INTRODUCTION
COMMAND LINE OPTIONS
EDITOR
COMMANDER
MENU
ENTERING AND LEAVING LOGO
TOKENIZATION
DATA STRUCTURE PRIMITIVES
COMMUNICATION
ARITHMETIC
LOGICAL OPERATIONS
GRAPHICS
WORKSPACE MANAGEMENT
CONTROL STRUCTURES
MACRO COMMANDS
ERROR PROCESSING
SPECIAL VARIABLES
HELP COMMANDS
DIRECTORY COMMANDS
WINDOWS FUNCTIONS
BITMAP FUNCTIONS
MULTI MEDIA/MIDI/SOUND
NETWORKING
ABBREVIATIONS
INTRODUCTION

Why LOGO
Where to Start
**Why LOGO**

This introduction does not do LOGO justice but it’s a start. LOGO is a programming language, pure and simple. There are two models that languages come in, compiled and interpreted.

What is a compiled language?

In a compiled language the program is written and fed to a compiler. A compiler reads all your code and converts it to an executable form that your computer understands.

What is an interpreted language?

An interpreted language does not get compiled, instead, as each line is read by the interpreter it executes it. This is a slow process to execute (on the fly) like this, but has the advantage of not requiring a complete compile for each change. It's ideal in a learning environment.

So have guessed what type of language LOGO is yet?

Right, it's an interpreted language; at least this LOGO is anyway.

LOGO also has another unique feature not offered in many other languages (none that I know of). That is, what's called "Turtle Graphics".

What are turtle graphics?

Turtle graphics is a simple and powerful set of commands to manipulate a turtle.

Why do they call it a turtle?

The first version of LOGO used an electronic robot that resembled a turtle. In the case of a video screen (like this LOGO) it's simply a cursor (or pointer) of where the turtle is.

What does the turtle do?

It draws, lines mostly, on the screen.

The gap that turtle graphics fills is what traditional languages do not. That is, it gives immediate feedback. Immediate feedback makes it fun and easier to learn programming. The purpose of LOGO is to teach young and old how to program. It was modeled after a very popular and power language called LISP. It is as powerful as any other programming language.

— Why LOGO
— Why LOGO
— Why LOGO
— Why LOGO
— LOGO:0
Where to Start

Novices can start in LOGO without having to program at all by just learning how to command the turtle. Learning turtle graphics will teach the user about geometry (and they won't even know it). It's amazing how soon you can introduce the concept of programming once they grasp the turtle concept. Let's look at some simple examples:

Draw a square using the turtle

FORWARD 100
RIGHT 90
FORWARD 100
RIGHT 90
FORWARD 100
RIGHT 90
FORWARD 100
RIGHT 90

That was easy but too much typing, let's try again.

REPEAT 4 [FD 100 RT 90]

That's it? Yes, that's the same square. We did two things. We noticed too much redundant code in our first example, so we asked logo to repeat the same sequence 4 times. We also used abbreviated forms of the same commands. But we can still do better. A square is a popular item wouldn't you say? Wouldn't it be more useful just to say square when you wanted a square?

EDIT "square
<Editor will pop up>
TO SQUARE
REPEAT 4 [FD 100 RT 90]
END
<Exit Editor and save>

SQUARE
SQUARE

What's the TO and END for? It's to define a procedure (a small program) for the square. The TO can be thought of as "to do something", the END terminates the TO. Once square was "defined" we then called it twice. That's all you need to get a square now, just type square. There is a problem, however. It only draws squares of 100 by 100. Wouldn't it be better to draw any size square? It sure would and it's easy.

EDIT "square

TO SQUARE :length
REPEAT 4 [FD :length RT 90]
END

SQUARE 100
SQUARE 200

Note all we did is replace 100 with a variable name called :length. Now when we call square we must specify how big we want it. Above we asked logo to draw one square at 100x100 and another at 200x200. Note the ":" in front of the word length tells logo that length is a variable. However, we can still even do better. What's wrong now, you ask. Well, wouldn't it be better if we could draw something other than a square like a triangle?
TO TRIANGLE :length
REPEAT 3 [FD :length RT 120]
END

TO SQUARE :length
REPEAT 4 [FD :length RT 90]
END

TO PENTAGON :length
REPEAT 5 [FD :length RT 72]
END

TRIANGLE 100
SQUARE 100
HEXAGON 100

Lot of typing (programmers hate to type). Why? Because there are more things to break and when a change needs to be made it might have to be made in many places. Smaller is not always better but it usually helps. Lets try again.

TO POLYGON :length :sides
REPEAT :sides [FD :length RT 360.0/:sides]
END

POLYGON 100 3
POLYGON 100 4
POLYGON 100 5

What happened to TRIANGLE, SQUARE and HEXAGON? POLYGON now acts as every equal-sided polygon possible and with only one line of code! We now repeat the sequence based on how many :sides the caller asked for and we turn (RT) the amount of degrees appropriate for that shape. You may not realize it but this is PROGRAMMING.

Now that we have a program it's a good idea to save it to the disk. The edits you've made so far are all within Logo's memory and not on the disk. How do you save your work, easy.

SAVE "SHAPES.LGO
BYE

If you ever want to use these definitions again you'll have to reload them. How do you reload you work from disk?

