for numerical arrays) or "" (for string arrays). Please be aware, that the `dim` reserves room for one element *more* than actually specified, e.g. `dim(10)` reserves memory for *11* elements. This makes it possible to access element 0 as well as element 10, which serves the conventions of C as well as basic.

If the array already existed, and the `dim`-statement specifies a larger size than the current size, the array is enlarged and any old content is preserved. But note, that `dim` cannot be used to shrink an array: If you specify a size, that is smaller than the current size, the `dim`-command does nothing.

Finally: To create an array, that is only known within a single subroutine, you should use the command `local`, which creates local variables as well as local arrays.

Example

```
dim a(5,5)

for x=1 to 5:for y=1 to 5
  a(x,y)=int(ran(100))
next y:next x

printmatrix(a())

dim a(7,7)

printmatrix(a())
```

```
sub printmatrix(ar())
  local x,y,p,q
  x=arraysize(ar(),1)
  y=arraysize(ar(),2)
  for q=1 to y
    for p=1 to y
      print ar(p,q),"\t";
    next p
    print
  next q
end sub
```

This example creates a 2-dimensional array (i.e. a *matrix*) with the `dim`-statement and fills it with random numbers. The second `dim`-statement enlarges the array, all new elements are filled with 0.

The subroutine `printmatrix` just does, what its name says.

See also

`arraysize`, `arraydim`, `local`

Name

do — start a (conditionless) `do-loop`

Synopsis

```
do
…
loop
```

Description

Starts a loop, which is terminated by `loop`; everything between `do` and `loop` will be repeated forever. This loop has no condition, so it is an infinite loop; note however, that a `break`- or `goto`-statement might be used to leave this loop anytime.

Example

```
do
  a=a+1
  print a
  if (a>100) break
loop
```

This example prints the numbers between 1 and 101. The `break`-statement is used to leave the loop.

See also

loop, repeat, while, break

Name

doc — special comment, which might be retrieved by the program itself

Synopsis

```
doc   This is a comment
docu  This is another comment
```

Description

Introduces a comment, which spans up to the end of the line. But other than the rem-comment, any docu-comment is collected within the special docu$-array and might be retrieved later on. Moreover you might invoke yabasic -docu foo.yab on the *command line* to retrieve the embedded documentation within the program foo.yab.

Instead of doc you may just as well write docu or even documentation.

Example

```
rem   Hi, this has been written by me
rem
doc   This program asks for a number and
doc   prints this number multiplied with 2
rem
rem   Print out the above message
rem
for a=1 to arraysize(docu$(),1):print docu$(a):next a

rem   Read and print the number
input "Please input a number: " x
print x*2
```

This program uses the comments within its code to print out a help message for the user; if you run this program, you get this output:

```
   This program asks for a number and
   prints this number multiplied with 2
 Please input a number: 2
 4
```

The contents of the doc-lines are retrieved from the docu$-array; if you do not want a comment to be collected within this array, use the rem-statement instead.

See also

docu$, rem

Name

docu$ — special array, containing the contents of all docu-statement within the program

Synopsis

```
a$=docu$(1)
```

Description

Before your program is executed, yabasic collects the content of all the `doc`-statements within your program within this 1-dimensional array (well only those within the main-program, libraries are skipped).

You may use the `arraysize` function to find out, how many lines it contains.

Example

```
docu
docu  This program reads two numbers
docu  and adds them.
docu

rem retrieve and print the embedded documentation
for a=1 to arraysize(docu$(),1)
  print docu$(a)
next a

input "First number: " b
input "Second number: " c

print "The sum of ",b," and ",c," is ",b+c
```

This program uses the embedded documentation to issue a usage-message.

See also

arraydim, rem

Name

dot — draw a dot in the graphic-window

Synopsis

```
dot x,y
```

```
clear dot x,y
```

Description

Draws a dot at the specified coordinates within your graphic-window. If printing is in effect, the dot appears on your printout too.

Use the functions `peek("winheight")` or `peek("winwidth")` to get the size of your window and hence the boundaries of the coordinates specified for the `dot`-command.

Example

```
open window 200,200
circle 100,100,100
do
  x=ran(200):y=ran(200)
  dot x,y
  total=total+1
  if (sqrt((x-100)^2+(y-100)^2)<100) in=in+1
  print 4*in/total
loop
```

This program uses a well known algorithm to compute π.

See also

`line`, `open window`

E

Name

else — mark an alternative within an `if`-statement

Synopsis

```
if (…) then
  …
else
  …
endif
```

Description

The `else`-statement introduces the alternate branch of an `if`-statement. I.e. it starts the sequence of statements, which is executed, if the condition of the `if`-statement is *not* true.

Example

```
input "Please enter a number: " a
if (mod(a,2)=1) then
  print a," is odd."
else
  print a," is even."
endif
```

This program detects, if the number you have entered is even or odd.

See also

if

Name

elsif — starts an alternate condition within an `if`-statement

Synopsis

```
if (…) then
  …
elseif (…)
  …
elsif (…) then
  …
else
  …
endif
```

Description

The `elsif`-statement is used to select a single alternative among a series of choices.

With each `elsif`-statement you may specify a condition, which is tested, if the main condition (specified with the `if`-statement) has failed. Note that `elsif` might be just as well written as `elseif`.

Within the example below, two variables `a` and `b` are tested against a range of values. The variable `a` is tested with the `elsif`-statement. The very same tests are performed for the variable `b` too; but here an involved series of `if`-`else`-statements is employed, making the tests much more obscure.

Example

```
input "Please enter a number: " a
if (a<0) then
  print "less than 0"
elseif (a<=10) then
  print "between 0 and 10"
elsif (a<=20)
  print "between 11 and 20"
else
```

```
  print "over 20"
endif

input "Please enter another number: " b
if (b<0) then
  print "less than 0"
else
  if (b<=10) then
    print "between 0 and 10"
  else
    if (b<=20) then
      print "between 11 and 20"
    else
      print "over 20"
    endif
  endif
endif
```

Note, that the very same tests are performed for the variables `a` and `b`, but can be stated much more clearly with the `elsif`-statement.

Note, that `elsif` might be written as `elseif` too, and that the keyword `then` is optional.

See also

`if`, `else`

Name

end — terminate your program

Synopsis

```
end
```

Description

Terminate your program. Much (but not exactly) like the `exit` command.

Note, that `end` may not end your program immediately; if you have opened a window or called `clear screen`, yabasic assumes, that your user wants to study the output of your program after it has ended; therefore it issues the line `---Program done, press RETURN---` and waits for a key to be pressed. If you do not like this behaviour, consider using `exit`.

Example

```
print "Do you want to continue ?"
input "Please answer y(es) or n(o): " a$
if (lower$(left$(a$,1))="n") then
  print "bye"
```

```
  end
fi
```

See also

exit

Name

endif — ends an `if`-statement

Synopsis

```
if (…) then
  …
endif
```

Description

The `endif`-statement closes (or ends) an `if`-statement.

Note, that `endif` may be written in a variety of other ways: `end if`, `end-if` or even `fi`.

The `endif`-statement must be omitted, if the `if`-statement does not contain the keyword `then` (see the example below). Such an `if`-statement without `endif` extends only over a single line.

Example

```
input "A number please: " a
if (a<10) then
  print "Your number is less than 10."
endif

REM  and now without endif

input "A number please: " a
if (a<10) print "Your number is less than 10."
```

See also

if

Name

end sub — ends a subroutine definition

Synopsis

```
sub foo(…)
  …
end sub
```

Description

Marks the end of a subroutine-definition (which starts with the `sub`-keyword). The whole concept of subroutines is explained within the entry for `sub`.

Example

```
print foo(3)

sub foo(a)
  return a*2
end sub
```

This program prints out `6`. The subroutine `foo` simply returns twice its argument.

See also

`sub`

Name

eof — check, if an open file contains data

Synopsis

```
open 1,"foo.bar"
if (eof(1)) then
  …
end if
```

Description

The `eof`-function checks, if there is still data left within an open file. As an argument it expects the file-number as returned by (or used within) the `open`-function (or statement).

As a special case, if the argument is zero: test if input from stdin is available.

Example

```
a=open("foo.bar")
```

```
while not eof(a)
  input #a,a$
  print a$
end while
```

This example will print the contents of the file "foo.bar". The `eof`-function will terminate the loop, if there is no more data left within the file.

See also

open

Name

eor() — compute the bitwise *exclusive or* of its two arguments

Synopsis

```
print eor(a,b)
```

Description

The `eor`-function takes two arguments and computes their bitwise *exclusive or*. I.e. treat each arguments as a sequence of bits and compare theses two sequences bit by bit to produce the result. If the bits from the arguments are equal, the resulting bit will be `0`, otherwise `1`.

The `xor`-function is the same as the `eor` function; both are synonymous; however they have each their own description, so you may check out the entry of `xor` for a slightly different view.

Example

```
for a=0 to 3
  for b=0 to 3
    print fill$(bin$(a))," eor ",fill$(bin$(b))," = ",fill$(bin$(eor(a,b)))
  next b
next a

sub fill$(a$)
  return right$("0"+a$,2)
end sub
```

This example prints a table, from which you may figure, how the `eor`-function is computed.

See also

and, or

Name

error — raise an error and terminate your program

Synopsis

```
error "Wrong, wrong, wrong !!"
```

Description

Produces the same kind or error messages, that yabasic itself produces (e.g. in case of a syntax-error). The single argument is issued along with the current line-number.

Example

```
input "Please enter a number between 1 and 10: " a
if (a<1 or a>10) error "Oh no ..."
```

This program is very harsh in checking the users input; instead of just asking again, the program terminates with an error, if the user enters something wrong.

The error message would look like this:

```
---Error in t.yab, line 2: Oh no ...
---Error: Program stopped due to an error
```

See also

Well, there *should* be a corresponding called `warning`; unfortunately ther is none yet.

Name

euler — another name for the constant `2.71828182864`

Synopsis

```
foo=euler
```

Description

`euler` is the well known constant named after *Leonard Euler*; its value is `2.71828182864`. `euler` is *not* a function, so parens are not allowed (i.e.

`euler()` will produce an error). Finally, you may not assign to `euler`; it wouldn't sense anyway, because it is a constant.

Example

```
print euler
```

See also

pi

Name

eval() — compile and execute a single numeric expression

Synopsis

```
print eval("1+2")
```

Description

`eval` accepts a string, which should be the text of a single numeric expression; it processes the expression and returns the result. All numeric functions and arithmetic operators of yabasic can be used as well as any variables known.

The string passed to `eval` is first compiled and then executed *right away*. The compilation happens just before the execution and may cause compilation errors, if you pass an invalid expression. `eval` might come handy, if you want to calculate an expression, that is not known at the start of your program, e.g. because it is read from the user; see the example below.

Example

```
input "Please enter an aritmetic expression involving the variable x: " expr$
first = true
for x=0 to 100 step 0.01
  result = eval(expr$)
  if (first or result > maximum) maximum = result: xmaximum = x
  first = false
next x
print "In the range 0 to 100, expression " + expr$ + " has its maximum of " + s
```

The example above reads an arithmetic expression from the user and steps through the range 0 … 100 to find its maximum. If the user types e.g. **`-(x-50)**2`**, the program would find a maximum of around zero (e.g. `-1.90013e-24`) at 50.

See also

adding code during execution, eval$, compile, execute, execute$

Name

eval$() — compile and execute a single string-expression

Synopsis

```
print eval$("a$ + b$")
```

Description

eval$ accepts a string, which should be the text of a single string-expression; it processes the expression and returns the result. All string-functions and string-operators of yabasic can be used as well as any variables known.

The string passed to eval$ is first compiled and then executed *right away*. The compilation happens right before the execution and may cause compilation errors, if you pass an invalid expression. See the example below for two interesting use-cases.

A short but useful Example

The example below allows to apply the quoting rules of yabasic to user-input:

```
input "Please enter a string with some excape-sequences (e.g. \\r,\\n,\\t): " a
print eval$("\"" + a$ + "\"")
```

If the user types **abc\ndef** at the prompt, the text is echoed like this:

```
abc
def
```

A longer Example

The next example shows the subroutine evemex$ (for *eval embedded expression*) that allows to embed expressions into a string, simply by enclosing them with {{ and }}:

```
input "Please enter your name: " name$
print evemex$("Hello {{name$}}, your name has {{len(name$)}} characters.")
```

```
sub evemex$(evemex_str$)

  local evemex_pos1, evemex_pos2, evemex_res$

  evemex_pos1 = 1
  evemex_pos2 = 1
  evemex_res$ = ""

  while (evemex_pos1 < len(evemex_str$))
    if (mid$(evemex_str$, evemex_pos1, 2) = "{{") then
      evemex_res$ = evemex_res$ + mid$(evemex_str$, evemex_pos2, evemex_pos1 -
      evemex_pos1 = evemex_pos1 + 2
      evemex_pos2 = evemex_pos1
      while (evemex_pos2 < len(evemex_str$))
        if (mid$(evemex_str$, evemex_pos2, 2) = "}}") then
           rem
           rem  See the use of eval in the next line
           rem
           evemex_res$ = evemex_res$ + eval$("str$(" + mid$(evemex_str$, evemex
           evemex_pos2 = evemex_pos2 + 2
           evemex_pos1 = evemex_pos2
           break
        else
          evemex_pos2 = evemex_pos2 + 1
        endif
      wend
    else
      evemex_pos1 = evemex_pos1 + 1
    endif
  wend

  evemex_res$ = evemex_res$ + mid$(evemex_str$, evemex_pos2, evemex_pos1 - ever
  return evemex_res$

end sub
```

If the user when prompted types **Marc**, he is greeted with `Hello Marc, your name has 4 characters`. The program uses `eval$` only once, and it adds `str$` around the embedded expression to ensure, that the result is always a string and can be concatenated with the other strings.

Please note, that the subroutine prefixes its local variables with `evemex` (for *eval embedded expression*) to avoid name clashes with any variable that might be used in expressions within the string passed.

See also

adding code during execution, `eval`, `compile`, `execute`, `execute$`

Name

execute() — execute a user defined subroutine, which must return a number

Synopsis

```
print execute("bar","arg1","arg2")
```

Description

The execute-function is the counterpart of the execute$-function (please see there for some caveats). execute may be used to execute subroutines, which return a number.

Example

```
print execute("bar",2,3)
sub bar(a,b)
  return a+b
end sub
```

This example would print out 5.

See also

adding code during execution, compile, execute$, eval, eval$

Name

execute$() — execute a user defined subroutine, which must return a string

Synopsis

```
print execute$("foo$","arg1","arg2")
```

Description

execute$ can be used to execute a user defined subroutine, whose name is specified as a string expression.

This function allows to execute a subroutine, whose name is not known by the time you write your program. This might happen, if you want to execute a subroutine, which is compiled (using the compile command) as late as of execution of your program.

Note however, that the execute$-function is *not* the preferred method to execute a user defined subroutine; in almost all cases you should just execute a subroutine by writing down its name within your yabasic program (see the example below).

Example

```
print execute$("foo$","Hello","world !")
sub foo$(a$,b$)
  return a$+" "+b$
end sub
```

```
```

The example simply prints `Hello world !`, which is the return value of the user defined subroutine `foo$`. The same could be achieved by executing:

```
print foo$(a$,b$)
```

So this example does not really *need* the `execute$`-function; see `compile` for examples, that *do*.

See also

adding code during execution, `compile`, `execute$`, `eval`, `eval$`

Name

exit — terminate your program

Synopsis

```
exit
exit 1
```

Description

Terminate your program and return any given value to the operating system. `exit` is similar to `end`, but it will terminate your program immediately, no matter what.

Example

```
print "Do you want to continue ?"
input "Please answer y(es) or n(o): " a$
if (lower$(left$(a$,1))="n") exit 1
```

See also

`end`

Name

exp() — compute the exponential function of its single argument

Synopsis

```
foo=exp(bar)
```

Description

This function computes *e* to the power of its argument, where *e* is the well known euler constant `2.71828182864`.

The `exp`-function is the inverse of the `log`-function.

Example

```
open window 100,100
for x=0 to 100
   dot x,100-100*exp(x/100)/euler
next x
```

This program plots part of the `exp`-function, however the range is rather small, so that you may not recognize the function from this plot.

See also

`log`

Name

export — mark a function as globally visible

Synopsis

```
export sub foo(bar)
…
end sub
```

Description

The `export`-statement is used within libraries to mark a user defined subroutine as visible outside the library wherein it is defined. Subroutines, which are not exported, must be qualified with the name of the library, e.g. `foo.baz` (where `foo` is the name of the library and `baz` the name of the subroutine); *exported* subroutines may be used without specifying the name of the library, e.g. `bar`.

Therefore `export` may only be useful within libraries.

Example

The library `foo.bar` (which is listed below) defines two functions `bar` and `baz`, however only the function `bar` is *exported* and therefore

visible even outside the library; `baz` is *not* exported and may only be used within the library `foo.yab`:

```
export sub bar()
  print "Hello"
end sub

sub baz()
  print "World"
end sub
```

Now within your main program `cux.yab` (which `import`s the library foo.yab); note that this program produces an error:

```
import foo

print "Calling subroutine foo.bar (okay) ..."
foo.bar()
print "done."

print "Calling subroutine bar (okay) ..."
bar()
print "done."

print "Calling subroutine foo.baz (okay) ..."
foo.baz()
print "done."

print "Calling subroutine baz (NOT okay) ..."
baz()
print "done."
```

The output when executing **`yabasic foo.yab`** is this:

```
Calling subroutine foo.bar (okay) ...
Hello
done.
Calling subroutine bar (okay) ...
Hello
done.
Calling subroutine foo.baz (okay) ...
World
done.
Calling subroutine baz (NOT okay) ...
---Error in main.yab, line 16: can't find subroutine 'baz'
---Dump: sub baz() called in main.yab,16
---Error: Program stopped due to an error
```

As the error message above shows, the subroutine `baz` must be qualified with the name of the library, if used outside the library, wherein it is defined (e.g. `foo.baz`). I.e. outside the library `foo.yab` you need to write `foo.baz`. `baz` alone would be an error.

The subroutine `bar` (without adding the name of the library) however may (and probably should) be used in any program, which imports the library `foo.yab`.

Note

In some sense the set of exported subroutines constitutes the *interface* of a library.

See also

Libraries, `sub`, `import`

F

Name

false — a constant with the value of 0

Synopsis

```
okay=false
```

Description

The constant `false` can be assigned to variables which later appear in conditions (e.g. within an `if`-statement.

`false` may also be written as `FALSE` or even `FaLsE`.