LOAD "SHAPES.LGO

I could go on for ever, yes for ever, even to the point of writing LOGO within LOGO. This should get you started. Where can I learn more?

Almost every command in this help file has a simple example of using it.
Check out the examples (see Help Menu).
The MSWLogo online Video Tutorial (separate kit) only basics.
Jim Muller’s Great Logo Adventure Book primarily based on MSWLogo.
Brian Harvey’s MIT Press Books for more advanced programming.

What ever do make sure you have FUN.
# $K COMMAND LINE OPTIONS

Command line options can be set as properties of your icon for MSWLogo. Just add these switches after the LOGO.EXE.

-w
-h
-g
-o
-w width

This sets the width of your virtual drawing workspace to width. Be aware that the larger you set this the more memory is required. The default is 1000.

Example:

LOGO.EXE -w 2000 -h 2000
# S K * -h

-h height

This sets the width of your virtual drawing workspace to height. Be aware that the larger you set this the more memory is required. The default is 1000.

Example:

LOGO.EXE -w 2000 -h 2000

# _h
s _h
k _h
* LOGO:0
#sk* -p

-p

This enables PERSPECTIVE mode to be the default mode to start in.

Example:

LOGO.EXE -p

# ..p
s -p
k -p
* LOGO:0
This enables "Expert" mode.

In "Expert" mode the following occurs:

POS and POSXYZ will not "round" the returned data and you may see positions such as [0 -1.0e-12]. This is due to "normal" floating point errors. Rounding POS serves no purpose other than to make the POS look good. But in some applications this rounding can introduce error if you save and restore the position. In other words SETPOS POS has no side effect (will not round) in "Expert" mode.

MSWLogo will allow Multiple Edit Sessions. Multiple edit sessions can lead to unexpected behavior if you don't understand how MSWLogo manages the workspace. If procedures are lost or have unexpected changes then you should not be using expert mode.

MSWLogo will allow saves directly to the Workspace. Saving directly to workspace can lead to unexpected behavior if you don't understand how MSWLogo manages the workspace. If procedures are lost or unexpected changes then you should not be using expert mode.

Example:

LOGO.EXE -e
MSWLogo allows you to LOAD one or more filenames on the command line when starting MSWLogo. These files will be loaded before the interpreter starts reading commands from the commander. If you load a file that executes some program that includes a "bye" command, MSWLogo will run that program and exit. You can therefore write standalone programs in Logo and run them as new icons. Note, the "-l" switch must follow any other switches.

Example:

LOGO.EXE -l myprog.lgo
When you exit the editor, Logo loads the revised definitions and modifies the workspace accordingly. Multiple Edit sessions are supported. But be careful of having multiple edits going that include the same definition. The last Editor Exit (and saved) is what takes precedence. Also realize if you delete procedures, property lists or names while in the Editor they will be erased (see ERASE) from the environment at the time of exiting (and saving) the edit session you deleted them from.

**Editor Menu**

**Edit Errors**

**Editing with Clipboard**

**Context Sensitive Help**
**Edit Errors**

If an error occurs when Logo "loads" in your edit you will be prompted to reenter the Editor. This situation commonly occurs when a continuation "\" is missing within a list or missing a matching \"]. When the editor reloads you will be placed at the start of the definition that caused the error. When this occurs make sure to check the commander window for a hint of what the problem was when loading.
Editing with Clipboard

Logo's Editor and many controls support the Clipboard. The Clipboard is where most Windows application store data during cut and paste operations. This means that when you cut text from an application, such as Notepad, it can be pasted into Logo's Editor (the reverse is also true). Even Windows-Help supports the Clipboard. This means you can copy examples in this document directly to the editor (see HELP command).

The Input Box also supports the Clipboard. Only one line of text is supported between the Input Box and the Clipboard. Note that the Input Box does not have an Edit Menu like the Editor. You must use the "Short-Cut" keys for the desired actions (Cut CTRL-X, Copy CTRL-C, Paste CTRL-V).

The Output/Command-Recall List Box also supports the Clipboard. This means you can turn code you entered in the commander into procedure by copying the Commander Output to the clipboard and paste it into the editor. Or take text output you generated and paste it into another application.
Context Sensitive Help

Logo's Editor, Input Box and Output/Command-Recall List Box all support context sensitive Help. If you select a keyword (such as FORWARD) in these places (double-click works best in the editor) you can ask Logo to look up the keyword without going through the Help Menu followed by a Search and so on. You simple ask by clicking the right button on the mouse (or by hitting the F1 key), abbreviations are also supported. See also Context Sensitive Help to bring information back from the help system (like example code) for you to work with.
Editor Menu

The Edit Menu is how you communicate with the editor for some tasks.

Editor File Menu
Editor Edit Menu
Editor Set Menu
Editor Test Menu
Editor Help Menu
Editor File Menu

Editor File Print Command
Editor File Exit Command

# Editor_File_Menu
$ Editor File Menu
® Editor File Menu
* LOGO:0
# Editor File Print Command

Prints the contents of the editor.
**Editor File Exit Command**

Exits the editor and prompts you whether to save the work to the Logo workspace. When you exit the editor, Logo loads the revised definitions and modifies the workspace accordingly. Multiple Edit sessions are supported. But be careful of having multiple edits going that include the same definition. The last Editor Exited (and saved) is what takes precedence. Also realize if you delete procedures, property lists or names while in the Editor they will be erased (see **ERASE**) from the environment at the time of exiting (and saving) the edit session you deleted them from. See also **edit errors**.