Example

```
input "Please enter a number between 1 and 10: " a
if (check_input(a)) print "Okay"

sub check_input(x)
  if (x>10 or x<1) return false
  return true
end sub
```

The subroutine `check_input` checks its argument and returns `true` or `false` according to the outcome of the check..

See also

`true`

Name

fi — another name for `endif`

Synopsis

```
if (…)
…
fi
```

Description

`fi` marks the end of an `if`-statement and is exactly equivalent to `endif`, please see there for further information.

Example

```
input "A number please: " a
if (a<10) then
  print "Your number is less than 10."
fi
```

See also

`endif`

Name

fill — draw a filled `circle`s, `rectangle`s or `triangle`s

Synopsis

```
fill rectangle 10,10,90,90
fill circle 50,50,20
fill triangle 10,20,20,10,20,20
```

Description

The keyword `fill` may be used within the `circle`, `rectangle` or `triangle` command and causes these shapes to be filled.

`fill` can be used in conjunction with and wherever the `clear`-clause may appear. Used alone, `fill` will fill the interior of the shape (circle, rectangle or triangle); together with `clear` the whole shape (including its interior) is erased.

Example

```
open window 200,200
fill circle 100,100,50
clear fill rectangle 10,10,90,90
```

This opens a window and draws a pacman-like figure.

See also

clear, circle, rectangle, triangle

Name

floor() — compute the floor for its (float) argument

Synopsis

```
print floor(x)
```

Description

The floor-function returns the largest integer number, that is smaller or equal than its argument. For positive numbers x, floor(x) is the same as int(x); for negaive numbers it can be different (see the example below).

Example

```
print int(-1.5),floor(-1.5)
print int(-1),floor(-1)
print int(1.5),floor(1.5)
```

This example compares the functions int and floor, starting with -1 -2, then -1 -1 and ending with 1 1, which shows the different behaviour of both functions.

See also

ceil, int, frac, round

Name

for — starts a for-loop

Synopsis

```
for a=1 to 100 step 2
  …
next a
```

Description

The for-loop lets its numerical variable (a in the synopsis) assume all values within the given range. The optional step-clause may specify a

value (default: 1) by which the variable will be incremented (or decremented, if `step` is negative).

Any `for`-statement can be replaced by a set of `if`s and `goto`s; as you may infer from the example below this is normally not feasible. However if you want to know in detail how the `for`-statement works, you should study this example, which presents a `for`-statement and an *exactly equivalent* series of `if`s and `goto`s.

Example

```
for a=1 to 10 step 2:print a:next

a=1
label check
if (a>10) goto done
  print a
  a=a+2
goto check
label done
```

This example simply prints the numbers 1, 3, 5, 7 and 9. It does this twice: First with a simple `for`-statement and then with `if`s and `goto`s.

See also

`step`, `next`

Name

foreign_buffer_alloc$() — Create a new buffer for use in a foreign function call

Synopsis

```
handle$=foreign_buffer_alloc$(10)
```

Description

`foreign_buffer_alloc$()` creates a new buffer of specified size (using the C-function `malloc`). This buffer can then be populated by `foreign_buffer_set` or `foreign_buffer_set_buffer`, passed to `foreign_function_call` and finally freed with `foreign_buffer_free`.

The special value of `-1` can be passed to create the equivalent of a *null-pointer* in C: when your yabasic-program passes such a buffer to an external function, it is replaced by a null-pointer.

See also

The overview-section on foreign functions, list of related functions and commands

---

Name

foreign_buffer_dump$() — return the content of a buffer as a hex-encoded string

Synopsis

```
print foreign_buffer_dump(handle$)
```

Description

`foreign_buffer_dump$()` is mostly used during development of your yabasic program and helps to investigate the content of a buffer; this can be helpful to find out, how a structure for a foreign function call is aggregated. To actually retrieve elements from the structure rather use `foreign_buffer_get`.

See also

The overview-section on foreign functions, list of related functions and commands

---

Name

foreign_buffer_free — free a foreign buffer

Synopsis

```
foreign_buffer_free handle$
```

Description

`foreign_buffer_free()` expects a handle for a buffer and frees this buffer (using the C-function `free`), i.e. gives this memory area back to the operating system. Any subsequent attempt to access part of this buffer (e.g. via `foreign_buffer_get`) will probably lead to an error (as will a second call to `foreign_buffer_free()`). The handle-argument must have been returned previously by `foreign_buffer_alloc` or `foreign_function_call`.

See also

The overview-section on foreign functions, list of related functions and commands

---

Name

foreign_buffer_get() — extract a number from a foreign buffer

Synopsis

```
print foreign_buffer_get(handle$,10,"int")
```

Description

`foreign_buffer_get()` retrieves a simple type from a buffer, that is assumed to contain a structure (see the overview-section for the necessary background). For this it needs three arguments:

- A handle to a buffer, that has been previously filled, e.g. by a foreign function call.
- An offset, counted in bytes from the start of the buffer, which specifies where the value can be found within the structure.
- A string, specifying the type of the value that should be retrieved, which for this function is always an integer type like `"int"`, `"short"` or `"long"`.

Correct usage of this function requires an good understanding of the respective structure contained within the buffer.

See also

The overview-section on foreign functions, list of related functions and commands, the neighbouring functions for retrieving other values from the buffer.

Name

foreign_buffer_get$() — extract a string from a foreign buffer

Synopsis

```
print foreign_buffer_get$(handle$,0,12)
```

Description

`foreign_buffer_get$()` retrieves a string from a foreign buffer; its areguments are

- A handle to a buffer, that has been previously filled, e.g. by a foreign function call.
- An offset within this buffer, where the string starts. In the

common case, where the buffer can be assumed to contain a string *only*, this offset should be 0.

- The maximum length of the expected string. This is a value not necessarily known but if the string is null-terminated (as usual) you may just specify a much larger number here.

Correct usage of this function requires an good understanding of the respective structure contained within the buffer.

See also

The overview-section on foreign functions, list of related functions and commands

---

Name

foreign_buffer_get_buffer() — take a buffer and construct a handle to a second buffer from its content

Synopsis

```
handle2$ = foreign_buffer_get_buffer(handle$,8)
```

Description

`foreign_buffer_get_buffer()` should be used, if a buffer (i.e. the contained structure) is known to contain a *pointer* to a string or another structure. `foreign_buffer_get_buffer()` then reads this pointer and transforms it into a handle, that can then be used by other functions from the `foreign_buffer_get`-family. The two arguments are:

- A handle to a buffer, that has been previously filled, e.g. by a foreign function call.
- An offset within the buffer to the start of the buffer. A (third) length-argument is not required, because typically all pointers on a platform have the same length.

See the libcurl-example in the overview-section for an example.

See also

The overview-section on foreign functions, list of related functions and commands

---

Name

foreign_buffer_set — store a given value within a buffer

Synopsis

```
foreign_buffer_set handle$,4,"Hello World"
foreign_buffer_set handle$,6,"long",42
```

Description

`foreign_buffer_set` can be used to populate a structure within a foreign buffer. It accepts strings (first line in synopsis) and numbers (second line in synopsis) and stores them at the given offset. The arguments are:

- A handle to a buffer, that might have been just allocated or been returned from a foreign function call.
- An offset, specifying the first byte where the given data will be stored.
- The third argument: If you want to store a string, specify it just here; if you want to store a number, specify its type (e.g. `int`).
- A fourth argument is only needed, when you want to store a number; this number needs then to be given here.

See also

The overview-section on foreign functions, list of related functions and commands

---

Name

foreign_buffer_set_buffer — store a pointer to one buffer within another buffer

Synopsis

```
foreign_buffer_set handle1$,16,handle2$
```

Description

`foreign_buffer_set_buffer` stores a pointer (as in C) to a buffer within another buffer. It accepts a handle to a buffer (`handle2$` in the synopsis) and stores it as a pointer at the given offset within the buffer given first (`handle1$`). The arguments are:

- A handle to a buffer, that might have been just allocated or been returned from a foreign function call.
- An offset, specifying the first byte where the pointer to the given buffer will be stored.

- A handle to a buffer whose address (pointer) will be stored within the first buffer.

See also

The overview-section on foreign functions, list of related functions and commands

---

Name

foreign_buffer_size() — return the size of the foreign buffer

Synopsis

```
size = foreign_buffer_size(handle$)
```

Description

Return the size of the given buffer; if your handle just encapsulates a null-pointer, this will return 0; if the size is not known (standard for buffers returned from a foreign function call), the size will be -1.

See also

The overview-section on foreign functions, list of related functions and commands

---

Name

foreign_function_call() — call a function (returning a number) from a non-yabasic library or dll

Synopsis

```
print foreign_function_call("libtimestwo","int","timestwo","int",3)
```

Description

`foreign_function_call` calls a function from an external library.

In general, this feature is mostly useful, if you have such a library written or aquired. If you have an external command, that can be called interactively (i.e. from the commandline), you might try the `system`-function.

Please see the overview-section on foreign functions for overview, background and more examples. For details on functionality and

arguments see below.

The example uses the `foreign_function_call` to invoke the `cos`-function from the standard C-library (`libm.so.6` under Unix or `msvcrt.dll` under Windows).

Example

```
if peek$("os") = "unix" then
  lib$ = "libm.so.6"
else
  lib$ = "msvcrt.dll"
fi
print foreign_function_call(lib$,"double","cos","double",2)
```

The `foreign_function_call`-function accepts a variable number of arguments; 3 at minimum, 5 in the example above:

- A string, containing the name of the library, e.g. `"libm.so.6"`.
- A string describing the type, that the function (specified with the next argument) returns (in the example: `"double"`). See below for a list of all such types.
- The name of the function, that should be called (in the example above: `"cos"`)

If the external function does not require arguments itself, the three arguments above are everything needed for `foreign_function_call`. However (as for `cos`), if the external function itself requires arguments, than for each of its arguments, two more arguments are needed `foreign_function_call`:

- A string describing the *type* of the parameter to the external function (in the example: `"double"`). See the overview-section on foreign functions for a list of all such types.
- A value, that can be converted to the specified type (in the example: `2`).

With all these arguments specified, yabasic will call the foreign function (in the example: `cos`) and return its result; this process can be influenced by specifying *options* (see below).

Options

Options can be appended in any number after the string `"options"`, e.g. like:

```
              foreign_function_call(lib$,"double","cos","double",2,"options","
```

Each option can be preceded with the string `no_` to invert its meaning. As given below (i.e. with or without `no_`) the values represent the

respective default.

error

In case of errors (e.g. if a library cannot be found), should the function report them actively (which terminates your yabasic-program) ? Or should the error silently stored away for later retrieval by `peek("last_foreign_function_call_okay")` and `peek$("last_foreign_function_call_error_text")` ?

no_copy_string_result

If the foreign function returns a string-value (like `strstr`), it should be invoked using `foreign_function_call$`, which returns a string. Now, depending on the foreign function invoked, it might be necessary to make a copy of its result before returning it to yabasic. In the case of `strstr` this is needed, because it returns just a pointer to part of its input string which will yabasic happily free later on, probably leading to a segfault after.

no_unload_library

Normally, after making a call to a foreign function, the named library is kept in memory for further use (see background). However, sometimes you might want to dismiss the library right after the call; then you may specify this option.

See also

The overview-section on foreign functions, list of related functions and commands, `system`

Name

foreign_function_call$() — call a function (returning a string or a buffer) from a non-yabasic library or dll

Synopsis

```
a$=foreign_function_call$("libupper","string","toupper","string","hello world")
```

Description

This function calls a function from an external library, just like `foreign_function_call` (which see for most of the description). The only difference is that `foreign_function_call$` should be used when the foreign function returns a string or a structure (which itself is contained in a buffer, which is represented by a handle, which is a string).

The arguments are just the same as in `foreign_function_call`, only the second argument (`"string"` in the example above) can only be `"string"`

or "buffer" in accord to the nature of foreign_function_call$, which itself must return a string.

Among the options described at foreign_function_call, the option "copy_string_result" can really only be applied here.

See also

The overview-section on foreign functions, list of related functions and commands

---

Name

foreign_function_size() — return the size of one of the types available for foreign function calls

Synopsis

```
offset=foreign_function_size("short")
```

Description

The function returns the size of any of the types available for calls to foreign functions (see the overview section for a complete list). This is useful when calculating offsets needed for foreign_buffer_set or foreign_buffer_get.

See also

The overview section on foreign functions, list of related functions and commands

---

Name

frnbf_ and frnfn_ — Abbreviations for foreign_buffer_ and foreign_function_

Synopsis

```
print frnfn_size("short")
```

Description

The abbreviations frnbf_ and frnfn_ are just short for foreign_buffer_ and foreign_function_; you might prefer one over the other; in any case a good editor should make it easy to replace the short form with the long one.

See also

The overview section on foreign functions, list of related functions and commands

---

Name

frac() — return the fractional part of its numeric argument

Synopsis

```
x=frac(y)
```

Description

The `frac`-function takes its argument, removes all the digits to the left of the comma and just returns the digits right of the comma, i.e. the fractional part.

Refer to the example to learn how to rewrite `frac` by employing the `int`-function (which is not suggested anyway).

Example

```
for a=1 to 10
  print frac(sqr(a))
  print sqr(a)-int(sqr(a))
next a
```

The example prints the fractional part of the square root of the numbers between 1 and 10. Each result is computed (and printed) twice: Once by employing the `frac`-function and once by employing the `int`-function.

See also

`int`, `floor`, `ceil`, `round`

G

Name

getbit$() — return a string representing the bit pattern of a rectangle within the graphic window

Synopsis

```
a$=getbit$(10,10,20,20)
a$=getbit$(10,10 to 20,20)
```

Description

The function `getbit` returns a string, which contains the encoded bit-pattern of a rectangle within graphic window; the four arguments specify two opposite corners of the rectangle. The string returned might later be fed to the `putbit`-command.

The `getbit$`-function might be used for simple animations (as in the example below).

Example

```
open window 40,40
fill circle 20,20,18
circle$=getbit$(0,0,40,40)
close window

open window 200,200
for x=1 to 200
  putbit circle$,x,80
next x
```

This example features a circle moving from left to right over the window.

See also

`putbit`

Name

getscreen$() — returns a string representing a rectangular section of the text terminal

Synopsis

```
a$=getscreen$(2,2,20,20)
```

Description

The `getscreen$` function returns a string representing the area of the screen as specified by its four arguments (which specify two opposite corners). I.e. everything you have printed within this rectangle will be encoded in the string returned (including any colour-information).

Like most other commands dealing with advanced text output, `getscreen$` requires, that you have called `clear screen` before.

Example

```
clear screen

for a=1 to 1000:
        print color("red") "1";
        print color("green") "2";
        print color("blue") "3";
next a
screen$=getscreen$(10,10,40,10)
print at(10,10) " Please Press 'y' or 'n' ! "
a$=inkey$
putscreen screen$,10,10
```

This program fills the screen with colored digits and afterwards asks the user for a choice (`Please press 'y' or 'n' !`). Afterwards the area of the screen, which has been overwritten by the question will be restored with its previous contents, whhch had been saved via `getscreen$`.

See also

`putscreen$`

Name

glob() — check if a string matches a simple pattern

Synopsis

`if (glob(string$,pattern$)) …`

Description

The `glob`-function takes two arguments, a string and a (glob-) pattern, and checks if the string matches the pattern. However `glob` does *not* employ the powerful rules of regular expressions; rather it has only two *special* characters: `*` (which matches any number (even zero) of characters) and `?` (which matches exactly a single character).

Example

```
for a=1 to 10
  read string$,pattern$
  if (glob(string$,pattern$)) then
    print string$," matches ",pattern$
  else
    print string$," does not match ",pattern$
  endif
next a

data "abc","a*"
data "abc","a?"
```

```
data "abc","a??"
data "abc","*b*"
data "abc","*"
data "abc","???"
data "abc","?"
data "abc","*c"
data "abc","A*"
data "abc","????"
```

This program checks the string `abc` against various patterns and prints the result. The output is:

```
abc matches a*
abc does not match a?
abc matches a??
abc matches *b*
abc matches *
abc matches ???
abc does not match ?
abc matches *c
abc does not match A*
abc does not match ????
```

See also

There are no related commands.

Name

gosub — continue execution at another point within your program (and return later)

Synopsis

```
gosub foo

…

label foo
…
return
```

Description

`gosub` remembers the current position within your program and then passes the flow of execution to another point (which is normally marked with a `label`). Later, when a `return`-statement is encountered, the execution is resumed at the previous location.

`gosub` is the traditional command for calling code, which needs to be executed from various places within your program. However, with *subroutines* yabasic offers a much more flexible way to achieve this (and more). Therefore `gosub` must to be considered obsolete.

Example

```
print "Do you want to exit ? "
gosub ask
if (r$="y") exit

label ask
input "Please answer yes or no, by typing 'y' or 'n': ",r$
return
```

See also

return, goto, sub, label, on gosub

Name

goto — continue execution at another point within your program (and never come back)

Synopsis

```
goto foo

…

label foo
```

Description

The goto-statement passes the flow of execution to another point within your program (which is normally marked with a label).

goto is normally considered obsolete and harmful, however in yabasic it may be put to the good use of leaving loops (e.g. while or for) prematurely. Note however, that subroutines may *not* be left with the goto-statement.

Example

```
print "Please press any key to continue."
print "(program will continue by itself within 10 seconds)"
for a=1 to 10
  if (inkey$(1)<>"") then goto done
next a
label done
print "Hello World !"
```

Here the goto-statement is used to leave the for-loop prematurely.

See also

gosub, on goto

H

Name

hex$() — convert a number into hexadecimal

Synopsis

```
print hex$(foo)
```

Description

The hex$-function converts a number into a string with its hexadecimal representation. hex$ is the inverse of the dec-function.

Example

```
open 1,"foo"
while !eof(1)
  print right$("0"+hex$(peek(1)),2)," ";
  i=i+1
  if (mod(i,10)=0) print
end while
print
```

This program reads the file foo and prints its output as a *hex*-dump using the hex-function.

See also

decbin, numbers with base 2 or 16

I

Name

if — evaluate a condition and execute statements or not, depending on the result

Synopsis

```
if (…) then
  …
endif

if (…) …

if (…) then
```

```
  …
else
  …
endif

if (…) then
  …
elsif (…)
  …
elsif (…) then
  …
else
  …
endif
```

Description

The `if`-statement is used to evaluate a conditions and take actions accordingly. (As an aside, please note that there is no real difference between conditions and expressions.)

There are two major forms of the `if`-statement:

- The *one-line*-form *without* the keyword `then`:

  ```
  if (…) …
  ```

  This form evaluates the condition and if the result is `true` executes all commands (separated by colons) upt to the end of the line. There is neither an `endif` keyword nor an `else`-branch.

- The *multi-line*-form *with* the keyword `then`:

  ```
  if (…) then … elsif (…) … else … endif
  ```

  (where `elsif` and `else` are optional, whereas `endif` is not.

  According to the requirements of your program, you may specify:

  - `elsif(…)`, which specifies a condition, that will be evaluated only if the condition(s) within `if` or any preceding `elsif` did not match.
  - `else`, which introduces a sequence of commands, that will be executed, if none of the conditions above did match.
  - `endif` is required and ends the `if`-statement.

Example

```
input "Please enter a number between 1 and 4: " a
if (a<=1 or a>=4) error "Wrong, wrong !"
if (a=1) then
  print "one"
elsif (a=2)
```

```
  print "two"
elsif (a=3)
  print "three"
else
  print "four"
endif
```

The input-number between 1 and 4 is simply echoed as text (`one`, `two`, ...). The example demonstrates both forms (*short* and *long*) of the `if`-statement (Note however, that the same thing can be done, probably somewhat more elegant, with the `switch`-statement).

See also

`else`, `elsif`, `endif`, `conditions and expressions`.

Name

import — import a library

Synopsis

```
import foo
```

Description

The `import`-statement imports a library. It expects a single argument, which must be the name of a library (without the trailing `.yab`). This library will then be read and parsed and its subroutines (and variables) will be made available within the importing program. Most of the time this will be the main program, but libraries my also import and use other libraries.

Libraries will first be searched in three locations in order:

- The current directory, i.e. the directory from which you have invoked yabasic)
- The directory, where your main program lives. This can be different from the first directory, if you specify a path for your main program, e.g. like `yabasic foo/bar.yab`.
- Finally, libraries are searched within a special directory, whose exact location depends on your system or options when invoking yabasic. Typical values would be `/usr/lib` under Unix or `C:\yabasic\lib` under Windows. Invoking **`yabasic --help`** will show the correct directory. The location of this directory may be changed with the option `--librarypath` (see options).

Example

Lets say you have a yabasic-program `foo.yab`, which imports a library `lib.yab`. `foo.yab` would read:

```
import lib

rem  This works
lib.x(0)

rem  This works too
x(1)

rem  And this
lib.y(2)

rem  But this not !
y(3)
```

Now the library `lib.yab`:

```
rem  Make the subroutine x easily available outside this library
export sub x(a)
  print a
  return
end sub

rem  sub y must be referenced by its full name
rem  outside this library
sub y(a)
  print a
  return
end sub
```

This program produces an error:

```
0
1
2
---Error in foo.yab, line 13: can't find subroutine 'y'
---Dump: sub y() called in foo.yab,13
---Error: Program stopped due to an error
```

As you may see from the error message, yabasic is unable to find the subroutine `y` without specifying the name of the library (i.e. `lib.y`). The reason for this is, that `y`, other than `x`, is *not* exported from the library `lib.yab` (using the `export`-statement).

See also

Libraries, `export`, `sub`

Name

inkey$ — wait, until a key is pressed

Synopsis

```
clear screen
```

```
foo$=inkey$
inkey$
foo$=inkey$(bar)
inkey$(bar)
```

Description

The `inkeys$`-function waits, until the user presses a key on the keyboard or a button of his mouse, and returns this very key. An optional argument specifies the number of seconds to wait; if omitted, `inkey$` will wait indefinitely.

`inkey$` may only be used, if `clear screen` has been called at least once.

For normal keys, yabasic simply returns the key, e.g. `a`, `1` or `!`. For function keys you will get `f1`, `f2` and so on. Other special keys will return these strings respectively: `enter`, `backspace`, `del`, `esc`, `scrnup` (for *screen up*), `scrndown` and `tab`. Modifier keys (e.g. `ctrl`, `alt` or `shift`) by themselves can *not* be detected (e.g. if you simultaneously press `shift` and '`a`', `inkey$` will return the letter '`A`' instead of '`a`' of course).

If a graphical window has been opened (via `open window`) any mouseclick within this window will be returned by `inkey$` too. The string returned (e.g. `MB1d+0:0028,0061`, `MB2u+0:0028,0061` or `MB1d+1:0028,0061`) is constructed as follows:

- Every string associated with a mouseclick will start with the fixed string `MB`
- The next digit (`1`, `2` or `3`) specifies the mousebutton pressed.
- A single letter, `d` or `u`, specifies, if the mousebutton has been pressed or released: `d` stands for *down*, i.e. the mousebutton has been pressed; `u` means *up*, i.e. the mousebutton has been released.
- The plus-sign ('`+`'), which follows is always fixed.
- The next digit (in the range 0 to 7) encodes the modifier keys pressed, where `1` stands for `shift`, `2` stands for `alt` and `4` stands for `ctrl`.
- The next four digits (e.g. `0028`) contain the x-position, where the mousebutton has been pressed.
- The comma to follow is always fixed.
- The last four digits (e.g. `0061`) contain the y-position, where the mousebutton has been pressed.

All those fields are of fixed length, so you may use functions like `mid$` to extract certain fields. However, note that with `mousex`, `mousey`, `mouseb` and `mousemod` there are specialized functions to return detailed information about the mouseclick. Finally it should be noted, that `inkey$` will only register mouseclicks within the graphic-window;

mouseclicks in the text-window cannot be detected.

`inkey$` accepts an optional argument, specifying a timeout in seconds; if no key has been pressed within this span of time, an empty string is returned. If the timeout-argument is omitted, `inkey$` will wait for ever.

Example

```
clear screen
open window 100,100
print "Press any key or press 'q' to stop."
repeat
  a$=inkey$
  print a$
until a$="q"
```

This program simply returns the key pressed. You may use it, to learn, which strings are returned for the special keys on your keyboard (e.g. function-keys).

See also

`clear screen`,`mousex`, `mousey`, `mouseb`, `mousemod`

Name

input — read input from the user (or from a file) and assign it to a variable

Synopsis

```
input a
input a,b,c
input a$
input "Hello" a
input #1 a$
```

Description

`input` reads the new contents of one or many (numeric- or string-) variables, either from the keyboard (i.e. from *you*) or from a file. An optional first string-argument specifies a prompt, which will be issued before reading any contents.

If you want to read from an open file, you need to specify a hash ('#'), followed by the number, under which the file has been opened.

Note, that the input is split at spaces, i.e. if you enter a whole line consisting of many space-separated word, the first `input`-statement will only return the first word; the other words will only be returned on subsequent calls to `input`; the same applies, if a single `input` reads

multiple variables: The first variable gets only the first word, the second one the second word, and so on. If you don't like this behaviour, you may use `line input`, which returns a whole line (including embedded spaces) at once.

Example

```
input "Please enter the name of a file to read: " a$
open 1,a$
while !eof(1)
  input #1 b$
  print b$
wend
```

If this program is stored within a file `test.yab` and you enter this name when prompted for a file to read, you will see this output:

```
Please enter the name of a file to read: t.yab
input
"Please
enter
the
name
of
a
file
to
read:
"
a$
open
1,a$
while
!eof(1)
input
#1
b$
print
b$
wend
```

See also

line input

Name

instr() — searches its second argument within the first; returns its position if found

Synopsis

```
print instr(a$,b$)
if (instr(a$,b$)) …
pos=instr(a$,b$,x)
```

Description

The `instr`-functions requires two string arguments and searches the second argument within the first. If the second argument can be found within the first, the position is returned (counting from one). If it can not be found, the `instr`-function returns 0; this makes this function usable within the condition of an `if`-statement (see the example below).

If you supply a third, numeric argument to the `instr`-function, it will be used as a starting point for the search. Therefore `instr("abcdeabcdeabcde","e",8)` will return 10, because the search for an "`e`" starts at position 8 and finds the "`e`" at position 10 (and not the one at position 5).

Example

```
input "Please enter a text containing the string 'cat': " a$
if (instr(a$,"cat")) then
  print "Well done !"
else
  print "No cat in your input ..."
endif
```

See also

rinstr

Name

int() — return the integer part of its single numeric argument

Synopsis

```
print int(a)
```

Description

The `int`-function returns only the digits before the comma; `int(2.5)` returns `2` and `int(-2.3)` returns `-2`.

Example

```
input "Please enter a whole number between 1 and 10: " a
if (a=int(a) and a>=1 and a<=10) then
  print "Thanx !"
else
  print "Never mind ..."
endif
```

See also

frac, floor, ceil, round

L

Name

label — mark a specific location within your program for `goto`, `gosub` or `restore`

Synopsis

```
label foo

…

goto foo
```

Description

The `label`-command can be used to give a name to a specific location within your program. Such a position might be referred from one of three commands: `goto`, `gosub` and `restore`.

You may use labels safely within libraries, because a label (e.g. `foo`) does not collide with a label with the same name within the main program or within another library; yabasic will not mix them up.

As an aside, please note, that line numbers are a special (however deprecated) case of labels; see the second example below.

Example

```
for a=1 to 100
  if (ran(10)>5) goto done
next a
label done

10 for a=1 to 100
20   if (ran(10)>5) goto 40
30 next a
40
```

Within this example, the `for`-loop will probably be left prematurely with a `goto`-statement. This task is done *twice*: First with labels and then again with line numbers.

See also

gosub, goto.

Name

left$() — return (*or change*) left end of a string

Synopsis

```
print left$(a$,2)
left$(b$,3)="foobar"
```

Description

The `left$`-function accepts two arguments (a string and a number) and returns the part from the left end of the string, whose length is specified by its second argument. Loosely spoken, it simply returns the requested number of chars from the left end of the given string.

Note, that the `left$`-function can be assigned to, i.e. it may appear on the left hand side of an assignment. In this way it is possible to change a part of the variable used within the `left$`-function. Note, that that way the *length* of the string cannot be changed, i.e. characters might be overwritten, but not added. For an example see below.

Example

```
input "Please answer yes or no: " a$
l=len(a$):a$=lower$(a$):print "Your answer is ";
if (left$("yes",l)=a$ and l>=1) then
  print "yes"
elsif (left$("no",l)=a$ and l>=1) then
  print "no"
else
  print "?"
endif
```

This example asks a simple yes/no question and goes some way to accept even incomplete input, while still being able to reject invalid input.

This second example demonstrates the capability to *assign* to the `left$`-function.

```
a$="Heiho World !"
print a$
left$(a$,5)="Hello"
print a$
```

See also

right$, mid$

Name

len() — return the length of a string

Synopsis

```
x=len(a$)
```

Description

The `len`-function returns the length of its single string argument.

Example

```
input "Please enter a password: " a$
if (len(a$)<6) error "Password too short !"
```

This example checks the length of the password, that the user has entered.

See also

`left$`, `right$` and `mid$`,

Name

line — draw a line

Synopsis

```
open window 100,100
line 0,0,100,100
line 0,0 to 100,100
new curve
line 100,100
line to 100,100

open window 100,100
clear line 0,0,100,100
clear line 0,0 to 100,100
new curve
clear line 100,100
clear line to 100,100
```

Description

The `line`-command draws a line. Simple as this is, the `line`-command has a large variety of forms as they are listed in the synopsis above. Lets look at them a little closer:

- A line has a starting and an end point; therefore the `line`-command (normally) needs four numbers as arguments, representing these two points. This is the first form appearing

within the synopsis.

- You may separate the two points with either ',' or `to`, which accounts for the second form of the `line`-command.
- The `line`-command may be used to draw a connected sequence of lines with a sequence of commands like `line x,y`; Each command will draw a line from the point where the last `line`-command left off, to the point specified in the arguments. Note, that you need to use the command `new curve` before you may issue such a `line`-command. See the example below.
- You may insert the word `to` for beauty: `line to x,y`, which does exactly the same as `line x,y`
- Finally, you may choose not to draw, but to erase the lines; this can be done by prepending the phrase `clear`. This account for all the other forms of the `line`-command.

Example

```
open window 200,200
line 10,10 to 10,190
line 10,190 to 190,190
new curve
for a=0 to 360
  line to 10+a*180/360,100+60*sin(a*pi/180)
next a
```

This example draws a sine-curve (with an offset in x- and y-direction). Note, that the first `line`-command after `new curve` does not draw anything. Only the coordinates will be stored. The second iteration of the loop then uses these coordinates as a starting point for the first line.

See also

`new curve`, `close curve`, `open window`

Name

line input — read in a whole line of text and assign it to a variable

Synopsis

```
line input a
line input a$
line input "Hello" a
line input #1 a$
```

Description

In most respects `line input` is like the `input`-command: It reads the new contents of a variable, either from keyboard or from a file. However, `line input` always reads a complete line and assigns it to its variable. `line input` does not stop reading at spaces and is therefore the best way to read in a string which might contain whitespace. Note, that the final newline is stripped of.

Example

```
line input "Please enter your name (e.g. Frodo Beutelin): " a$
print "Hello ",a$
```

Note that the usage of `line input` is essential in this example; a simple `input`-statement would only return the string up to the first space, e.g. `Frodo`.

See also

`input`

Name

local — mark a variable as local to a subroutine

Synopsis

```
sub foo()

  local a,b,c$,d(10),e$(5,5)

  …

end sub
```

Description

The `local`-command can (and should be) used to mark a variable (or array) as *local* to the containing subroutine. This means, that a local variable in your subroutine is totally different from a variable with the same name within your main program. Variables which are known everywhere within your program are called *global* in contrast.

Declaring variables within the subroutine as *local* helps to avoid hard to find bugs; therefore local variables should be used whenever possible.

Note, that the parameters of your subroutines are always local.

As you may see from the example, local arrays may be created without using the keyword `dim` (which is required only for global arrays).

Example

```
a=1
b=1
print a,b
foo()
print a,b

sub foo()
  local a
  a=2
  b=2
end sub
```

This example demonstrates the difference between `local` and global variables; it produces this output:

```
 1 1
 1 2
```

As you may see, the content of the global variable `a` is unchanged after the subroutine `foo`; this is because the assignment `a=2` within the subroutine affects the *local* variable `a` only and not the global one. However, the variable `b` is never declared local and therefore the subroutine changes the global variable, which is reflected in the output of the second print-statement.

See also

`sub`, `static`, `dim`

Name

log() — compute the natural logarithm

Synopsis

```
a=log(x)
a=log(x,base)
```

Description

The `log`-function computes the logarithm of its first argument. The optional second argument gives the base for the logarithm; if this second argument is omitted, the *euler*-constant 2.71828… will be taken as the base.

Example

```
open window 200,200
```

```
for x=10 to 190 step 10:for y=10 to 190 step 10
  r=3*log(1+x,1+y)
  if (r>10) r=10
  if (r<1) r=1
  fill circle x,y,r
next y:next x
```

This draws another nice plot.

See also

exp

Name

loop — marks the end of an infinite loop

Synopsis

```
do
  …
loop
```

Description

The `loop`-command marks the ends of a loop (which is started by `do`), wherein all statements within the loop are repeated forever. In this respect the `do loop`-loop is infinite, however, you may leave it anytime via `break` or `goto`.

Example

```
print "Hello, I will throw dice, until I get a 2 ..."
do
  r=int(ran(6))+1
  print r
  if (r=2) break
loop
```

See also

do, for, repeat, while, break

Name

lower$() — convert a string to lower case

Synopsis

```
l$=lower$(a$)
```

Description

The `lower$`-function accepts a single string-argument and converts it to all lower case.

Example

```
input "Please enter a password: " a$
if (a$=lower$(a$)) error "Your password is NOT mixed case !"
```

This example prompts for a password and checks, if it is really lower case.

See also

`upper$`

Name

ltrim$() — trim spaces at the left end of a string

Synopsis

```
a$=ltrim$(b$)
```

Description

The `ltrim$`-function removes all whitespace from the left end of a string and returns the result.

Example

```
input "Please answer 'yes' or 'no' : " a$
a$=lower$(ltrim$(rtrim$(a$)))
if (len(a$)>0 and a$=left$("yes",len(a$))) then
  print "Yes ..."
else
  print "No ..."
endif
```

This example prompts for an answer and removes any spaces, which might precede the input; therefore it is even prepared for the (albeit somewhat pathological case, that the user first hits *space* before entering his answer.

See also

rtrim$, trim$

M

Name

max() — return the larger of its two arguments

Synopsis

```
print max(a,b)
```

Description

Return the *maximum* of its two arguments.

Example

```
dim m(10)
for a=1 to 1000
  m=0
  For b=1 to 10
    m=max(m,ran(10))
  next b
  m(m)=m(m)+1
next a

for a=1 to 9
  print a,": ",m(a)
next a
```

Within the inner `for`-loop (the one with the loop-variable `b`), the example computes the maximum of 10 random numbers. The outer loop (with the loop variable `a`) now repeats this process 1000 times and counts, how often each maximum appears. The last loop finally reports the result.

Now, the interesting question would be, which will be approached, when we increase the number of iterations from thousand to infinity. Well, maybe someone could just tell me `:-)`

See also

min

Name

mid$() — return (*or change*) characters from within a string

Synopsis

```
print mid$(a$,2,1)
print mid$(a$,2)
mid$(a$,5,3)="foo"
mid$(a$,5)="foo"
```

Description

The `mid$`-function requires three arguments: a string and two numbers, where the first number specifies a position within the string and the second one gives the number of characters to be returned; if you omit the third argument, the `mid$`-function returns all characters up to the end of the string.

Note, that you may assign to the `mid$`-function, i.e. `mid$` may appear on the left hand side of an assignment. In this way it is possible to change a part of the variable used within the `mid$`-function. Note, that that way the *length* of the string cannot be changed, i.e. characters might be overwritten, but not added. For an example see below.

Example

```
input "Please enter a string: " a$
for a=1 to len(a$)
  if (instr("aeiou",lower$(mid$(a$,a,1)))) mid$(a$,a,1)="e"
next a
print "When you turn everything to lower case and"
print "replace every vowel with 'e', your input reads:"
print
print a$
```

This example transforms the input string a bit, using the `mid$`-function to retrieve a character from within the string as well as to change it.

See also

`left$` and `right$`.

Name

min() — return the smaller of its two arguments

Synopsis

```
print min(a,b)
```

Description

Return the *minimum* of its two argument.

Example

```
dim m(10)
for a=1 to 1000
  m=min(ran(10),ran(10))
  m(m)=m(m)+1
next a

for a=1 to 9
  print a,": ",m(a)
next a
```

For each iteration of the loop, the lower of two random number is recorded. The result is printed at the end.

See also

max

Name

mod — compute the remainder of a division

Synopsis

```
print mod(a,b)
```

Description

The `mod`-function divides its two arguments and computes the remainder. Note, that `a/b-int(a/b)` and `mod(a,b)` are always equal.