---

# Editor_File_EXIT_Command
$ Editor File Exit Command
& Editor File Exit Command
+ LOGO:0
Editor Edit Menu

- Editor Edit Undo Command
- Editor Edit Cut Command
- Editor Edit Copy Command
- Editor Edit Paste Command
- Editor Edit Delete Command
- Editor Edit Clear All Command
Editor Edit Undo Command

Undo the last edit operation.
Editor Edit Cut Command

Copy the selected text to the clipboard and delete it. See also Editing With Clipboard.
Copy the selected text to the clipboard. See also Editing With Clipboard.
Paste the text in the clipboard to the cursor position. If Text is selected it will be replaced with the pasted text. See also Editing With Clipboard.
Delete the selected text.
Delete all text.
Editor Set Menu

Editor Set Font Command

# Editor_Set_Menu
$ Editor Set Menu
+ Editor Set Menu
* LOGO:0
Editor Set Font Command

You can change the Editors Font by going to the Font Menu of the Editor and Selecting the desired Font. The new font affects the entire Editor not just selected Text. The chosen font will remain selected even after MSWLogo has been restarted.
Editor Test Menu

You can test your code by selecting a section of code and clicking on Test! in the menu. This will take each line and send it to the commander for execution. Keep in mind that if you select code that has variables or other context that they must be defined in the selection or already defined in the workspace. Using this feature is as if you typed in that selection into the commander.
Editor Help Menu

Editor Help Index Command
Editor Help Editor Command
Editor Help Topic Search Command
Editor Help Index Command

Brings you to the help on MSWLogo. See also Context Sensitive Help.
Editor Help Editor Command

Brings you to the help on the Editor. See also Context Sensitive Help.
Editor Help Topic Search Command

Searches for the topic selected by the mouse in the online help. See also Context Sensitive Help.
COMMANDER

The commander is where you will spend most of your LOGO session. It is the means by which you instruct LOGO to do what you want. The most important control (box) within the commander is the INPUT BOX. It is located in the bottom left portion of the commander window. For information on the different controls (boxes) see the specific box below.

Input Box
Output/Command-Recall List Box
Execute Button
Status Button
Trace Button
Halt Button
Reset Button
Step Button
Pause Button
Input Box

The input box is tied to the Output/Command-Recall List Box and to the Execute Button. You can also edit the text in the input box. If what your typing doesn't fit, it will scroll automatically. Once your command is in the input box you need to execute it. You can do this by hitting ENTER or clicking on the execute button. Using the Up/Down arrow keys will automatically jump to (set focus to) the output/command-recall list box. This control also supports Context Sensitive Help and Editing with Clipboard.
# Output/Command-Recall List Box

The output/command-recall list box will record all output including what you type into the Input Box. You can select a command for re-execution by clicking on the desired line or by using the arrow keys. If something went out of view, use the scroll bar. Once a line is clicked on it is automatically copied to the Input Box. A double-click on the mouse will automatically execute what you're pointing at. This control also supports Context Sensitive Help and Editing with Clipboard.

- Output_Command_Recall_List_Box
- Output/Command-Recall List Box
- Output/Command-Recall List Box
- LOGO:0
Execute Button

The execute button executes what is in the Input Box and is also "PUSHED" when you hit ENTER key.
Status Button

This button pops up a status window telling you what LOGO is up to. Click it again to close the window. See also the Status and NoStatus commands.
**Trace Button**

The trace button turns on tracing for debugging your programs. Click again to disable tracing. You can turn tracing on or off even while Logo is running. Note that the trace button works independent of what you are tracing with the TRACE command. See also TRACE and UNTRACE commands.
Halt Button

The halt button immediately stops LOGO from processing any further. Logo is now waiting for a new command. See also the HALT command.

# S K • 

Halt_Button

$ Halt_Button

K Halt_Button

* LOGO:0
Reset Button

The reset button is like a CLEARSCREEN command and resets LOGO.

# $ K * Reset Button
↓ Reset Button
↓ Reset Button
↓ Reset Button
↑ LOGO:0
# $ K • Step Button

The step button turns on single stepping for debugging your programs. Click again to disable single stepping. You can turn single stepping on or off even while Logo it running. Note that the step button works independent of what you are stepping with the step command. See also STEP and UNSTEP commands.
Pause Button

The pause button stops LOGO so that you can examine variables, make changes or whatever. Once paused the pause button will show what depth you paused to. To continue you can issue a CONTINUE command or hit Cancel. You can also issue a PAUSE command within code to act as a "Break Point". You can think of pause and continue as sort of a Halt-n-Push and Pop-n-Continue of your state respectively.
The MENU is where you do high level tasks in LOGO, such as loading a LOGO program, loading an BITMAP image, setting up a printer or perhaps even read this help file. See the specific Menu item for more information.

**File Menu**
**Bitmap Menu**
**Set Menu**
**Zoom Menu**
**Help Menu**
# $K • File Menu

The File menu includes commands that enable you to operate on logo procedure files. Note that, as a side effect, any selection in this menu that has a directory in its dialog box will effectively change to that directory as the current working directory. For more information, select the File menu command name.