Example

```
clear screen
print at(10,10) "Please wait ";
p$="-\|/"
for a=1 to 100
  rem  ... do something lengthy here, or simply sleep :-)
  pause(1)
  print at(22,10) mid$(p$,1+mod(a,4))
next a
```

This example executes some time consuming action within a loop (in fact, it simply sleeps) and gives the user some indication of progress by displaying a rotating bar (that's where the `mod`-function comes into play).

See also

int, frac, round

Name

mouseb — extract the state of the mousebuttons from a string returned by `inkey$`

Synopsis

```
inkey$
print mouseb()
print mouseb
a$=inkey$
print mouseb(a$)
```

Description

The `mouseb`-function is a helper function for decoding part of the (rather complicated) strings, which are returned by the `inkey$`-function. If a mousebutton has been pressed, the `mouseb`-function returns the number (1,2 or 3) of the mousebutton, when it is pressed and returns its negative (-1,-2 or -3), when it is released.

The `mouseb`-function accepts zero or one arguments. A single argument should be a string returned by the `inkey$`-function; if `mouseb` is called without any arguments, it returns the values from the last call to `inkey$`, which are stored implicitly and internally by yabasic.

> **Note**
>
> Note however, that the value returned by the `mouseb`-function does *not* reflect the *current* state of the mousebuttons. It rather extracts the information from the string passed as an argument (or from the last call to the `inkey$`-function, if no argument is passed). So the value returned by `mouseb` reflects the state of the mousebuttons at the time the `inkey$`-function has been called; as opposed to the time the `mouseb`-function is called.

Example

```
open window 200,200
clear screen
print "Please draw lines; press (and keep it pressed)"
print "the left mousebutton for the starting point,"
print "release it for the end-point."
do
  if (mouseb(release$)=1) press$=release$
  release$=inkey$
  if (mouseb(release$)=-1) then
    line mousex(press$),mousey(press$) to mousex(release$),mousey(release$)
  endif
loop
```

This is a maybe the most simplistic line-drawing program possible, catching presses as well as releases of the first mousebutton.

See also

inkey$, mousex, mousey and mousemod

Name

mousemod — return the state of the modifier keys during a mouseclick

Synopsis

```
inkey$
print mousemod()
print mousemod
a$=inkey$
print mousemod(a$)
```

Description

The mousemod-function is a helper function for decoding part of the (rather complicated) strings, which are returned by the inkey$-function if a mousebutton has been pressed. It returns the state of the keyboard modifiers (shift, ctrl or alt): If the shift-key is pressed, mousemod returns 1, for the alt-key 2 and for the ctrl-key 4. If more than one key is pressed, the sum of these values is returned, e.g. mousemod returns 5, if shift and ctrl are pressed simultaneously.

The mousemod-function accepts zero or one arguments. A single argument should be a string returned by the inkey$-function; if mousemod is called without any arguments, it returns the values from the last call to inkey$ (which are stored implicitly and internally by yabasic).

> **Note**
>
> Please see also the Note within the mouseb-function.

Example

```
open window 200,200
clear screen
do
  a$=inkey$
  if (left$(a$,2)="MB") then
    x=mousex(a$)
    y=mousey(a$)
    if (mousemod(a$)=0) then
      circle x,y,20
    else
```

```
      fill circle x,y,20
    endif
  endif
loop
```

This program draws a circle, whenever a mousebutton is pressed; the circles are filled, when any modifier is pressed, and empty if not.

See also

inkey$, mousex, mousey and mouseb

Name

mousex — return the x-position of a mouseclick

Synopsis

```
inkey$
print mousex()
print mousex
a$=inkey$
print mousex(a$)
```

Description

The mousex-function is a helper function for decoding part of the (rather complicated) strings, which are returned by the inkey$-function; It returns the x-position of the mouse as encoded within its argument.

The mousex-function accepts zero or one arguments. A single argument should be a string returned by the inkey$-function; if mousex is called without any arguments, it returns the values from the last call to inkey$ (which are stored implicitly and internally by yabasic).

> **Note**
> Please see also the Note within the mouseb-function.

Example

```
open window 200,200
clear screen
do
  a$=inkey$
  if (left$(a$,2)="MB") then
    line mousex,0 to mousex,200
  endif
loop
```

This example draws vertical lines at the position, where the mousebutton has been pressed.

See also

inkey$, mousemod, mousey and mouseb

Name

mousey — return the y-position of a mouseclick

Synopsis

```
inkey$
print mousey()
print mousey
a$=inkey$
print mousey(a$)
```

Description

The mousey-function is a helper function for decoding part of the (rather complicated) strings, which are returned by the inkey$-function. mousey returns the y-position of the mouse as encoded within its argument.

The mousey-function accepts zero or one arguments. A single argument should be a string returned by the inkey$-function; if mousey is called without any arguments, it returns the values from the last call to inkey$ (which are stored implicitly and internally by yabasic).

> **Note**
> Please see also the Note within the mouseb-function.

Example

```
open window 200,200
clear screen
do
  a$=inkey$
```

```
  if (left$(a$,2)="MB") then
    line 0,mousey to 200,mousey
  endif
loop
```

This example draws horizontal lines at the position, where the mousebutton has been pressed.

See also

inkey$, mousemod, mousex and mouseb

N

Name

new curve — start a new curve, that will be drawn with the line-command

Synopsis

```
new curve
line to x,y
```

Description

The new curve-function starts a new sequence of lines, that will be drawn by repeated line to-commands.

Example

```
open window 200,200
ellipse(100,50,30,60)
ellipse(150,100,60,30)
sub ellipse(x,y,xr,yr)
  new curve
  for a=0 to 2*pi step 0.2
    line to x+xr*cos(a),y+yr*sin(a)
  next a
  close curve
end sub
```

This example defines a subroutine ellipse that draws an ellipse. Within this subroutine, the ellipse is drawn as a sequence of lines started with the new curve command and closed with close curve.

See also

line, close curve

Name

next — mark the end of a for loop

Synopsis

```
for a=1 to 10
next a
```

Description

The `next`-keyword marks the end of a `for`-loop. All statements up to the `next`-keyword will be repeated as specified with the `for`-clause. Note, that the name of the variable is optional; so instead of `next a` you may write `next`.

Example

```
for a=1 to 300000
  for b=1 to 21+20*sin(pi*a/20)
    print "*";
  next b
  print
  sleep 0.1
next a
```

This example simply plots a sine-curve until you fall asleep.

See also

`for`

Name

not — negate a logical expression; can be written as `!`

Synopsis

```
if not a<b then …
bad=!okay
```

Description

The keyword `not` (or `!` for short) is mostly used within conditions (e.g. within `if`- or `while`-statements). There it is employed to negate the condition or expression (i.e. turn `TRUE` into `FALSE` and vice versa)

However `not` can be used within arithmetic calculations too., simply because there is no difference between arithmetic and logical

expressions.

Example

```
input "Please enter three ascending numbers: " a,b,c
if (not (a<b and b<c)) error " the numbers you have entered are not ascending
```

See also

and,or

Name

numparams — return the number of parameters, that have been passed to a subroutine

Synopsis

```
sub foo(a,b,c)
  if (numparams=1) …
  …
end sub
```

Description

Within a subroutine the local variable `numparam` or `numparams` contains the number of parameters, that have been passed to the subroutine. This information can be useful, because the subroutine may have been called with fewer parameters than actually declared. The number of values that actually have been passed while calling the subroutine, can be found in `numparams`.

Note, that arguments which are used in the definition of a subroutine but are left out during a call to it (thereby reducing the value of `numparams`), receive a value of `0` or `""` (empty string) respectively.

Example

```
a$="123456789"
print part$(a$,4)
print part$(a$,3,7)

sub part$(a$,f,t)
  if (numparams=2) then
    return mid$(a$,f)
  else
    return mid$(a$,f,t-f+1)
  end if
end sub
```

When you run this example, it will print `456789` and `34567`. Take a look at the subroutine `part$`, which returns part of the string which has been passed as an argument. If (besides the string) two numbers are passed, they define the starting and end position of the substring, that will be returned. However, if only one number is passed, the rest of the string, starting from this position will be returned. Each of these cases is recognized with the help of the `numparams`-variable.

See also

`sub`

O

Name

on goto — jump to one of multiple `gosub`-targets

Synopsis

```
on a gosub foo,bar,baz
  …
label foo
  …
return

label bar
  …
return

label baz
  …
return
```

Description

The `on gosub` statement uses its numeric argument (the one between `on` and `gosub`) to select an element from the list of labels, which follows after the `gosub`-keyword: If the number is 1, the program does a `gosub` to the first label; if the number is 2, to the second and, so on. if the number is zero or less, the program continues at the position of the first label; if the number is larger than the total count of labels, the execution continues at the position of the last label; i.e. the first and last label in the list constitute some kind of fallback-slot.

Note, that the `on gosub`-command can no longer be considered *state of the art*; people (not me !) may even start to mock you, if you use it.

Example

```
do
  print "Please enter a number between 1 and 3: "
  print
  input "Your choice " a
```

```
  on a gosub bad,one,two,three,bad
loop

label bad
  print "No. Please between 1 and 3"
return

label one
  print "one"
return

label two
  print "two"
return

label three
  print "three"
return
```

Note, how invalid input (a number less than 1, or larger than 3) is automatically detected.

See also

goto, on gosub/function>

Name

on goto — jump to one of many goto-targets

Synopsis

```
on a goto foo,bar,baz
  …
label foo
  …
label bar
  …
label baz
  …
```

Description

The on goto statement uses its numeric argument (the one between on and goto to select an element from the list of labels, which follows after the goto-keyword: If the number is 1, the execution continues at the first label; if the number is 2, at the second, and so on. if the number is zero or less, the program continues at the position of the first label; if the number is larger than the total count of labels, the execution continues at the position of the last label; i.e. the first and last label in the list constitute some kind of fallback-slot.

Note, that (unlike the goto-command) the on goto-command can no longer be considered *state of the art*; people may (not me !) even start to mock you, if you use it.

Example

```
label over
print "Please Select one of these choices: "
print
print "  1 -- show time"
print "  2 -- show date"
print "  3 -- exit"
print
input "Your choice " a
on a goto over,show_time,show_date,terminate,over

label show_time
  print time$()
goto over

label show_date
  print date$()
goto over

label terminate
exit
```

Note, how invalid input (a number less than 1, or larger than 3) is automatically detected; in such a case the question is simply issued again.

See also

goto, on gosub/function>

Name

on interrupt — change reaction on keyboard interrupts

Synopsis

```
on interrupt break
…
on interrupt continue
```

Description

With the `on interrupt`-command you may change the way, how yabasic reacts on a keyboard interrupt; it comes in two variants: `on interrupt break` and `on interrupt continue`. A keyboard interrupt is produced, if you press **`ctrl-C`** on your keyboard; normally (and certainly after you have called `on interrupt break`), yabasic will terminate with an error message. However after the command `on interrupt continue` yabasic ignores any keyboard interrupt. This may be useful, if you do not want your program being interruptible during certain critical operations (e.g. updating of files).

Example

```
print "Please stand by while writing a file with random data ..."
on interrupt continue
open "random.data" for writing as #1
for a=1 to 100
  print #1 ran(100)
  print a," percent done."
  sleep 1
next a
close #1
on interrupt continue
```

This program writes a file with 100 random numbers. The `on interrupt continue` command insures, that the program will not be terminated on a keyboard interrupt and the file will be written entirely in any case. The `sleep`-command just stretches the process artificially to give you a chance to try a **`ctrl-C`**.

See also

There is no related command.

Name

open — open a file

Synopsis

```
open a,"file","r"
open #a,"file","w"
open #a,printer
open "file" for reading as a
open "file" for writing as #a
a=open("file")
a=open("file","r")
if (open(a,"file")) …
if (open(a,"file","w")) …
```

Description

The `open`-command opens a file for reading or writing or a printer for printing text. `open` comes in a wide variety of ways; it requires these arguments:

filenumber

In the synopsis this is `a` or `#a`. In yabasic each file is associated with a number between 1 and a maximum value, which depends on the operating system. For historical reasons the filenumber can be preceded by a hash ('`#`'). Note, that specifying a filenumber is optional; if it is omitted, the `open`-function will return a filenumber, which should then be stored in a variable for

later reference. This filenumber can be a simple number or an arbitrary complex arithmetic expression, in which case braces might be necessary to save yabasic from getting confused.

filename

In the synopsis above this is `"file"`. This string specifies the name of the file to open (note the important caveat on specifying these filenames).

accessmode

In the synopsis this is `"r"`, `"w"`, `for reading` or `for writing`. This string or clause specifies the mode in which the file is opened; it may be one of:

"r"

Open the file for reading (may also be written as `for reading`). If the file does not exist, the command will fail. This mode is the default, i.e. if no mode is specified with the `open`-command, the file will be opened with this mode.

"w"

Open the file for writing (may also be written as `for writing`). If the file does not exist, it will be created.

"a"

Open the file for appending, i.e. what you write to the file will be appended after its initial contents. If the file does not exist, it will be created.

"b"

This letter may not appear alone, but may be combined with the other letters (e.g. `"rb"`) to open a file in binary mode (as opposed to text mode).

As you may see from the synopsis, the `open`-command may either be called as a command (without braces) or as a function (with braces). If called as a function, it will return the filenumber or zero if the operation fails. Therefore the `open`-function may be used within the condition of an `if`-statement.

If the `open`-command fails, you may use `peek("error")` to retrieve the exact nature of the error.

Furthermore note, that there is another, somewhat separate usage of the `open`-command; if you specify the bareword `printer` instead of a filename, the command opens a printer for printing text. Every text (and only text) you print to this file will appear on your printer. Note, that this is very different from printing graphics, as can be done with `open printer`.

Example

```
open "foo.bar" for writing as #1
print #1 "Hallo !"
close #1
if (not open(1,"foo.bar")) error "Could not open 'foo.bar' for reading"
while not eof(1)
  line input #1 a$
  print a$
wend
```

This example simply opens the file `foo.bar`, writes a single line, reopens it and reads its contents again.

See also

`close`, `print`, `peek`, `peek("error")` and `open printer`

Name

open printer — open printer for printing graphics

Synopsis

```
open printer
open printer "file"
```

Description

The `open printer`-command opens a printer for printing graphics. The command requires, that a graphic window has been opened before. Everything that is drawn into this window will then be sent to the printer too.

A new piece of paper may be started with the `clear window`-command; the final (or only) page will appear after the `close printer`-command.

Note, that you may specify a filename with `open printer`; in that case the printout will be sent to a filename instead to a printer. Your program or the user will be responsible for sending this file to the printer afterwards.

If you use yabasic under Unix, you will need a postscript printer (because yabasic produces postscript output). Alternatively you may use *ghostscript* to transform the postscript file into a form suitable for your printer; but that is beyond the responsibility of yabasic.

Example

```
open window 200,200
open printer
```

```
line 0,0 to 200,200
text 100,100,"Hallo"
close window
close printer
```

This example will open a window, draw a line and print some text within; everything will appear on your printer too.

See also

close printer

Name

open window — open a graphic window

Synopsis

```
open window x,y
open window x,y,"font"
```

Description

The `open window`-command opens a window of the specified size. Only one window can be opened at any given moment of time.

An optional third argument specifies a font to be used for any text within the window. It can however be changed with any subsequent text-command.

Example

```
for a=200 to 400 step 10
  open window a,a
  for b=0 to a
    line 0,b to a,b
    line b,0 to b,a
  sleep 0.1
  close window
next a
```

See also

close window, text

Name

or — logical or, used in conditions

Synopsis

```
if a or b …
while a or b …
```

Description

Used in conditions (e.g within `if` or `while`) to join two expressions. Returns `true`, if either its left or its right or both arguments are `true`; returns `false` otherwise.

Example

```
input "Please enter a number"
if (a>9 or a<1) print "a is not between 1 and 9"
```

See also

`and`,`bitnot`

Name

or() — arithmetic or, used for bit-operations

Synopsis

```
x=or(a,b)
```

Description

Used to compute the bitwise `or` of both its argument. Both arguments are treated as binary numbers (i.e. a sequence of digits 0 and 1); a bit of the resulting value will then be 1, if any of its arguments has 1 at this position in their binary representation.

Note, that both arguments are silently converted to integer values and that negative numbers have their own binary representation and may lead to unexpected results when passed to `or`.

Example

```
print or(14,3)
```

This will print `15`. This result is clear, if you note, that the binary representation of 14 and 3 are 1110 and 0011 respectively; this will yield 1111 in binary representation or 15 as decimal.

See also

and, eor and bitnot

P

Name

pause — pause, sleep, wait for the specified number of seconds

Synopsis

```
pause 5
```

Description

The pause-command has many different names: You may write pause, sleep or wait interchangeably; whatever you write, yabasic will always do exactly the same.

The pause-command will simply wait for the specified number of seconds. This may be a fractional number, so you may well wait less than a second. However, if you try to pause for a smaller and smaller interval (e.g. 0.1 seconds, 0.01 seconds, 0.001 seconds and so on) you will find that at some point yabasic will not wait at all. The minimal interval that can be waited depends on the system (Unix, Windows) you are using.

The pause-command cannot be interrupted. However, sometimes you may want the wait to be interruptible by simply pressing a key on the keyboard. In such cases you should consider using the inkey$-function, with a number of seconds as an argument).

Example

```
deg=0
do
  maxx=44+40*sin(deg)
  for x=1 to maxx
    print "*";
  next x
  pause 0.1+(maxx*maxx/(4*84*84))
  print
  deg=deg+0.1
loop
```

This example draws a sine-curve; due to the pause-statement the speed of drawing varies in the same way as the speed of a ball might vary, if it would roll along this curve under the influence of gravity.

See also

sleep, wait

Name

peek — retrieve various internal information

Synopsis

```
print peek("foo")
a=peek(#1)
```

Description

The peek-function has many different and mostly unrelated uses. It is a kind of grab-bag for retrieving all kinds of numerical information, internal to yabasic. The meaning of the numbers returned be the peek-function depends on the string or number passed as an argument.

peek always returns a number, however the closely related peek$-function exists, which may be used to retrieve string information from among the internals of yabasic. Finally note, that some of the values which are retrieved with peek may even be changed, using the poke-function.

There are two variants of the peek-function: One expects an integer, positive number and is described within the first entry of the list below. The other variant expects one of a well defined set of strings as described in the second and all the following entries of the list below.

peek(a)

> Read a single byte (a number between 0 and 255) from the file a (which must be open of course). You may use the chr$-function to convert this byte to a string of one character.
>
> As a special case, if the argument is zero: read a single byte from stdin.

peek("argument")

> Return the number of arguments, that have been passed to yabasic at invocation time. E.g. if yabasic has been called like this: yabasic foo.yab bar baz, then peek("argument") will return 2. This is because foo.yab is treated as the name of the program to run, whereas bar and baz are considered arguments to the program, which are passed on the command line. *Note*, that for windows-users, who tend to click on the icon (as opposed to starting yabasic on the command line), this peekwill mostly return 0.
>
> The function peek("argument") can be written as peek("arguments") too.
>
> You will want to check out the corresponding function

peek$("argument") to actually *retrieve* the arguments. *Note*, that each call to peek$("argument") reduces the number returned by peek("argument").

peek("error")

Return a number specifying the nature of the last error in an open- or seek-statement. Normally an error within an open-statement immediately terminates your program with an appropriate error-message, so there is no chance and no need to learn more about the nature of the error. However, if you use open as a condition (e.g. if (open(#1,"foo")) …) the outcome (success or failure) of the open-operation will determine, if the condition evaluates to true or false. If now such an operation fails, your program will not be terminated and you might want to learn the reason for failure. This reason will be returned by peek("error") (as a number) or by peek$("error") (as a string)

The table below shows the various error codes; the value returned by peek$("error") explains the nature of the error. Note, that the codes 10,11 and 12 refer to the seek-command.

Table 7.1. Error codes

peek("error")	peek$("error")	Explanation
2	Stream already in use	Do not try to open one and the same filenumber twice; rather close it first.
3	'x' is not a valid filemode	The optional *filemode* argument, which may be passed to the open-function, has an invalid value
4	could not open 'foo'	The open-call did not work, no further explanation is available.
5	reached maximum number of open files	You have opened more files than your operating system permits.
6	cannot open printer: already printing graphics	The commands open printer and open #1,printer both open a printer (refer to their description for the difference). However, only one can be active at a time; if you try to do both at the same time, you will receive this error.
7	could not open line printer	Well, it simply did not work.
9	invalid stream number	An attempt to use an invalid (e.g. negative) stream number; example:

peek("error")	peek$("error")	Explanation
		open(-1,"foo")
10	could not position stream x to byte y	seek did not work.
11	stream x not open	You have tried to seek within a stream, that has not been opened yet.
12	seek mode 'x' is none of begin,end,here	The argument, which has been passed to seek is invalid.

peek("fontheight")

Return the height of the font used within the graphic window. If none is open, this peek will return the height of the last font used or 10, if no window has been opened yet.

peek("screenheight")

Return the height in characters of the window, wherein yabasic runs. If you have not called clear screen yet, this peekwill return 0, regardless of the size of your terminal.

peek("screenwidth")

Return the width in characters of the window, wherein yabasic runs. If you have not called clear screen yet, this peekwill return 0, regardless of the size of your terminal.

peek("secondsrunning")

Return the number of seconds that have passed since the start of yabasic.

peek("millisrunning")

Return the number of milliseconds, that have passed since the start of yabasic.

peek("version")

Return the version number of yabasic, e.g. 2.77. See also the related peek$("version"), which returns nearly the same information (plus the patchlevel) as a string, e.g. "2.77.1".

peek("winheight")

Return the height of the graphic-window in pixels. If none is open, this peek will return the height of the last window opened or 100, if none has been opened yet.

peek("winwidth")

Return the width of the graphic-window in pixels. If none is

open, this peek will return the width of the last window opened or 100, if none has been opened yet.

peek("isbound")

Return true, if the executing yabasic-program is part of a standalone program; see the section about creating a *standalone*-program for details.

peek("last_foreign_function_call_okay")

Check for error: If passing "no_error" to foreign_function_call, any error (e.g. failure to load the specified library), will *not* terminate your yabasic-program but rather store a descriptive error message away for later retrieval. Later on you may then check peek("last_foreign_function_call_okay") to find out, if something went wrong and retrieve a description with peek$("last_foreign_function_call_error_text").

This peek can be abbreviated as peek("last_frnfn_call_okay")

Example

```
open "foo" for reading as #1
open "bar" for writing as #2
while not eof(#1)
  poke #2,chr$(peek(#1));
wend
```

This program will copy the file foo byte by byte to bar.

Note, that each peek does something entirely different, and only one has been demonstrated above. Therefore you need to make up examples yourself for all the other peeks.

See also

peek$, poke, open

Name

peek$ — retrieve various internal string-information

Synopsis

```
print peek$("foo")
```

Description

The peek$-function has many different and unrelated uses. It is a kind

of grab-bag for retrieving all kinds of string information, internal to yabasic; the exact nature of the strings returned be the `peek$`-function depends on the string passed as an argument.

`peek$` always returns a string, however the closely related `peek`-function exists, which may be used to retrieve numerical information from among the internals of yabasic. Finally note, that some of the values which are retrieved with `peek$` may even be changed, using the `poke`-function.

The following list shows all possible arguments to `peek$`:

`peek$("infolevel")`

Returns either `"debug"`, `"note"`, `"warning"`, `"error"` or `"fatal"`, depending on the current infolevel. This value can be specified with an option on the command line or changed during the execution of the program with the corresponding `poke`; however, normally only the author of yabasic (*me* !) would want to change this from its default value `"warning"`.

`peek$("textalign")`

Returns one of nine possible strings, specifying the default alignment of text within the graphics-window. The alignment-string returned by this `peek` describes, how the `text`-command aligns its string-argument with respect to the coordinates supplied. However, this value does *not apply*, if the `text`-command explicitly specifies an alignment. Each of these strings is two characters long. The first character specifies the horizontal alignment and can be either `l`, `r` or `c`, which stand for *left*, *right* or *center*. The second character specifies the vertical alignment and can be one of `t`, `b` or `c`, which stand for *top*, *bottom* or *center* respectively.

You may change this value with the corresponding command `poke "textalign",…`; the initial value is `lb`, which means the top of the left and the top edge if the text will be aligned with the coordinates, that are specified within the `text`-command.

`peek$("windoworigin")`

This `peek` returns a two character string, which specifies the position of the origin of the coordinate system of the window; this string might be changed with the corresponding command `poke "windoworigin",x,y` or specified as the argument of the `origin` command; see there for a detailed description of the string, which might be returned by this `peek`.

`peek$("program_name")`

Returns the name of the yabasic-program that is currently executing; typically this is the name, that you have specified on the commandline, but without any path-components. So this `peek$` might return `foo.yab`. As a special case when yabasic has been

invoked without the name of a program to be executed this peek will return the literal strings `standard input` or, when also the option `-e` has been specified, `command line`. See also `peek$("program_file_name")` and `peek$("interpreter_path")` for related information.

`peek$("program_file_name")`

Returns the full file-name of the yabasic-program that is currently executing; typically this is the name, that you have specified on the commandline, including any path-components. For the special case, that you have bound your yabasic-program with the interpreter to a single standalone executable, this `peek$` will return its name. See also `peek$("program_name")` and `peek$("interpreter_path")` for related information.

`peek$("interpreter_path")`

Return the full file-name of the yabasic-interpreter that is currently executing your program; typically this will end on `yabasic` or `yabasic.exe` depending on your platform and the path will be where you installed yabasic. For bound programs (see creating a *standalone*-program) however, this may be different and will include whatever you specified during the `bind`-command.

See also `peek$("program_name")` and `peek$("program_file_name")` for related information. Employing these, it would be possible for a yabasic-program to start itself: `system(peek$("interpreter_path") + " " + peek$("program_file_name"))`. Of course, in this simple form this would be a bad idea, because this would start concurrent instances of yabasic without end.

`peek$("error")`

Return a string describing the nature of the last error in an `open`- or `seek`-statement. See the corresponding `peek("error")` for a detailed description.

`peek$("library")`

Return the name of the library, this statement is contained in. See the `import`-command for a detailed description or for more about libraries.

`peek$("version")`

Version of yabasic as a string; e.g. `2.77.1`. See also the related `peek("version")`, which returns nearly the same information (minus the patchlevel) as a number, e.g. `2.77`.

`peek$("os")`

This `peek` returns the name of the operating system, where your program executes. This can be either `windows` or `unix`.

peek$("font")

Return the name of the font, which is used for text within the graphic window; this value can be specified as the third argument to the open window-command.

peek$("env","*NAME*")

Return the environment variable specified by *NAME* (which may be any string expression). Which kind of environment variables are available on your system depends, as well as their meaning, on your system; however typing **env** on the command line will produce a list (for Windows and Unix alike). Note, that peek$("env",...) can be written as peek$("environment",...) too.

peek$("argument")

Return one of the arguments, that have been passed to yabasic at invocation time (the next call will return the the second argument, and so on). E.g. if yabasic has been called like this: yabasic foo.yab bar baz, then the first call to peek$("argument") will return bar. This is because foo.yab is treated as the name of the program to run, whereas bar and baz are considered arguments to this program, which are passed on the command line. The second call to peek$("argument") will return baz. *Note*, that for windows-users, who tend to click on the icon (as opposed to starting yabasic on the command line), this peekwill mostly return the empty string.

Note, that peek$("argument") can be written as peek$("arguments").

Finally you will want to check out the corresponding function peek("argument").

peek$("last_foreign_function_call_error_text")

Retrieve error text: If passing "no_error" to foreign_function_call, any error (e.g. failure to load the specified library), will *not* terminate your yabasic-program but rather store a descriptive error message away for later retrieval. Later on you may then check peek("last_foreign_function_call_okay") to find out, if something went wrong and retrieve a description with peek$("last_foreign_function_call_error_text").

This peek can be abbreviated as peek$("last_frnfn_call_error_text")

Example

```
print "You have supplied these arguments: "
while peek("argument")
  print peek("argument"),peek$("argument")
wend
```

If you save this program in a file foo.yab and execute it via **yabasic t.yab**

a b c (for windows users: please use the command line for this), your will get this output:

```
3a
2b
1c
```

See also

peek, poke, open

Name

pi — a constant with the value 3.14159

Synopsis

```
print pi
```

Description

pi is 3.14159265359 (well at least for yabasic); do not try to assign to pi (e.g. pi=22/7) this would not only be mathematically dubious, but would also result in a syntax error.

Example

```
for a=0 to 180
  print "The sine of ",a," degrees is ",sin(a*pi/180)
next a
```

This program uses pi to transform an angle from degrees into radians.

See also

euler

Name

poke — change selected internals of yabasic

Synopsis

```
poke "foo","bar"
poke "foo",baz
poke #a,"bar"
poke #a,baz
```

Description

The poke-command may be used to change details of yabasic's behaviour. Like the related function peek, poke does many different things, depending on the arguments supplied.

Here are the different things you can do with poke:

poke 5,a

Write the given byte (a in the example above) to the specified stream (5#a in the example).

See also the related function function peek(1).

poke "dump","filename.dump"

Dump the internal form of your basic-program to the named file; this is only useful for debugging the internals of yabasic itself.

The second argument ("filename.dump" in the example) should be the name of a file, that gets overwritten with the dump, please be careful.

poke "fontheight",12

This poke changes the default fontheight. This can only have an effect, if the fonts given in the commands text or open window do not specify a fontheight on their own.

poke "font","fontname"

This poke specifies the default font. This can only have an effect, if you do not supply a fontname with the commands text or open window.

poke "infolevel","debug"

Change the amount of internal information, that yabasic outputs during execution.

The second argument can be either "debug", "note", "warning", "error" or "fatal". However, normally you will not want to change this from its default value "warning".

See also the related peek$("infolevel").

poke "random_seed",42

Set the seed for the random number generator; if you do this, the ran-function will return the same sequence of numbers every time the program is started.

poke "stdout","some text"

Send the given text to standard output. Normally one would use

print for this purpose; however, sending e.g. control characters to your terminal is easier with this poke.

poke "textalign","cc"

This poke changes the *default* alignment of text with respect to the coordinates supplied within the text-command. However, this value does *not apply*, if the text-command explicitly specifies an alignment. The second argument ("cc" in the example) must always be two characters long; the first character can be one of l (*left*), r (*right*) or c (*center*); the second character can be either t (*top*), b (*bottom*) or c (*center*); see the corresponding peek$("textalign") for a detailed description of this argument.

poke "windoworigin","lt"

This poke moves the origin of the coordinate system of the window to the specified position. The second argument ("lt" in the example) must always be two characters long; the first character can be one of l (*left*), r (*right*) or c (*center*); the second character can be either t (*top*), b (*bottom*) or c (*center*). Together those two characters specify the new position of the coordinate-origin. See the corresponding peek$("windoworigin") for a more in depth description of this argument.

Example

```
print "Hello, now you will see, how much work"
print "a simple for-loop involves ..."
input "Please press return " a$
poke "infolevel","debug"
for a=1 to 10:next a
```

This example only demonstrates one of the many pokes, which are described above: The program switches the infolevel to debug, which makes yabasic produce a lot of debug-messages during the subsequent for-loop.

See also

peek, peek$

Name

print — Write to terminal or file

Synopsis

```
print "foo",a$,b
print "foo",a$,b;
print #a "foo",a$
```

```
print #a "foo",a$;
print foo using "##.###"
print reverse "foo"
print at(10,10) a$,b
print @(10,10) a$,b
print color("red","blue") a$,b
print color("magenta") a$,b
print color("green","yellow") at(5,5) a$,b
```

Description

The `print`-statement outputs strings or characters, either to your terminal (also known as *console*) or to an open file.

To understand all those uses of the `print`-statement, let's go through the various lines in the synopsis above:

`print "foo",a$,b`

> Print the string `foo` as well as the contents of the variables `a$` and `b` onto the screen, silently adding a *newline*.

`print "foo",a$,b;`

> (Note the trailing semicolon !) This statement does the same as the one above; only the implicit *newline* is skipped, which means that the next `print`-statement will append seamlessly.

`print #a "foo",a$`

> This is the way to write to files. The file with the number `a` must be open already, an implicit *newline* is added. Note the file-number `#a`, which starts with a hash ('`#`') amd is separated from the rest of the statement by a space only. The file-number (contained in the variable `a`) must have been returned by a previous `open`-statement (e.g. `a=open("bar")`).

`print #a "foo",a$;`

> The same as above, but without the implicit *newline*.

`print foo using "##.###"`

> Print the number `foo` with as many digits before and after the decimal dot as given by the number of '`#`'-signs. See the entries for `using` and `str$` for a detailed description of this format.

`print reverse "foo"`

> As all the `print`-variants to follow, this form of the `print`-statement can only be issued after `clear screen` has been called. The strings and numbers after the `reverse`-clause are simply printed inverse (compared to the normal `print`-statement).

`print at(10,10) a$,b`

> Print at the specified (x,y)-position. This is only allowed after

`clear screen` has been called. You may want to query `peek$("screenwidth")` or `peek$("screenheight")` to learn the actual size of your screen. You may add a semicolon to suppress the implicit newline.

`print @(10,10) a$,b`

This is exactly the same as above, however, `at` may be written as `@`.

`print color("red","blue") at(5,5) a$,b`

Print with the specified fore- (`"red"`) and background (`"blue"`) color (or colour). The possible values are `"black"`, `"white"`, `"red"`, `"blue"`, `"green"`, `"yellow"`, `"cyan"` or `"magenta"`. Again, you need to call `clear screen` first and add a semicolon if you want to suppress the implicit newline.

`print color("magenta") a$,b`

You may specify the foreground color only.

`print color("green","yellow") a$,b`

A color and a position (in this sequence, not the other way around) may be specified at once.

Example

```
clear screen
columns=peek("screenwidth")
lines=peek("screenheight")
dim col$(7)
for a=0 to 7:read col$(a):next a
data "black","white","red","blue","green","yellow","cyan","magenta"

for a=0 to 2*pi step 0.1
  print colour(col$(mod(i,8))) at(columns*(0.8*sin(a)+0.9)/2,lines*(0.8*cos(a)-
  i=i+1
next a
```

This example draws a colored ellipse within the text window.

See also

`at`, `print color`, `input`, `clear screen`, `using`, `;`

Name

print color — print with color

Synopsis

```
print color(fore$) text$
print color(fore$,back$) text$
```

Description

Not a separate command, but part of the `print`-command; may be included just after `print` and can only be issued after `clear screen` has been executed.

`color()` takes one or two string-arguments, specifying the color of the text and (optionally) the background.

The one or two strings passed to `color()` can be one of these: `"black"`, `"white"`, `"red"`, `"blue"`, `"green"`, `"yellow"`, `"cyan"` and `"magenta"` (which can be abbreviated as `"bla"`, `"whi"`, `"red"`, `"blu"`, `"gre"`, `"yel"`, `"cya"` and `"mag"` respectively).

`color()` can only be used, if `clear scren` has been issued at least once.

Note, that `color()` can be written as `colour()` too.

Example

```
clear screen
dim col$(7):for a=0 to 7:read col$(a):next a
do
  print color(col$(ran(7)),col$(ran(7))) " Hallo ";
  pause 0.01
loop
data "black","white","red","blue"
data "green","yellow","cyan","magenta"
```

This prints the word `" Hallo "` in all colors across your screen.

See also

`print`, `clear screen`, `at`

Name

print colour — see `print color`

Synopsis

```
print colour(fore$) text$
print colour(fore$,back$) text$
```

See also

`color`

Name

putbit — draw a rectangle of pixels encoded within a string into the graphics window

Synopsis

```
open window 200,200
…
a$=getbit(20,20,50,50)
…
putbit a$,30,30
putbit a$ to 30,30
putbit a$,30,30,"or"
```

Description

The `putbit`-command is the counterpart of the `getbit$`-function. `putbit` requires a string as returned by the `getbit`-function. Such a string contains a rectangle from the graphic window; the `putbit`-function puts such a rectangular region back into the graphic-window.

Note, that the `putbit`-command currently accepts a fourth argument. However only the string value `"or"` is supported here. The effect is, that only those pixel, which are set in the string will be set in the graphic window. Those pixels, which are not set in the string, will not change in the window (as opposed to being cleared).