File New Command
File Load Command
File Save Command
File Save As Command
File Edit Command
File Erase Command
File Exit Command
**File New Command**

This will clear (DELETE, PURGE, ZAP) all procedures currently loaded in memory. It's like starting with a "New" session.

---

* File>New>Command
< File New Command
\ File New Command
@ File New Command
# LOGO:0
$\text{File Load Command}$

This allows you to load in procedures from disk into memory so that they can be executed or edited. See also $\text{LOAD}$ command.
File Save Command

This allows you to save everything that is loaded in memory onto the disk. See also SAVE command.

# $ K • File Save Command

# File_Save_Command
$ File Save Command
K File Save Command
* LOGO:0
# File Save As Command

This is the same as File Save Command, but prompts your for a new file name.
File Edit Command

This is how you edit procedures that have already been loaded (or developed) in memory. You will be prompted with all existing procedures (currently loaded within memory) and you can also enter a new one. See also EDIT command.
**File Erase Command**

This is how you erase procedures that have already been loaded (or developed) in memory. You will be prompted with all existing procedures (currently loaded within memory). See also `ERASE` command.

* File_Erase_Command
* File_Erase_Command
* File_Erase_Command
* LOGO:0
File Exit Command

This is how you exit MSWLogo. Also see the RYE command.
* § K * **Bitmap Menu**

The Bitmap menu includes commands that enable you to operate on bitmap files. For more information, see the specific Bitmap menu command name.

- Bitmap New Command
- Bitmap Load Command
- Bitmap Save Command
- Bitmap Save As Command
- Bitmap Active Area Command
- Bitmap Print Setup Command
- Bitmap Print Command
# $K * Bitmap New Command

This will clear the work done on the screen and create a new environment to save things in.
**Bitmap Load Command**

This allows you to read in an image you already saved in the past. The format of the file you save things in, is known as, a Microsoft Windows Bitmap (.BMP). You can interchange these files with other applications such as Paint. Note that these files can be BIG and can take a while to read or write. Also see the `BITLOAD` command.

See also the section on [USING COLOR](#).
**Bitmap Save Command**

This allows you to save a PICTURE (bitmap image) of your work on the computer's disk so that the computer will not forget it. It also allows you to add more work to an existing piece of work. REMEMBER if your image was generated with a LOGO program you really don't need to save it as an image unless you want to use the image in another application such as Paint or as a Wallpaper. Also see the `BITSAVE` command.

The format of the file you save things in, is known as, a Microsoft Windows Bitmap (.BMP). You can interchange these files with other applications such as Paint. Note that these files can be BIG and can take a while to read or write.

See also the `Bitmap Active Area Command`.
# Bitmap Save As Command

This is the same as Bitmap Save Command, but prompts you for a new file name.
# $ K • Bitmap Active Area Command

This allows you to select the work area to be printed or saved. The primary purpose of this option is performance and space. You no longer need to wait for the software to build a full image. It takes less time and less memory to print and disk space to save a partial image. As a side effect you can adjust where your image ends up on the page by selecting different extents. You may at anytime click on reset to put the extents back to the full image. MSWLogo will remember what you set them to even after you exit.

This command also allows you to set how many "Turtle Pixels" are mapped to an Inch on your printout. The default is 125 Turtle Pixels per inch of paper. Since MSWLogo’s workspace defaults to 1000x1000 turtle pixels, that equates to about 8x8 inches (1000/125x1000/125). You may increase this number to take advantage of higher resolution printers but your image will become smaller. You can compensate for this by increasing your work space (and you’re drawing size).

You can also set the active area with the Logo command SETACTIVEAREA and obtain it with the Logo command ACTIVEAREA.
• Bitmap Print Setup Command

This allows you to setup your printer before you print.
Bitmap Print Command

This allows you to print your work on the printer.
Set Menu

The Set menu allows you to SET some of the characteristics of how LOGO behaves when drawing.

Set PenSize Command
Set Label Font Command
Set Commander Font Command
Set PenColor Command
Set FloodColor Command
Set ScreenColor Command

* Set_Menu
" Set_Menu
$ Set Menu
^ Set Menu
° LOGO:0
Set PenSize Command

This command allows you to select the pen size in which the turtle will draw. You can also select the pen size with the Logo command SETPENSIZE and obtain it with the Logo command PENSIZE.
Set Label Font Command

This command allows you to select the font in which the command LABEL will draw with. You can also select the font with the Logo command SETLABELFONT and obtain it with the Logo command LABELFONT.
Set Commander Font Command

This command allows you to choose the Commander’s Font. The chosen font will remain selected even after MSWLogo has been restarted.
**Set PenColor Command**

This command allows you to select the pen color. You can also select the pen color with the Logo command `SETPENCOLOR` and obtain it with the Logo command `PENCOLOR`. 
Set FloodColor Command

This command allows you to select the flood color in which the turtle will fill. You can also select the flood color with the Logo command SETFLOODCOLOR and obtain it with the Logo command FLOODCOLOR.
Set ScreenColor Command

This command allows you to select the screen color. You can also select the screen color with the Logo command SETSCREENCOLOR and obtain it with the Logo command SCREENCOLOR.
Zoom Menu

The Zoom menu allows you set the ZOOM factor.