Example

```
repeat
  read d$
  c$ = c$ + d$
until( d$ = "")

open window 200,200

do
  x=ran(220)-10
  y=ran(220)-10
  putbit c$,x,y,"transparent"
loop

data "rgb 21,21:000000000000000000000000000000000000000000000000000000000000000000000
data "32c8000000000000000000000000000000000000000000000000000000000000000000000000000
data "00000000000000000000000000032c80032c80032c80032c80032c80032c80032c800
data "32c80032c8000000000000000000000000000000000000000000000000000000000000000000000
data "32c80032c80032c80032c80032c80032c80032c80032c80032c80032c80032c800
data "00000000000000000000000000000000000000000000000032c80032c80032c80032c800
data "32c80032c80032c80032c80032c80032c80032c80032c80032c80032c800000000
data "00000000000000000000000032c80032c80032c80032c80032c80032c80032c8c8
data "ff000032c80032c80032c80032c80032c80032c80000000000000000000000000000
data "00000032c80032c80032c80032c80032c8c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8
data "ff000032c80032c80032c80032c80032c80000000000000000000032c80032c80032c800
data "32c80032c8c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff000032c800
data "32c80032c80032c80032c80000000000000032c80032c80032c80032c8c8ff00c8ff00c8
data "ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff000032c80032c80032c800
data "32c80000000000000032c80032c80032c80032c8c8ff00c8ff00c8ff00c8ff00c8ff00c8
data "ff00c8ff00c8ff00c8ff00c8ff00c8ff000032c80032c80032c80032c800000000000000
data "32c80032c80032c80032c8c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8
```

```
data "ff00c8ff00c8ff000032c80032c80032c80032c80000000032c80032c80032c80032c8c8"
data "ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff0000"
data "32c80032c80032c80032c80000000000000032c80032c80032c80032c8c8ff00c8ff00c8"
data "ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff000032c80032c80032c800"
data "32c80000000000000032c80032c80032c80032c8c8ff00c8ff00c8ff00c8ff00c8ff00c8"
data "ff00c8ff00c8ff00c8ff00c8ff00c8ff000032c80032c80032c80032c800000000000000"
data "32c80032c80032c80032c8c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8"
data "ff00c8ff00c8ff000032c80032c80032c80032c80000000000000032c80032c80032c800"
data "32c80032c8c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff000032c800"
data "32c80032c80032c80032c8000000000000000000000032c80032c80032c80032c80032c8c8"
data "ff00c8ff00c8ff00c8ff00c8ff00c8ff00c8ff000032c80032c80032c80032c80032c800"
data "00000000000000000000000032c80032c80032c80032c80032c80032c80032c80032c800"
data "32c80032c80032c80032c80032c80032c80032c800000000000000000000000000000000"
data "00000000000032c80032c80032c80032c80032c80032c80032c80032c80032c80032c800"
data "32c80032c80032c80032c80032c8000000000000000000000000000000000000000000000"
data "32c80032c80032c80032c80032c80032c80032c80032c80032c80032c80032c80032c800"
data "0000000000000000000000000000000000000000000000000000000000000032c800"
data "32c80032c80032c80032c80032c80032c80032c800000000000000000000000000000000"
data "00000000000000000000000000000000000000000000000000000000000000000000000"
data "00000000000000000000000000000000000000000000000000000000000000000000000"
data ""
```

This program uses a precanned string (containing the image of a blue circle with a yellow centre) and draws it repeatedly into the graphic-window. The mode `"transparent"` ensures, that no pixels will be cleared.

There are two possible values for the third argument of `putbit`. Both modes differ in the way, they replace (or not) any pixels from the window with pixels from the bitmap having the background colour.

`transparent` or `t`

> With this mode the pixels from the window will be kept, if the bitmap contains pixels with background colour at this position; i.e. the bitmap is *transparent*

`solid` or `s`

> With this mode the pixels from the window will be overpainted with the pixels from the bitmap in any case; i.e. the bitmap is *solid*

If you omit this argument, the default `transparent` applies.

See also

`getbit$`, `open window`

Name

putscreen — draw a rectangle of characters into the text terminal

Synopsis

```
clear screen
…
```

```
a$=getscreen$(5,5,10,10)
…
putscreen a$,7,7
```

Description

The putscreen-command is the counterpart of the getscreen$-function. putscreen requires a string as returned by the getscreen-function. Such a string contains a rectangular detail from the terminal; the putscreen-function puts such a region back into the terminal-window.

Note, that clear screen must have been called before.

Example

```
clear screen
for a=1 to 200
  print color("red") "Hallo !";
  print color("blue") "Welt !";
next a
r$=getscreen$(0,0,20,20)
for x=0 to 60
  putscreen r$,x,0
  sleep 0.1
next x
```

This example prints the string "Hallo !Welt !" all over the screen and then moves a rectangle from one side to the other.

See also

getscreen$, clear screen

R

Name

ran() — return a random number

Synopsis

```
print ran()
x=ran(y)
```

Description

The ran-function returns a random number. If no argument is given, the number returned is in the range from 0 to 1; where only 0 is a possible value; 1 will never be returned. If an argument is supplied, the number returned will be in the range from 0 up to this argument, whereas this argument itself is not a possible return value. Regardless of the range, ran is guaranteed to have exactly 2**30

different return values.

If you call `ran` multiple times during your program, the sequence of random numbers will be different each time you invoke your program; however, if, e.g. for testing you prefer to always have the same sequence of random numbers you may issue `poke "random_seed",123`.

Example

```
clear screen
c=peek("screenwidth")-1
l=peek("screenheight")

dim col$(8)
for a=0 to 7:read col$(a):next a
data "black","white","red","blue","green","yellow","cyan","magenta"

do
  x=ran(c)
  y=l-ran(l*exp(-32*((x/c-1/2)**2)))
  i=i+1
  print color(col$(mod(i,8))) at(x,y) "*";
loop
```

This example will print a colored bell-curve.

See also

`int`

Name

read — read data from `data`-statements

Synopsis

```
read a$,a
…
data "Hello !",7
```

Description

The `read`-statement retrieves literal data, which is stored within `data`-statements elsewhere in your program.

Example

```
read num
dim col$(num)
for a=1 to num:read col$(a):next a
clear screen
print "These are the colours known to yabasic:\n"
for a=1 to num
```

```
  print colour(col$(a)) col$(a)
next a

data 8,"black","white","red","blue"
data "green","yellow","cyan","magenta"
```

This program prints the names of the colors known to yabasic in those very colors.

See also

data, restore

Name

rectangle — draw a rectangle

Synopsis

```
open window 100,100
rectangle 10,10 to 90,90
rectangle 20,20,80,80
rect 20,20,80,80
box 30,30,70,70
clear rectangle 30,30,70,70
fill rectangle 40,40,60,60
clear fill rectangle 60,60,40,40
```

Description

The `rectangle`-command (also known as `box` or `rect`, for short) draws a rectangle; it accepts four parameters: The x- and y-coordinates of two facing corners of the rectangle. With the optional clauses `clear` and `fill` (which may appear together and in any sequence) the rectangle can be cleared and filled respectively.

Example

```
open window 200,200
c=1
do
  for phi=0 to pi step 0.1
    if (c) then
      rectangle 100+100*sin(phi),100+100*cos(phi) to 100-100*sin(phi),100-100*c
    else
      clear rectangle 100+100*sin(phi),100+100*cos(phi) to 100-100*sin(phi),100
    endif
    sleep 0.1
  next phi
  c=not c
loop
```

This example draws a nice animated pattern; watch it for a couple of hours, to see how it develops.

See also

open window, open printer, line, circle, triangle

Name

redim — create an array prior to its first use. A synonym for dim

Synopsis

See the dim-command.

Description

The redim-command does exactly the same as the dim-command; it is just a *synonym*. redim has been around in older versions of basic (not even yabasic) for many years; therefore it is supported in yabasic for compatibility reasons.

Please refer to the entry for the dim-command for further information.

Name

rem — start a comment

Synopsis

```
rem  Hey, this is a comment
#    the hash-sign too (at beginning of line)
// even the double slash
' and the single quote (at beginning of line)
print "Not a comment" #    This is an error !!
print "Not a comment"://   But this is again a valid comment
print "Not a comment" //   even this.
print "Not a comment" rem  and this !
```

Description

rem introduces a comment (like # or //), that extends up to the end of the line.

Those comments do not even need a colon (':') in front of them; they (rem, #, ' (single quite) and //) all behave alike except for # and ', which may only appear at the very beginning of a line; therefore the fourth example in the synopsis above (print "Not a comment" # This is an error !!) is indeed an error.

Note, that rem is an abbreviation for *remark*. remark however is *not* a valid command in yabasic.

Finally note, that a comment introduced with '#' may have a special meaning under unix; see the entry for # for details.

Example

```
#
rem   comments on data structures
#     are more useful than
//    comments on algorithms.
rem
```

This program does nothing, but in a splendid and well commented way.

See also

#, //

Name

repeat — start a repeat-loop

Synopsis

```
repeat
  …
until …
```

Description

The repeat-loop executes all the statements up to the final until-keyword over and over. The loop is executed as long as the condition, which is specified with the until-clause, becomes true. By construction, the statements within the loop are executed at least once.

Example

```
x=0
clear screen
print "This program will print the numbers from 1 to 10"
repeat
  x=x+1
  print x
  print "Press any key for the next number, or 'q' to quit"
  if (inkey$="q") break
until x=10
```

This program is pretty much useless, but self-explanatory.

See also

until, break, while, do

Name

restore — reposition the data-pointer

Synopsis

```
read a,b,c,d,e,f
restore
read g,h,i
restore foo
data 1,2,3
label foo
data 4,5,6
```

Description

The restore-command may be used to *reset* the reading of data-statements, so that the next read-statement will read data from the first data-statement.

You may specify a label with the restore-command; in that case, the next read-statement will read data starting at the given label. If the label is omitted, reading data will begin with the first data-statement within your program.

Example

```
input "Which language (german/english) ? " l$
if (instr("german",l$)>0) then
  restore german
else
  restore english
endif

for a=1 to 3
  read x,x$
  print x,"=",x$
next a

label english
data 1,"one",2,"two",3,"three"
label german
data 1,"eins",2,"zwei",3,"drei"
```

This program asks to select one of those languages known to me (i.e. english or german) and then prints the numbers 1,2 and 3 and their textual equivalents in the chosen language.

See also

read, data, label

Name

return — return from a subroutine or a gosub

Synopsis

```
gosub foo
…
label foo
…
return

sub bar(baz)
  …
  return quertz
end sub
```

Description

The return-statement serves two different (albeit somewhat related) purposes. The probably more important use of return is to return control from within a subroutine to the place in your program, where the subroutine has been called. If the subroutine is declared to return a value, the return-statement might be accompanied by a string or number, which constitutes the *return value* of the subroutine.

However, even if the subroutine should return a value, the return-statement need not carry a value; in that case the subroutine will return 0 or the empty string (depending on the type of the subroutine). Moreover, feel free to place multiple return-statements within your subroutine; it's a nice way of controlling the flow of execution.

The second (but historically first) use of return is to return to the position, where a prior gosub has left off. In that case return may *not* carry a value.

Example

```
do
  read a$
  if (a$="") then
    print
    end
  endif
  print mark$(a$)," ";
loop

data "The","quick","brown","fox","jumped"
data "over","the","lazy","dog",""

sub mark$(a$)
  if (instr(lower$(a$),"q")) return upper$(a$)
  return a$
```

```
end sub
```

This example features a subroutine `mark$`, that returns its argument in upper case, if it contains the letter "q", or unchanged otherwise. In the test-text the word `quick` will end up being marked as `QUICK`.

The example above demonstrates `return` within subroutines; please see `gosub` for an example of how to use `return` in this context.

See also

`sub`, `gosub`

Name

reverse — print reverse (background and foreground colors exchanged)

Synopsis

```
clear screen
…
print reverse "foo"
```

Description

`reverse` may be used to `print` text in reverse. `reverse` is not a separate command, but part of the `print`-command; it may be included just after the `print` and can only be issued once that `clear screen` has been issued.

Example

```
clear screen

print "1 ";
c=3
do
  prim=true
  for a=2 to sqrt(c)
    if (frac(c/a)=0) then
      prim=false
      break
    endif
  next a
  if (prim) then
    print
    print reverse c;
  else
    print c;
  endif
  print " ";
  c=c+1
loop
```

This program prints numbers from 1 on and marks each prime number in reverse.

See also

at, print color, print, clear screen

Name

right$() — return (*or change*) the right end of a string

Synopsis

```
print right$(a$,2)
right$(b$,2)="baz"
```

Description

The `right$`-function requires two arguments (a string and a number) and returns the part from the right end of the string, whose length is specified by its second argument. So, `right$` simply returns the requested number of chars from the right end of the given string.

Note, that the `right$`-function can be assigned to, i.e. it may appear on the left hand side of an assignment. In this way it is possible to change a part of the variable used within the `right$`-function. Note, that that way the *length* of the string cannot be changed, i.e. characters might be overwritten, but not added. For an example see below.

Example

```
print "Please enter a length either in inch or centimeter"
print "please add 'in' or 'cm' to mark the unit."
input "Length: " a$
if (right$(a$,2)="in") then
   length=val(a$)*2.56
elsif (right$(a$,2)="cm") then
   length=val(a$)
else
   error "Invalid input: "+a$
endif
```

This program allows the user to enter a length qualified with a unit (either inch or centimeter).

This second example demonstrates the capability to *assign* to the `right$`-function.

```
a$="Heiho World !"
print a$
right$(a$,7)="dwarfs."
```

```
print a$
```

See also

right$ and mid$

Name

rinstr() — find the rightmost occurrence of one string within the other

Synopsis

```
pos=rinstr("Thequickbrownfox","equi")
pos=rinstr(a$,b$,x)
```

Description

The rinstr-function accepts two string-arguments and tries to find the second within the first. However, unlike the instr, the rinstr-function finds the *rightmost* (or last) occurrence of the string; whereas the instr-function finds the *leftmost* (or first) occurrence. In any case however, the position is counted from the left.

If you supply a third, numeric argument to the rinstr-function, it will be used as a starting point for the search. Therefore rinstr("abcdeabcdeabcde","e",8) will return 5, because the search for an "e" starts at position 8 and finds the first one at position 5.

Example

```
print rinstr("foofoofoobar","foo")
```

This simple example will print 7, because it finds the *rightmost* among the three occurrences of foo within the string. Note, that

```
print instr("foofoofoobar","foo")
```

would have printed 1.

See also

instr

Name

round() — round its argument to the nearest integer

Synopsis

```
print round(x)
```

Description

The `found`-function returns the nearest integer (e.g. `3.0` for an argument of `2.6`). An argument with a fractional part of `0.5` (e.g. `2.5`) represents an edge case, as such an argument has the same distance to two numbers, `3.0` and `2.0` in the example; this ambiguity is resolved by rounding away from zero and returning `3.0`. By the same rule `round(-2.5)` returns `-3.0`; so you see, that `round(x)` always equals `round(-x)`.

Example

```
print round(2.3), round(2.5), round(2.7)
print round(2), round(-2)
print int(-2.3),round(-2.5), round(-2.7)
```

These examples return in order `2 3 3`, then `2 -2` and finally `2 -3 -3`.

See also

ceil, floor, int, frac

Name

rtrim$() — trim spaces at the right end of a string

Synopsis

```
a$=rtrim$(b$)
```

Description

The `rtrim$`-function removes all whitespace from the right end of a string and returns the result.

Example

```
open 1,"foo"
dim lines$(100)
l=1
while not eof(1)
  input #1 a$
  a$=rtrim$(a$)
  if (right$(line$,1)="\\") then
    line$=line$+" "+a$
  else
    lines$(l)=line$
```

```
    l=l+1
    line$=a$
  endif
end while
print "Read ",l," lines"
```

This example reads the file `foo` allowing for *continuation lines*, which are marked by a `\`, which appears as the last character on a line. For convenience whitespace at the right end of a line is trimmed with `rtrim`.

See also

`ltrim$`, `trim$`

S

Name

screen — as `clear screen` clears the text window

Synopsis

```
clear screen
```

Description

The keyword `screen` appears only within the sequence `clear screen`; please see there for a description.

See also

`clear screen`

Name

seek() — change the position within an open file

Synopsis

```
open 1,"foo"
seek #1,q
seek #1,x,"begin"
seek #1,y,"end"
seek #1,z,"here"
```

Description

The `seek`-command changes the position, where the next `input` (or `peek`)

statement will read from an open file. Usually files are read from the beginning to the end sequentially; however sometimes you may want to depart from this simple scheme. This can be done with the `seek`-command, allowing you to change the position, where the next piece of data will be read from the file.

`seek` accepts two or three arguments: The first one is the number of an already open file. The second one is the position where the next read from the file will start. The third argument is optional and specifies the the point from where the position (the second argument) will count. It can be one of:

`begin`

Count from the beginning of the file.

`end`

Count from the end of the file.

`here`

Count from the current position within the file.

Example

```
open #1,"count.dat","w"
for a=1 to 10
  print #1,"00000000";
  if (a<10) print #1,";";
next a

dim count(10)
do
  x=int(ran(10))
  i=i+1
  if (mod(i,1000)=0) print ".";
  count(x)=count(x)+1
  curr$=right$("00000000"+str$(count(x)),8)
  seek #1,9*x,"begin"
  print #1,curr$;
loop
```

This example increments randomly one of ten counters (in the array `count()`); however, the result is always kept and updated within the file `count.dat`, so even in case of an unexpected interrupt, the result will not be lost.

See also

tell, open, print, peek

Name

sig() — return the sign of its argument

Synopsis

```
a=sig(b)
```

Description

Return `+1`, `-1` or `0`, if the single argument is positive, negative or zero.

Example

```
clear screen
dim c$(3):c$(1)="red":c$(2)="white":c$(3)="green"
do
  num=ran(100)-50
  print color(c$(2+sig(num))) num
loop
```

This program prints an infinite sequence of random number; positive numbers are printed in green, negative numbers are printed red (an exact zero would be printed white). (With a little extra work, this program could be easily extended into a brokerage system)

See also

`abs`, `int`, `frac`

Name

sin() — return the sine of its single argument

Synopsis

```
y=sin(angle)
```

Description

The `sin`-function expects an angle (in radians, *not* degrees) and returns its sine.

Example

```
open window 200,200
new curve
for phi=0 to 2*pi step 0.1
  line to 100+90*sin(phi),100+90*cos(phi)
next phi
close curve
```

This program draws a circle (ignoring the existence of the `circle`-command).

See also

`asin`, `cos`

Name

shl() — shift its argument bitwise to the left

Synopsis

```
print shl(0b11001,8)
```

Description

The `shl`-function (`shl` stands for *shift left*) treats its first argument as a binary number and shifts it to the left as specified by its second argument, filling up the gaps with zeroes. So `bin$(shl(0b11011,4))` returns `110110000` (the example uses `bin$` and a number with base 2).

Please note: as the argument of the function is converted to a 32-bit integer, all results are also confined to this range.

Example

```
print "Some powers of two:"
for i=0 to 5
  print shl(1,i)
next i
```

This will print the powers of two from 1 to 32, because the left-shift operation is equivalent to a multiplication with two.

See also

`and`, `or`, `eor`, `bitnot`, `shr`

Name

shr() — shift its argument bitwise to the right

Synopsis

```
print shr(0b110010000,4)
```

Description

The `shr`-function (`shr` stands for *shift right*) treats its first argument as a binary number and shifts it to the right as specified by its second argument; the rightmost binary digits are discarded during this operation. So `bin$(shr(0b1101100,2))` returns `11011` (the example uses `bin$` and a number with base 2).

Please note: as the argument of the function is converted to a 32-bit integer, all results are also confined to this range.

Example

```
print "Some powers of two:"
for i=0 to 5
  print shr(32,i)
next i
```

This will print the powers of two from 32 downto 1, because the right-shift operation is equivalent to a division by two (discarding any fractional part).

See also

`and`, `or`, `eor`, `bitnot`, `shl`

Name

sleep — pause, sleep, wait for the specified number of seconds

Synopsis

```
sleep 4
```

Description

The `sleep`-command has many different names: You may write `pause`, `sleep` or `wait` interchangeably; whatever you write, yabasic will always do exactly the same.

Therefore you should refer to the entry for the `pause`-function for further information.

Name

split() — split a string into many strings

Synopsis

```
dim w$(10)
…
num=split(a$,w$())
num=split(a$,w$(),s$)
```

Description

The `split`-function requires a string (containing the text to be split), a reference to a string-array (which will receive the resulting strings, i.e. the *tokens*) and an optional string (with a set of characters, at which to split, i.e. the *delimiters*).

The `split`-function regards its first argument (a string) as a list of *tokens* separated by *delimiters* and it will store the list of tokens within the array-reference you have supplied. Note, that the array, which is passed as a reference (`w$()` in the synopsis), will be resized accordingly, so that you don't have to figure out the number of tokens in advance. The element at position zero (i.e. `w$(0)`) will not be used.

normally (i.e. if you omit the third, which is the delimiter-argument) the function will regard *space* or *tab* as delimiters for tokens; however by supplying a third argument, you may split at *any single* of the characters within this string. E.g. if you supply `":;"` as the third argument, then colon (`:`) or semicolon (`;`) will delimit tokens.

Note, that a sequence of separator-characters will produce a sequence of empty tokens; that way, the number of tokens returned will always be one plus the number of separator characters contained within the string. Refer to the closely related `token`-function, if you do not like this behaviour. In some way, the `split`-function focuses on the separators (other than the `token`-function, which focuses on the *tokens*), hence its name.

The second argument is a reference on a string-array, where the tokens will be stored; this array will be expanded (or shrunk) to have room for all tokens, if necessary.

The first argument finally contains the text, that will be split into tokens. The `split`-function returns the number of tokens that have been found.

Please see the examples below for some hints on the exact behaviour of the `split`-function and how it differs from the `token`-function:

Example

```
print "This program will help you to understand, how the"
```

```
print "split()-function exactly works and how it behaves"
print "in certain special cases."
print
print "Please enter a line containing tokens separated"
print "by either '=' or '-'"
dim t$(10)
do
  print
  input "Please enter a line: " l$
  num=split(l$,t$(),"=-")
  print num," Tokens: ";
  for a=1 to num
    if (t$(a)="") then
      print "(EMPTY)";
    else
      print t$(a);
    endif
    if (a<num) print ",";
  next a
  print
loop
```

This program prints the following output:

```
Please enter a line: a
1 Tokens: a

Please enter a line:
0 Tokens:

Please enter a line: ab
1 Tokens: ab

Please enter a line: a=b
2 Tokens: a,b

Please enter a line: a-
2 Tokens: a,(EMPTY)

Please enter a line: a-=
3 Tokens: a,(EMPTY),(EMPTY)

Please enter a line: =a-
3 Tokens: (EMPTY),a,(EMPTY)

Please enter a line: a=-b
3 Tokens: a,(EMPTY),b

Please enter a line: a--b-
4 Tokens: a,(EMPTY),b,(EMPTY)

Please enter a line: -a==b-c==
7 Tokens: (EMPTY),a,(EMPTY),b,c,(EMPTY),(EMPTY)
```

See also

token

Name

sqr() — compute the square of its argument

Synopsis

```
a=sqr(b)
```

Description

The `sqr`-function computes the square of its numerical argument (i.e. it multiplies its argument with itself).

Please note, that other dialects of basic use `sqr` as the *square root*, rather than the *square*; this needs to be checked especially when porting programs from other Versions of basic.

Example

```
for a=1 to 10
  print a,sqr(a),a**2
next a
```

As you may see from the output, `sqr` can be written as `**2` (or `^2`) too.

See also

sqrt, **, ^

Name

sqrt() — compute the square root of its argument

Synopsis

```
to be written
```

Description

The `sqrt`-function computes the square root of its numerical argument.

Example

```
for a=1 to 5
  print a,sqrt(a),a**(1/2)
next a
```

As you may see from the output, `sqrt` can be written as `**(1/2)` (or `^(1/2)`) too.

See also

sqr, **, ^

Name

static — preserves the value of a variable between calls to a subroutine

Synopsis

```
sub foo()

  static a

  …

end sub
```

Description

The `static` keyword can be used within subroutines to mark variables as *static*. This has two effects: First, the variable is *local* to the subroutine, i.e. its value is not know outside the subroutine (this is the effect of the `local` keyword). Second, the `static`-keyword arranges things, so that the variable keeps its value between invocations of the subroutine (this is different from the `local`-keyword).

Example

```
foo()
foo()
foo()

sub foo()
  static a
  local b
  a=a+1
  b=b+1
  print a,b
end sub
```

This program shows the difference between `static` and `local` variables within a subroutine; it produces this output:

```
 1 1
 2 1
 3 1
```

The output shows, that the `static` variable `a` keeps its value between subroutine calls, whereas `b` is initialized with the value 0 at every call to the subroutine `foo`.

See also

sub, local

Name

step — specifies the increment step in a for-loop

Synopsis

```
for a=1 to 10 step 3
  …
next a
```

Description

Specify, by which amount the loop-variable of a for-loop will be incremented at each step.

The step (as well as the lower and upper bound) are computed anew in each step; this is not common, but possible, as the example below demonstrates.

Example

```
for x=1 to 1000 step y
  y=x+y
  print x," ",y," ";
next x
print
```

This program computes the fibonacci numbers between 1 and 1000.

See also

for

Name

str$() — convert a number into a string

Synopsis

```
a$=str$(a)
b$=str$(x,"##.###")
b$=str$(x,"###,###.##","_.")
c$=str$(x,"%g")
```

Description

The str$-function accepts a numeric argument and returns it as a

string.

Note: As a special and trivial case `str$` also accepts a single string-argument, which it just returns unchanged; this can be useful e.g. when used with `eval$`, see the example there for an application.

For the common case of converting a number to a string, the process can be controlled with an optional third argument (the *format* argument). See the following table of examples to learn about valid values of this argument. Note, that those examples fall in one of two categories: *C-style* and *basic-style*; the first 4 examples in the table below are C-style, the rest of the examples are basic-style. For more information on the C-style formats, you may refer to your favorite documentation on the C programming language. The basic-style formats are much simpler, they just depict the desired output, marking digits with '#'; groups of (usually three) digits may be separated with colons (','), the decimal dot must be marked by a literal dot ('.'). Moreover these characters (colons and dot) may be replaced by other characters to satisfy the needs of non-english (e.g. german) languages; see the examples below.

Note, that for clarity, each space in the result has been replaced by the letter 'x', because it would be hard to figure out, how many spaces are produced *exactly* otherwise.

Table 7.2. Examples for the format argument

Example string	Result for converting 1000*pi	Description
`%g`	`3141.59`	Internally yabasic uses double precision, so its numbers can be formatted with the `%g` format specifier.
`%2.5f`	`3141.59265`	The '`2`' determines the minimum length of the output; but if needed (as in the example) the output can be longer. The '`5`' is the number of digits after the decimal point.
`%12.5f`	`xx3141.59265`	Two spaces (which appear as '`x`') are added to pad the output to the requested length of 12 characters.
`%012.5g`	`0000003141.6`	The '`g`' requests, that the precision ('`5`') specifies the *overall* number of digits (before and after the decimal point).
`%-12.5f`	`3141.59265xx`	The '`-`' requests the output to be left-centered (therefore the filling space appears at the right).

Example string	Result for converting 1000*pi	Description
`#####.##`	`x3141.59`	Each '`#`' specifies a digit (either before or after the dot), the '`.`' specifies the position of the dot. As 1000*`pi` does not have enough digits, the 5 requested digits before the dot are filled up with a space (which shows up as an '`x`').
`##,###.##`	`x3,141.59`	Nearly the same as above, but the colon from the format shows up within the result.
`##,###.##` and an *additional* argument of `","`	`x3.141,59`	Similar to the example above, but colon and dot are replaced with dot and colon respectively.
`##,###.##` and an *additional* argument of `"_"`	`x3_141,59`	Similar to the example above, but colon and dot are replaced with underscore and colon respectively.
`#####`	`x3142`	The format string does not contain a dot, and therefore the result does not have any fractional digits.
`##.###`	`##.###`	As 1000*`pi` has 4 digits in front of the decimal dot and the format only specifies 2, yabasic does not know what to do; therefore it chooses just to reproduce the format string.

Example

```
do
  input "Please enter a format string: " f$
  a$=str$(1000*pi,f$)
  for a=1 to len(a$)
    if (mid$(a$,a,1)=" ") mid$(a$,a,1)="x"
  next a
  print a$
loop
```

This is the program, that has been used to get the results shown in the table above.

See also

`print`, `using`

Name

sub — declare a user defined subroutine

Synopsis

```
foo(2,"hello")

…

sub foo(bar,baz$)
  …
  return qux
  …
end sub
```

Description

The `sub`-keyword starts the definition of a *user defined subroutine*. With user defined subroutines you are able to somewhat extend yabasic with your own commands or functions. A subroutine accepts arguments (numbers or strings) and returns a number or a string (however, you are not required to assign the value returned to a variable).

The name of the subroutine follows after the keyword `sub`. If the name (in the synopsis: `foo`) ends on a '`$`', the subroutine should return a string (with the `return`-statement), otherwise a number.

After the name of the subroutine yabasic requires a pair of braces; within those braces you may specify a list of parameters, for which values can (but need not) be included when calling the subroutine. If you omit one of those parameters when calling such a subroutine, it assumes the value zero (for numeric parameters) or the empty string (for string-parameters). However from the special variable `numparams` you may find out, how many arguments have really been passed when calling the subroutine.

Parameters of a subroutine are always local variables (see the keyword `local` for more explanation).

From within the subroutine you may return any time with the keyword `return`; along with the `return`-keyword you may specify the return value. Note that more than one `return` is allowed within a single subroutine.

Finally, the keyword `end sub` ends the subroutine definition. Note, that the definition of a subroutine *need not* appear within the program *before* the first call to this sub.

> **Note**

As *braces* have two uses in yabasic (i.e. for supplying arguments to a subroutine as well as to list the indices of an array). yabasic can not tell apart an array from a subroutine with the same name. Therefore you *cannot* define a subroutine with the same name as an array !

Example

```
p=2
do
  if (is_prime(p)) print p
  p=p+1
loop

sub is_prime(a)
  local b
  for b=2 to sqrt(a)
    if (frac(a/b)=0) return false
  next b
  return true
end sub
```

This example is not the recommended way to compute prime numbers. However it gives a nice demonstration of using a subroutine.

See also

subroutines, `local`, `static`, `peek`

Name

switch — select one of many alternatives depending on a value

Synopsis

```
switch a
  case 1
  case 2
  …
end switch

…

switch a$
  case "a"
  case "b"
end switch
```

Description

The `switch`-statement selects one of many codepaths depending on a numerical or string expression. I.e. it takes an expression (either

numeric or string) and compares it with a series of values, each wrapped within a `case`-clause. If the expression equals the value given in a `case`-clause, the subsequent statements are executed.

The `default`-clause allows one to specify commands, which should be executed, if none of `case`-clauses matches.

Note, that many `case`-clauses might be clustered (e.g. `case "a":case "b":case "c"`). Or put another way: You need a `break`-statement at the end of a `case`-branch, if you do not want to run into the next `case`.

Example

```
input "Please enter a single digit: " n
switch n
  case 0:print "zero":break
  case 1:print "one":break
  case 2:print "two":break
  case 3:print "three":break
  case 4:print "four":break
  case 5:case 6: case 7:case 8:case 9
    print "Much !":break
  default:print "Hey ! That was more than a single digit !"
end switch
```

This example translates a single digit into a string; note, how the cases 5 to 7 are clustered.

See also

`switch`, `case`, `break`

Name

system() — hand the name of an external command over to your operating system and return its exitcode

Synopsis

```
ret=system("foo")
system("bar")
```

Description

The `system`-command accepts a single string argument, which specifies a command to be executed. The function will return the exitcode of the command; its output (if any) will be lost.

Example

```
print "Please enter the name of the file, that should be deleted."
```

```
input f$
if (system("rm "+f$+" >/dev/null 2>&1")) then
  print "Error !"
else
  print "okay."
endif
```

This program is Unix-specific: It uses the Unix-command `rm` to remove a file.

See also

system$

Name

system$() — hand the name of an external command over to your operating system and return its output

Synopsis

```
print system$("dir")
```

Description

The `system$`-command accepts a single string argument, specifying a command, that can be found and executed by your operating system. It returns the output of this command as one big string.

Example

```
input "Please enter the name of a directory: " d$
print
print "This is the contents of the '"+d$+"':"
print system$("dir "+d$)
```

This example lists the contents of a directory, employing the dir-command (which is about the only program, that is known under Unix as well as Windows).

See also

system, chomp

T

Name

tan() — return the tangent of its argument

Synopsis

```
foo=tan(bar)
```

Description

The `tan`-function computes the tangent of its arguments (which should be specified in radians).

Example

```
for a=0 to 45
  print tan(a*pi/180)
next a
```

This example simply prints the tangent of all angles between 0 and 45 degrees.

See also

atan, sin

Name

tell — get the current position within an open file

Synopsis

```
open #1,"foo"
  …
position=tell(#1)
```

Description

The `tell`-function requires the number of an open file as an argument. It returns the position (counted in bytes, starting from the beginning of the file) where the next read will start.

Example

```
open #1,"foo","w"
print #1 "Hello World !"
close #1

open #1,"foo"
seek #1,0,"end"
print tell(#1)
close 1
```

This example (mis)uses `tell` to get the size of the file. The `seek` positions the file pointer at the end of the file, therefore the call to `tell` returns the total length of the file.

See also

`tell`, `open`

Name

text — write text into your graphic-window

Synopsis

```
text x,y,"foo"
text x,y,"foo","lb"
text x,y,"foo","cc","font"
text x,y,"foo","font","rt"
```

Description

The `text`-commands displays a text-string (the third argument) at the given position (the first two arguments) within an already `open`ed `window`. The font to be used can be optionally specified as either the fourth or fifth argument (`"font"` in the example above). A font specified this way will also be used for any subsequent `text`-commands, as long as they do not specify a font themselves.

The fourth or fifth optional argument (`"lb"` in the example above) can be used to specify the alignment of the text with respect to the specified position. This argument is always two characters long: The first character specifies the horizontal alignment and can be either `l`, `r` or `c`, which stand for *left, right* or *center*. The second character specifies the vertical alignment and can be one of `t`, `b` or `c`, which stand for *top, bottom* or *center* respectively. If you omit this alignment argument, the default `"lb"` applies; however this default may be changed with `poke "textalign","xx"`

Example

```
open window 500,200
clear screen
data "lt","lc","lb","ct","cc","cb","rt","rc","rb"
for a=1 to 9
  read align$
  print "Alignment: ",align$
  line 50*a-15,100,50*a+15,100
  line 50*a,85,50*a,115
  text 50*a,100,"Test",align$
  inkey$
next a
```

This program draws nine crosses and writes the same text at each; however it goes through all possible nine alignment strings, showing their effect.

See also

open window, peek, poke

Name

then — tell the long from the short form of the if-statement

Synopsis

```
if (a<b) then
  …
endif
```

Description

The keyword then is part of the if-statement; please see there for further explanations. However, not every if-statement requires the keyword then: If the keyword then is present, the if-clause may extend over more than one line, and the keyword endif is required to end it. If the keyword then is *not* present, the if-statement extends up to the end of the line, and any endif would be an error.

Example

```
if (1<2) then
  print "Hello ";
endif

if (2<3) print "world"
if (2<1)
  print "!"
```

This example prints Hello world. Note, that no exclamation mark (!) is printed, which might come as a surprise and may be changed in future versions of yabasic.

See also

if

Name

time$ — return a string containing the current time

Synopsis

```
print time$
print time$()
```

Description

The `time$` function returns the current time in four fields separated by hyphens '`-`'. The fields are:

- The current hour in the range from 0 to 23, padded with zeroes (e.g. `00` or `04`) to a length of two characters.
- The number of minutes, padded with zeroes.
- The number of seconds, padded with zeroes.
- The number of seconds, that have elapsed since the program has been started. This value increases as long as your program runs and is therefore unbound and not padded with zeroes.

At the time of writing this documentation, `time$` returns `22-58-53-0`. Note, that the first three of the four fields returned by `time$` have a fixed width; therefore it is easy to extract some fields with the usual string-functions `mid$` (and others).

Example

```
print "Hello it is ",time$
print "An empty for-loop with ten million iterations takes ";
for a=1 to 10000000:next a
print "Now it is ",time$
print peek("secondsrunning")," seconds have passed."
```

This program benchmarks the `for`-loop; however, it does not use the fourth field of the string returned by `time$`, because that string wraps around every 60 seconds; rather the `peek` `"secondsrunning"` is queried.

See also

`date`

Name

to — this keyword appears as part of other statements

Synopsis

```
for a=1 to 100 step 2
  …
next a

line x,y to a,b
```

Description

The `to`-keyword serves two purposes (which are not related at all):

- within `for`-statements, to specify the upper bound of the loop.
- Within any graphical command (e.g. `line`), that requires two points (i.e. four numbers) as arguments, a comma ',' might be replaced with the keyword `to`. I.e. instead of `100,100,200,200` you may write `100,100 to 200,200` in such commands.

Example

Please see the command listed under "See also" for examples.

See also

`for`, `line`, `rectangle`

Name

token() — split a string into multiple strings

Synopsis

```
dim w$(10)
…
num=token(a$,w$())
num=token(a$,w$(),s$)
```

Description

The `token`-function accepts a string (containing the text to be split), a reference to a string-array (which will receive the resulting strings, i.e. the *tokens*) and an optional string (with a set of characters, at which to split, i.e. the *delimiters*).