- Zoom In Command
- Zoom Out Command
- Zoom Normal Command

# $ K • Zoom Menu

$ Zoom_Menu
K Zoom_Menu
* LOGO:0
# $ K * Zoom In Command

This command allows you to double the zoom factor. You can also select the zoom factor with the Logo command ZOOM.

* Zoom_In_Command
* Zoom_In_Command
* Zoom_In_Command
* LOGO:0
**Zoom Out Command**

This command allows you to half the zoom factor. You can also select the zoom factor with the Logo command `ZOOM`.
# $K • Zoom Normal Command

This command allows you to reset the zoom factor to 1.0. You can also select the zoom factor with the Logo command ZOOM.
Help Menu

The Help menu allows you to learn more about LOGO. For more information, select the Help menu command name. Also see the HELP command.

Help Index Command
Help MCI Command
Help Demo Command
Help Tutorial Command
Help Examples Command
Help Release Notes Command
Help About Command

* Help_Menu
5 Help_Menu
K Help_Menu
* LOGO:0
# Help Index Command

This command puts you in Microsoft Windows Help for LOGO.
Help MCI Command

This puts you into the MCI help file. It explains the syntax of the arguments to the MCI command.
Help Demo Command

This runs the DEMO program from the Logo library.

# $ K •

" Help Demo Command
$ Help Demo Command
K Help Demo Command
* LOGO:0
Help Tutorial Command

This runs the Tutorial program from the Logo library.
Help Examples Command

This uses NotePad to present a Brief outline of all the supplied Examples.
Help Release Notes Command

This uses NotePad to present the Release notes for this release of MSWLogo.
Help About Command

This gives some details about the LOGO program like its version.

# $ K * Help About Command
$ Help About Command
K Help About Command
* LOGO:0
ENTERING AND LEAVING LOGO

To start MSWLogo, just click on the MSWLogo icon. To leave Logo, enter the command BYE or File Exit Command.

If you invoke a procedure that has not been defined, Logo first looks for a file in the current directory named proc.igo where "proc" is the procedure name in lower case letters. If such a file exists, Logo loads that file. If the missing procedure is still undefined, or if there is no such file, Logo then looks in the library directory for a file named proc (no "igo") and, if it exists, loads it. If neither file contains a definition for the procedure, then Logo signals an error. Several procedures that are primitive in most versions of Logo are included in the default library, so if you use a different library you may want to include some or all of the default library in it.
Names of procedures, variables, and property lists are case-insensitive. So are the special words END, TRUE, and FALSE. Case of letters is preserved in everything you type, however.

Within square brackets, words are delimited only by spaces and square brackets. \([2+3]\) is a list containing one word.

After a quotation mark outside square brackets, a word is delimited by a space, a square bracket, or a parenthesis.

A word not after a quotation mark or inside square brackets is delimited by a space, a bracket, a parenthesis, or an infix operator +,-,=><. Note that words following colons are in this category. Note that quote and colon are not delimiters.

A word consisting of a question mark followed by a number (e.g., ?3), when unparsed (i.e., where a procedure name is expected), is treated as if it were the sequence \((?3)\)

making the number an input to the ? procedure. (See \texttt{TEMPLATE-BASED ITERATION}) This special treatment does not apply to words read as data, to words with a non-number following the question mark, or if the question mark is backslashed.

A line (an instruction line or one read by \texttt{READLIST} or \texttt{READWORD}) can be continued onto the following line if its last character is a tilde (~). \texttt{READWORD} preserves the tilde and the newline; \texttt{READLIST} does not.

A semicolon begins a comment in an instruction line. Logo ignores characters from the semicolon to the end of the line. A tilde (~) as the last character still indicates a continuation line, but not a continuation of the comment.

Example:

\begin{verbatim}
print "abc:comment ~
def abc
def
\end{verbatim}

Semicolon has no special meaning in data lines read by \texttt{READWORD} or \texttt{READLIST}, but such a line can later be reparsed using \texttt{RUNPARSE} and then comments will be recognized. If a tilde is typed at the terminal for line continuation, Logo will issue a tilde as a prompt character for the continuation line.

To include an otherwise delimiting character (including semicolon or tilde) in a word, precede it with backslash (\(\backslash\)). If the last character of a line is a backslash, then the newline character following the backslash will be part of the last word on the line, and the line continues onto the following line. To include a backslash in a word, use \(\\backslash\). If the combination backslash-newline is entered at the terminal, Logo will issue a backslash as a prompt character for the continuation line. All of this applies to data lines read with \texttt{READWORD} or \texttt{READLIST} as well as to instruction lines.

In MSWLogo there is no "prompt character" nor is \(<\text{CR}>\) (carriage return) passed from the input control box of the commander. However, MSWLogo has added the "\(n\)" control character which will translate to a \(<\text{CR}>\).

Example:

\begin{verbatim}
print "Hello\nhow\nare
Hello
how
are
\end{verbatim}
you

This will work in a procedure or from the input control box.

A character entered with backslash is **EQUAL** to the same character without the backslash, but can be distinguished by the **BACKSLASHEDP** predicate. (In Europe, backslashing is effective only on characters for which it is necessary: whitespace, parentheses, brackets, infix operators, backslash, vertical bar, tilde, quote, question mark, colon, and semicolon.)

An alternative notation to include otherwise delimiting characters in words is to enclose a group of characters in vertical bars (**|**). All characters between vertical bars are treated as if they were letters. In data read with **READWORD** the vertical bars are preserved in the resulting word. In data read with **READLIST** (or resulting from a **PARSE** or **RUNPARSE** of a word) the vertical bars do not appear explicitly; all potentially delimiting characters (including spaces, brackets, parentheses, and infix operators) appear as though entered with a backslash. Within vertical bars, backslash may still be used; the only characters that must be backslashed in this context are backslash and vertical bar themselves.

Characters entered between vertical bars are forever special, even if the word or list containing them is later reparsed with **PARSE** or **RUNPARSE**. The same is true of a character typed after a backslash, except that when a quoted word containing a backslashed character is runparsed, the backslashed character loses its special quality and acts thereafter as if typed normally. This distinction is important only if you are building a Logo expression out of parts, to be **RUN** later, and want to use parentheses. For example,

```
PRINT RUN (SE "\\( 2 " * 3 "\))
```

will print 5, but

```
RUN (SE "MAKE "\( 1 2")
```

will create a variable whose name is open-parenthesis. (Each example would fail if vertical bars and backslashes were interchanged.)
DATA STRUCTURE PRIMITIVES

CONSTRUCTORS
SELECTORS
MUTATORS
PREDICATES (Data)
QUERIES
CONSTRUCTORS

WORD
LIST
SENTENCE
FPUT
LPUT
ARRAY
MARRAY
LISTTOARRAY
ARRAYTOLIST
COMBINE
REVERSE
GENSYM
word WORD word1 word2
word (WORD word1 word2 word3 ...)

Outputs a word formed by concatenating its inputs.

word:(WORD) Concatenation of inputs.

word1:(WORD) First word to be concatenated.
word2:(WORD) Second word to be concatenated.

Example:

show word "o "k
ok
show (word "a "o "k "t.o)  
aokto
# $ K * LIST

list LIST thing1 thing2
list (LIST thing1 thing2 thing3 ...)

Outputs a list whose members are its inputs, which can be any Logo object (word, list, or array).

list:(LIST) Newly formed list of its inputs.

thing1:(THING) First thing to be a member of the output list.
thing2:(THING) Second thing to be a member of the output list.

Example:

show (list "This "is "a "List)
[This is a List]
show list [1 2 3] [a b c]
[[1 2 3] [a b c]]

Example2:

make "red 100
make "green 100
make "blue 100
show (list :red :green :blue)
[100 100 100]
list SENTENCE thing1 thing2
list SE thing1 thing2
list (SENTENCE thing1 thing2 thing3 ...)
list (SE thing1 thing2 thing3 ...)

Outputs a list whose members are its inputs, if those inputs are not lists.
Outputs the members of its inputs, if those inputs are lists.

list (LIST) Sentence formed by inputs.

thing1:(THING) First thing to be a member of the output sentence.
thing2:(THING) Second thing to be a member of the output sentence.

Example:

show (se "A "Sentence "is "simply "a "list "of "words)  
[A Sentence is simply a list of words]
**newlist FPUT thing list**

Outputs a **list** equal to its second input with one extra member, the first input, at the beginning.

**newlist:** (LIST) New list formed by inputs.

**thing:** (THING) Thing to be added to front new list.

**list:** (LIST) Existing list to be added to.

**Example:**

```
show fput 1 [2 3 4]
[1 2 3 4]
```
newlist LPUT thing list

Outputs a list equal to its second input with one extra member, the first input, at the end.

newlist:(LIST) New list formed by inputs.
thing:(THING) Thing to be added to end of new list.
list:(LIST) Existing list to be added to.

Example:

show lput 5 [1 2 3 4]
[1 2 3 4 5]
array ARRAY size
array (ARRAY size origin)

Outputs an array of size elements (must be a positive integer), each of which initially is an empty list. Array elements can be selected with ITEM and changed with SETITEM. The first element of the array is element number 1, unless an origin input (must be an integer) is given, in which case the first element of the array has that number as its index. (Typically 0 is used as the origin if anything.) Arrays are printed by PRINT and friends, and can be typed in, inside curly braces; indicate an origin with {a b c}@0.

array:(ARRAY) Newly formed array.
size:(INTEGER) Size of the new array.
origin:(INTEGER) Where to start indexing the new array.

Example:

make "myarray (array 3 0)
setitem 2 :myarray 1
setitem 1 :myarray 2
setitem 0 :myarray 3
show :myarray
{3 2 1}
MDARRAY

mdarray MDARRAY sizelist  (library procedure)
mdarray (MDARRAY sizelist origin)

Outputs a multi-dimensional array. The first input must be a list of one or more positive integers. The second input, if present, must be a single integer that applies to every dimension of the array.

mdarray:MDARRAY) Newly formed multi-dimensional array.
sizelist:(LIST) List of sizes (each an integer) of the new multi-dimensional array.
origin:(INTEGER) Where to start indexing the new multi-dimensional array.

Example:

make "myarray (mdarray [2 3] 0)
mdsetItem [0 0] :myarray 1
mdsetItem [0 1] :myarray 2
mdsetItem [0 2] :myarray 3
mdsetItem [1 0] :myarray 4
mdsetItem [1 1] :myarray 5
mdsetItem [1 2] :myarray 6
show :myarray
{{[1 2 3] [4 5 6]}}
LISTTOARRAY

array LISTTOARRAY list (library procedure)
array (LISTTOARRAY list origin)

Outputs an array of the same size as the input list, whose elements are the members of the input list.

array:(ARRAY) Newly converted array.
list:(LIST) Existing list to be converted to an array.
origin:(INTEGER) Where to start indexing the new array.