The `token`-function regards its first argument as a list of *tokens* separated by *delimiters* and it will store the list of tokens within the array-reference that has been supplied. Note, that the array, which is passed as a reference (`w$()` in the synopsis), will be resized accordingly, so that you don't have to figure out the number of tokens in advance. The element at position zero (i.e. `w$(0)`) will not be used.

Normally (i.e. if you omit the third, the delimiter-argument) the function will regard *space* or *tab* as delimiters for tokens; however by

supplying a third argument, you may split at *any single* of the characters within this string. E.g. if you supply `":;"` as the third argument, then colon (`:`) or semicolon (`;`) will delimit tokens.

Note, that `token` will never produce empty tokens, even if two or more separators follow in sequence. Refer to the closely related `split`-function, if you do not like this behaviour. In some way, the `token`-function focuses on the tokens and not on the separators (other than the `split`-function, which focuses on the separators).

The second argument is a reference on a string-array, where the tokens will be stored; this array will be expanded (or shrunk) as necessary to have room for all tokens.

The first argument finally contains the text, that will be split into tokens. The `token`-function returns the number of tokens, that have been found.

Please see the examples below for some hints on the exact behaviour of the `token`-function and how it differs from the `split`-function:

Example

```
print "This program will help you to understand, how the"
print "token()-function exactly works and how it behaves"
print "in certain special cases."
print
print "Please enter a line containing tokens separated"
print "by either '=' or '-'"
dim t$(10)
do
  print
  input "Please enter a line: " l$
  num=token(l$,t$(),"=-")
  print num," Tokens: ";
  for a=1 to num
    if (t$(a)="") then
      print "(EMPTY)";
    else
      print t$(a);
    endif
    if (a<num) print ",";
  next a
  print
loop
```

This program prints the following output:

```
Please enter a line: a
1 Tokens: a

Please enter a line:
0 Tokens:

Please enter a line: ab
1 Tokens: ab

Please enter a line: a=b
2 Tokens: a,b
```

```
Please enter a line: a-
1 Tokens: a

Please enter a line: a-=
1 Tokens: a

Please enter a line: =a-
1 Tokens: a

Please enter a line: a=-b
2 Tokens: a,b

Please enter a line: a--b-
2 Tokens: a,b

Please enter a line: -a==b-c==
3 Tokens: a,b,c
```

See also

split

Name

triangle — draw a triangle

Synopsis

```
open window 100,100
triangle 100,100,50,50,100,50
fill triangle 50,100,100,50,200,200
clear fill triangle 20,20,10,10,200,200
```

Description

The `triangle`-command draws a triangle; it requires 6 parameters: The x- and y-coordinates of the three points making up the triangle. With the optional keywords `clear` and `fill` (which may appear both and in any sequence) the triangle can be cleared and filled respectively.

Example

```
open window 200,200
do
  phi=phi+0.2
  i=i+2
  color mod(i,255),mod(85+2*i,255),mod(170+3*i,255)
  dx=100*sin(phi):dy=20*cos(phi)
  fill triangle 100+20*sin(phi),100+20*cos(phi),100-20*sin(phi),100-20*cos(phi)
  sleep 0.1
loop
```

This example draws a colored triangles until you get exhausted.

See also

open window, open printer, line, circle, rectangle

Name

trim$() — remove leading and trailing spaces from its argument

Synopsis

```
a$=trim$(b$)
```

Description

The `trim$`-function removes all whitespace from the left and from the right end of a string and returns the result. Calling `trim$` is equivalent to calling `rtrim$(ltrim$())`.

Example

```
do
  input "Continue ? Please answer yes or no: " a$
  a$=lower$(trim$(a$))
  if (len(a$)>0 and a$=left$("no",len(a$)) exit
loop
```

This example asks for an answer (`yes` or `no`) and removes spaces with `trim$` to make the comparison with the string `"no"` more bulletproof.

See also

ltrim$, rtrim$

Name

true — a constant with the value of 1

Synopsis

```
okay=true
```

Description

The constant `true` can be assigned to variables which will later appear in conditions (e.g. an `if`-statement.

`true` may also be written as `TRUE` or even `TrUe`.

Example

```
input "Please enter a string of all upper letters: " a$
if (is_upper(a$)) print "Okay"

sub is_upper(a$)
  if (a$=upper$(a$)) return true
  return false
end sub
```

See also

false

U

Name

until — end a repeat-loop

Synopsis

```
repeat
  …
until …
```

Description

The until-keyword ends a loop, which has been introduced by the repeat-keyword. until requires an expression (see here for details) as an argument; the loop will continue *until* this condition evaluates to true.

Example

```
c=1
s=1
repeat
  l=c
  s=-(s+sig(s))
  c=c+1/s
  print c
until abs(l-c)<0.000001
```

This program calculates the sequence 1/1-1/2+1/3-1/4+1/5-1/6+1/7-1/8+ … ; please let me know, if you know against which value this converges.

See also

repeat

Name

upper$() — convert a string to upper case

Synopsis

```
u$=upper$(a$)
```

Description

The `upper$`-function accepts a single string argument and converts it to all upper case.

Example

```
line input "Please enter a sentence without the letter 'e': " l$
p=instr(upper$(l$),"E")
if (p) then
  l$=lower$(l$)
  mid$(l$,p,1)="E"
  print "Hey, you are wrong, see here!"
  print l$
else
  print "Thanks."
endif
```

This program asks for a sentence and marks the first (if any) occurrence of the letter 'e' by converting it to upper case (in contrast to the rest of the sentence, which is converted to lower case).

See also

`lower$`

Name

using — Specify the format for printing a number

Synopsis

```
print a using "##.###"
print a using("##.###",",.")
```

Description

The `using`-keyword may appear as part of the `print`-statement and specifies the format (e.g. the number of digits before and after the decimal dot), which should be used to print the number.

The possible values for the format argument (`"##.###"` in the synopsis

above) are described within the entry for the str$-function; especially the second line in the synopsis (`print a using("##.###",",.")`) will become clear after referring to `str$`. In fact the `using` clause is closely related to the `str$`-function; the former can always be rewritten using the latter; i.e. `print foo using bar$` is always equivalent to `print str$(foo,bar$)`. Therefore you should check out str$ to learn more.

Example

```
for a=1 to 10
  print sqrt(ran(10000*a)) using "#########.#####"
next a
```

This example prints a column of square roots of random number, nicely aligned at the decimal dot.

See also

print, str$

V

Name

val() — converts a string to a number

Synopsis

```
x=val(x$)
```

Description

The `val`-function checks, if the start of its string argument forms a floating point number and then returns this number. The string therefore has to start with digits (only whitespace in front is allowed), otherwise the `val`-function returns zero.

Example

```
input "Please enter a length, either in inches (in) or centimeters (cm) " l$
if (right$(l$,2)="in") then
  l=val(l$)*2.51
else
  l=val(l$)
print "You have entered ",l,"cm."
```

This example queries for a length and checks, if it has been specified in inches or centimeters. The length is then converted to centimeters.

See also

str$

W

Name

wait — pause, sleep, wait for the specified number of seconds

Synopsis

```
wait 4
```

Description

The wait-command has many different names: You may write pause, sleep or wait interchangeably; whatever you write, yabasic will always do exactly the same.

Therefore you should refer to the entry for the pause-function for further information.

Name

wend — end a while-loop

Synopsis

```
while a<b
  …
wend
```

Description

The wend-keyword marks the end of a while-loop. Please see the while-keyword for more details.

wend can be written as end while or even end-while.

Example

```
line input "Please enter a sentence: " a$
p=instr(a$,"e")
while p
  mid$(a$,p,1)="E"
  p=instr(a$,"e")
wend
print a$
```

This example reads a sentence and converts every occurrence of the letter `e` into uppercase (`E`).

See also

`while` (which is just the following entry).

Name

while — start a `while`-loop

Synopsis

```
while …
  …
wend
```

Description

The `while`-keyword starts a `while`-loop, i.e. a loop that is executed as long as the condition (which is specified after the keyword `while`) evaluates to `true`.

Note, that the body of such a `while`-loop will not be executed at all, if the condition following the `while`-keyword is not true initially.

If you want to leave the loop prematurely, you may use the `break`-statement.

Example

```
open #1,"foo"
while !eof(1)
  line input #1 a$
  print a$
wend
```

This program reads the file `foo` and prints it line by line.

See also

`until`, `break`, `wend`, `do`

Name

origin — move the origin of a window

Synopsis

```
open window 200,200
origin "cc"
```

Description

The `origin`-command applies to graphic windows and moves the origin of the coordinate system to one of nine point within the window. The normal position of the origin is in the upper left corner of the window; however in some cases this is inconvenient and moving the origin may save you from subtracting a constant offset from all of your coordinates.

However, you may not move the origin to an arbitrary position; in horizontal position there are only three positions: left, center and right, which are decoded by the letters `l`, `c` and `r`. In vertical position the allowed positions are top, center and bottom; encoded by the letters `t`, `c` and `b`. Taking the letters together, you arrive at a string, which might be passed as an argument to the command; e.g. `"cc"` or `"rt"`.

Example

100,100

```
open window 200,200
window origin "cc"
circle 0,0,60
```

This example draws a circle, centered at the center of the window.

See also

`open window`

X

Name

xor() — compute the exclusive or

Synopsis

```
x=xor(a,b)
```

Description

The `xor`-function computes the bitwise *exclusive or* of its two numeric arguments. To understand the result, both arguments should be viewed as binary numbers (i.e. a sequence of digits 0 and 1); a bit of the result will then be 1, if exactly one argument has a 1 and the

other has a 0 at this position in their binary representation.

Note, that both arguments are silently converted to integer values and that negative numbers have their own binary representation and may lead to unexpected results when passed to `and`.

Example

```
print xor(7,4)
```

This will print `3`. This result is obvious, if you note, that the binary representation of 7 and 4 are 111 and 100 respectively; this will yield 011 in binary representation or 2 as decimal.

The `eor`-function is the same as the `xor`-function; both are synonymous; however they have each their own description, so you may check out the entry of `eor` for a slightly different view.

See also

`and`, `or`, `eor`, `bitnot`

Symbols and Special characters

Name

— either a comment or a marker for a file-number

Synopsis

```
# This is a comment, but the line below not !
open #1,"foo"
```

Description

The hash ('`#`') has two totally unrelated uses:

- A hash might appear in commands related with file-io. yabasic uses simple numbers to refer to open files (within `input`, `print`, `peek` or `eof`). In those commands the hash may precede the number, which species the file. Please see those commands for further information and examples; the rest of *this* entry is about the second use (as a comment).
- As the *very first* character within a line, a hash introduces comments (similar to `rem`).

'`#`' as a comment is common in most scripting languages and has a special use under Unix: If the *very first line* of any Unix-program begins with the character sequence '`#!`' ("she-bang", no spaces

allowed), the rest of the line is taken as the program that should be used to execute the script. I.e. if your yabasic-program starts with '`#!/usr/local/bin/yabasic`', the program `/usr/local/bin/yabasic` will be invoked to execute the rest of the program. As a remark for windows-users: This mechanism ensures, that yabasic will be invoked to execute your program; the ending of the file (e.g. `.yab`) will be ignored by Unix.

Example

```
# This line is a valid comment
print "Hello " : # But this is a syntax error, because
print "World!" : # the hash is not the first character !
```

Note, that this example will produce a syntax error and is *not* a valid program !

See also

input, print, peek or eof, //, rem

Name

// — starts a comment

Synopsis

```
//  This is a comment !
```

Description

The double-slash ('`//`') is (besides `REM` and '`#`') the third way to start a comment. '`//`' is the latest and greatest in the field of commenting and allows yabasic to catch up with such cool languages like C++ and Java.

Example

```
// Another comment.
print "Hello world !" // Another comment
```

Unlike the example given for '#' this example is syntactically correct and will not produce an error.

See also

#, rem

Name

@ — synonymous to at

Synopsis

```
clear screen
…
print @(a,b)
```

Description

As '@' is simply a synonym for at, please see at for further information.

See also

at

Name

: — separate commands from each other

Synopsis

```
print "Hello ":print "World"
```

Description

The *colon* (':') separates multiple commands on a single line.

The *colon* and the *newline*-character have mostly the same effect, only that the latter, well, starts a new line too. The only other difference is their effect within the (so-called) *short* if, which is an if-statement without the keyword then. Please see the entry for if for more details.

Example

```
if (a<10) print "Hello ":print "World !"
```

This example demonstrates the difference between colon and newline as described above.

See also

if

Name

; — suppress the implicit newline after a `print`-statement

Synopsis

```
print "foo",bar;
```

Description

The semicolon (';') may only appear at the last position within a `print`-statement. It suppresses the implicit newline, which yabasic normally adds after each `print`-statement.

Put another way: Normally the output of each `print`-statement appears on a line by itself. If you rather want the output of many `print`-statements to appear on a single line, you should end the `print`-statement with a semicolon.

Example

```
print "Hello ";:print "World !"
```

This example prints `Hello World !` *in a single line*.

See also

`print`

Name

** or ^ — raise its first argument to the power of its second

Synopsis

```
print 2**b
print 3^4
```

Description

`**` (or `^`, which is an exact synonym), is the arithmetic operator of exponentiation; it requires one number to its left and a second one to its right; `**` then raises the first argument to the power of the second and returns the result. The result will only be computed if it yields a *real* number (as opposed to a *complex* number); this means, that the power can *not* be computed, if the first argument is negative and the second one is fractional. On the other hand, the second argument can be fractional, if the first one ist positive; this means, that `**` may be

used to compute arbitrary roots: e.g. `x**0.5` computes the square root of `x`.

Example

```
print 2**0.5
```

See also

sqrt

Name

< <= > >= = == <> != — Compare numbers or strings

See also

Comparing strings or numbers for some background.

Chapter 8. A few example programs

A very simple program

The program below is a very simple program:

```
repeat
  input "Please enter the first number, to add " a
  input "Please enter the second number, to add " b
  print a+b
until a=0 and b=0
```

This program requests two numbers, which it than adds. The process is repeated until you enter zero (or nothing) twice.

Graphics with bitmaps

yabasic allows to retrieve and put back rectangular regions of the screen with simple commands:

```
open window 200,200

rem prepare picture of a star
```

```
dim p(3,2)
for off=0 to 90 step 30
  for a=0 to 2
    phi = (off + 120*a)*2*pi/360
    p(a,0) = 50 + 20*cos(phi)
    p(a,1) = 50 + 20*sin(phi)
  next a
  fill triangle p(0,0),p(0,1),p(1,0),p(1,1),p(2,0),p(2,1)
next off

star$ = getbit$(30,30,80,80)
clear window

for a=0 to 200 step 10
  line a,0 to a,200:line 0,a to 200,a: rem draw some pattern on the screen
next a

for a=10 to 150 step 5
  saved$=getbit$(a,80,a+40,120): rem save old content of window
  putbit star$ to a,80,"t": rem put star at new location
  pause 0.5
  putbit saved$ to a,80: rem restore old window content
next a
```

This program moves moves the picture of a star across the graphics window. The first part of the program draws such a star and then retrieves the bitmap with `getbit$()`. Yabasic stores bitmaps within ordinary strings and so the star-bitmap can simply be stored within the variable `star$`.

Once the program has prepared the bitmap-string, it puts it back into the window with the `putbit`-command and various locations. Each time before the star-bitmap is put into the window, the prior content is saved within the variable `saved$` and restored later.

A menu to choose from

This example program presents a menu (e.g. a predefined set of 3 choices) and lets the user choose one of them. The menu is similar (but simpler) to the one employed in the demo of yabasic

```
//  Initialize menu
restore menudata
read menusize:dim menutext$(menusize)
for a=1 to menusize:read menutext$(a):next a

ysel=1
clear screen
blank$ = "                          "
hash$ = "######################################"
print colour("cyan","magenta") at(7,2) hash$
print colour("cyan","magenta") at(7,3) hash$
print colour("cyan","magenta") at(7,4) hash$
print colour("yellow","blue") at(8,3) " This is a simple menu to choose from "

for a=1 to menusize
  if (a=menusize) then yoff=1:else yoff=0:fi
  if (a=ysel) then
    print colour("blue","green") at(5,7+yoff+a) menutext$(a);
  else
    print at(5,7+yoff+a) menutext$(a);
  endif
next a
print at(3,15) "Select with cursor keys UP or DOWN (or letters u and d),"
```

```
print at(3,16) "Press RETURN or SPACE to choose, ESC to quit."

do

  k$=inkey$

  yalt=ysel

  switch k$
  case "up":case "u":
    if (ysel=1) then ysel=menusize else ysel=ysel-1 fi
    redraw()
    break
  case "down":case "d":
    if (ysel=menusize) then ysel=1 else ysel=ysel+1 fi
    redraw()
    break
  case " ":case "enter":case "right":
    if (ysel=menusize) end_it()
    print at(3,18) "You have chosen: " + menutext$(ysel) + blank$
    sleep 1
    print at(3,18) blank$ + blank$
    break
  case "esc":
    end_it()
  default:
    print at(3,18) "Invalid key: " + k$ + blank$
    sleep 1
    print at(3,18) blank$ + blank$
  end switch
loop

//  redraw line
sub redraw()
  local yoff
  if (yalt=menusize) then yoff=1:else yoff=0:fi
  print at(5,7+yalt+yoff) menutext$(yalt);
  if (ysel=menusize) then yoff=1:else yoff=0:fi
  print colour("blue","green") at(5,7+ysel+yoff) menutext$(ysel);
  return
end sub

// terminate program
sub end_it()
  print at(3,18) "Bye ..."
  sleep 1
  exit
end sub

//  Data section ...
label menudata
//  Data for main menu: Number and text of entries in main menu
data 4
data "   First Item            "
data "   Item number two       "
data "   Last Item             "
data "   None of the above     "
```

Some interesting aspects from top to bottom:

- The text of the various menu-items is initialized at the top of the program; the needed text-strings are kept in `data`-lines at the end of the program.
- User-input is aquired with the `inkey$`-function.
- A `switch`-statement is employed to process the input; several keys (e.g. **`right`** and **`enter`**) are handled in the same `case`-clause.

- Two subroutines `redraw()` and `end_it()` handle common work, that needs to be done at multiple places in the program.

Chapter 9. The Copyright of yabasic

yabasic may be copied under the terms of the *MIT License*, which is distributed with yabasic in the file `LICENSE`.

The MIT License grants extensive rights as long as you keep the copyright notice present in most files untouched. Here is a list of things that are possible under the terms of the MIT License:

- Put yabasic on your own homepage or CD and even charge for the service of distributing yabasic.
- Write your own yabasic-programs, pack your program and yabasic into a package and sell the whole thing.
- Modify yabasic and add or remove features, sell the modified version without adding the sources.