Example:

show listtoarray [1 2 3]
{1 2 3}
ARRAYLIST

list ARRAYLIST array  (library procedure)

Outputs a list whose members are the elements of the input array. The first member of the output is the first element of the array, regardless of the array's origin.

list:(LIST) Newly converted list.

array:(ARRAY) Existing array to be converted to a list.

Example:

show arraytolist (1 2 3)
[1 2 3]
thing COMBINE thing1 thing2  (library procedure)

If thing2 is a word, outputs WORD thing1 thing2.
If thing2 is a list, outputs FPUT thing1 thing2.

thing: (LIST) Newly formed thing from its combined inputs.

thing1: (THING) First thing to be combined.
thing2: (THING) Second thing to be a combined.

Example:

```log
show combine "a "b
ab
show combine "a [b]
[a b]
```

# $ K * COMBINE

$ COMBINE

k COMBINE

* LOGO:0
REVERSE

newlist REVERSE list  (library procedure)

Outputs a list whose members are the members of the input list, in reverse order.

newlist:(LIST) Newly formed list.

list:(LIST) Existing list to be reversed.

Example:

show reverse [1 2 3]
[3 2 1]
**GENSYM**

`word` **GENSYM**  (library procedure)

Outputs a unique `word` each time it's invoked. The words are of the form G1, G2, etc.

`word:(WORD)` Newly formed symbol.

Example:

```lisp
(make gensym 1
(show :g1
1
(make gensym "Hello"
(show :g2
Hello
```

* GENSYSM
  * GENSYSM
  * GENSYSM
  * GENSYSM
  * LOGO:0
SELECTORS

FIRST
FIRSTS
LAST
BUTFIRST
BUTEIRSTS
BUTLAST
ITEM
MDITEM
PICK
REMOVE
REMDUP
QUOTED

* SELECTORS
* SELECTORS
* SELECTORS
* LOGO:0
firstthing FIRST thing

If the input is a word, outputs the first character of the word.
If the input is a list, outputs the first member of the list.
If the input is an array, outputs the origin of the array (that is, the INDEX OF the first element of the array).

`firstthing`: (THING) First thing of input.

`thing`: (THING) Existing thing to be extracted from.

Example:

```plaintext
print first [1 2 3]
1
print first "Hello
H
```
# § K • FIRSTS

**firsts** FIRSTS list

Outputs a list containing the **FIRST** of each member of the input list. It is an error if any member of the input list is empty. (The input itself may be empty, in which case the output is also empty.

**firsts** : (LIST) New list of first of each element of input.

**list** : (LIST) Existing list to be extracted from.

Example:

```lisp
show firsts [[1 2 3] [a b c]]
[1 a]
```

Example:

```lisp
to transpose :matrix
if emptyp first :matrix [op []]
op fput firsts :matrix transpose bfs :matrix end
```
# § K • LAST

lastthing LAST thing

If the input is a word, outputs the last character of the word.
If the input is a list, outputs the last member of the list.

lastthing:(THING) Last thing of input.

thing:(THING) Existing thing to be extracted from.

Example:

```logo
print last [1 2 3]
3
print last "Hello
```

* LAST
* LAST
* LAST
* LOGO:0
# $ K BUTFIRST

newthing BUTFIRST thing
newthing BF thing

If the input is a word, outputs a word containing all but the first character of the input.
If the input is a list, outputs a list containing all but the first member of the input.

newthing(THING) Every thing but the first of its input.
thing(THING) Existing thing to be extracted from.

Example:

show butfirst [1 2 3]
[2 3]
show butfirst "Hello
e110
```
# $ K • BUTFIRSTS

newlist BUTFIRSTS list
newlist BFS list

Outputs a list containing the BUTFIRST of each member of the input list. It is an error if any member of the input list is empty or an array. (The input itself may be empty, in which case the output is also empty.)

newlist:(LIST) Everything but the firsts of its input.

list:(LIST) Existing list to be extracted from.

Example:

show butfirsts [[[1 2 3] [a b c]]
[[[2 3] [b c]]]
```
newthing BUTLAST thing
newthing BL thing

If the input is a word, outputs a word containing all but the last character of the input.
If the input is a list, outputs a list containing all but the last member of the input.

newthing.(THING) Every thing but the last of its input.
thing.(THING) Existing thing to be extracted from.

Example:

show butlast [1 2 3]
[1 2]
show butlast "Hello
Hello
ITEM

item ITEM index thing

If the thing is a word, outputs the index-th character of the word. If the thing is a list, outputs the index-th member of the list. If the thing is an array, outputs the index-th element of the array. An index starts at 1 for words and lists; the starting index of an array is specified when the array is created.

item:(THING) The item extracted from its input.
index:(INTEGER) The index of the item to be extracted.
thing:(THING) Existing thing to be extracted from.

Example:

show item 2 [a b c]
b
show item 3 "ABC
c
MDITEM

item MDITEM indexlist mdarray (library procedure)

Outputs the an element of the multidimensional array selected by the list of numbers indexlist.

item: (THING) The item extracted from its input.

indexlist: (LIST) The list of indices (INTEGERS) to specify the item to be extracted.

mdarray: (MDARRAY) Existing multidimensional array to be extracted from.

Example:

show mditem [2 2] {{0 1} {2 3}}

3
# $ K • PICK

item PICK thing (library procedure)

Outputs a randomly chosen item of the input thing.

item:(THING) The item extracted from its input.

thing:(THING) Existing thing to be extracted from.

Example:

show pick [1 2 3]
2
show pick [1 2 3]
2
show pick [1 2 3]
3
show pick "Hello

* PICK
$ PICK
K PICK
* LOGO:0
# $K • REMOVE

newthings REMOVE thing things  (library procedure)

Outputs newthings which is the input things with every member equal to thing removed.

newthings:(THING) All of things with thing removed.

thing:(THING) Existing thing to be removed.

things:(THING) Existing thing to be removed from.

Example:

(show remove "b [a b c b]"
 [a c]
(show remove "e "Hello

Hello

* REMOVE
* REMOVE
* REMOVE
* LOGO:0
newthings REMDUP things (library procedure)

Outputs newthings which is a copy of things with duplicate members removed. If two or more members of the input are equal, the rightmost of those members is the one that remains in the output.

newthings: (THING) All of things with duplicates removed.

things: (THING) Existing thing to be removed from.

Example:

show remdup [a b c b]
[a c b]
show remdup "Hello
Hello
# QUOTED

**QUOTED**

*QUOTED_thing_ QUOTED thing 

(library procedure)

Outputs its input, if a list.
Outputs its input with a quotation mark prepended, if a word.

**QUOTED_thing_ :** (THING) Quoted thing.

**thing_ :** (THING) Existing thing to be quoted.

Example:

```lisp
show "Hello
Hello
show quoted "Hello
"Hello
```
# $K$ MUTATORS

SETITEM
MDSETITEM
.SETFIRST
.SETBF
.SETITEM
PUSH
POP
QUEUE
DEQUEUE

* MUTATORS
§ MUTATORS
*K MUTATORS
+ LOGO:0
SETITEM

SETITEM index array value

Command that replaces the index-th element of array with the new value. This command will ensure that the resulting array is not circular, i.e., value may not be a list or array that contains the same array.

array:(ARRAY) Existing array to have an item set in.
index:(INTEGER) The index of the item to be placed in the array.
value:(THING) Thing to be placed at index of array.

Example:

make "myarray (array 3 0)
setitem 2 :myarray 1
setitem 1 :myarray [2]
setitem 0 :myarray 3
show :myarray
{3 [2] 1}
MDSETITEM

MDSETITEM indexlist mdarray value (library procedure)

Command that replaces the element of mdarray chosen by indexlist with the new value.

mdarray: (ARRAY) Existing array to have an item set in.
indexlist: (LIST) The list of indices (INTEGERS) to specify the item placed in the mdarray.
value: (THING) Thing to be placed at index of mdarray.

Example:

make "myarray (mdarray [2 3] 0)
mdsetItem [0 0] :myarray 1
mdsetItem [0 1] :myarray 2
mdsetItem [0 2] :myarray 3
mdsetItem [1 0] :myarray 4
mdsetItem [1 1] :myarray 5
mdsetItem [1 2] :myarray 6
show :myarray
  {{1 2 3} {4 5 6}}
.SETFIRST

.list value

Command that changes the first member of list to be value. WARNING: Primitives whose names start with a period are DANGEROUS. Their use by non-experts is not recommended. The use of .SETFIRST can lead to circular list structures, which will get some Logo primitives into infinite loops; unexpected changes to other data structures that share storage with the list being modified; and the permanent loss of memory if a circular structure is released.

list:(LIST) Existing list to have its first element set.
value:(THING) Thing to be first of list.

Example:

make "mylist [1 2 3]
.setfirst :mylist 0
show :mylist
[0 2 3]

# $K * .SETFIRST

$ .SETFIRST
K .SETFIRST
* LOGO:0
# § K • .SETBF

.SETBF list value

Command that changes the butfirst of list to be value. WARNING: Primitives whose names start with a period are DANGEROUS. Their use by non-experts is not recommended. The use of .SETBF can lead to circular list structures, which will get some Logo primitives into infinite loops; unexpected changes to other data structures that share storage with the list being modified; Logo crashes and core dumps if the butfirst of a list is not itself a list; and the permanent loss of memory if a circular structure is released.

list:(LIST) Existing list to have its butfirst set in.
value:(LIST) List to be the butfirst of list.

Example:

make  "mylist [1 2 3]
.setbf :mylist [a b]
show :mylist
[1 a b]
#SETITEM

.SETITEM index array value

Command that changes the index-th element of array to be value, like SETITEM, but without checking for circularity. WARNING: Primitives whose names start with a period are DANGEROUS. Their use by non-experts is not recommended. The use of .SETITEM can lead to circular arrays, which will get some Logo primitives into infinite loops; and the permanent loss of memory if a circular structure is released.

array:(ARRAY) Existing array to have an item set in.
index:(INTEGER) The index of the item to be placed in the array.
value:(THING) Thing to be placed at index of array.

Example:

make "myarray (array 3 0)
.setitem 2 :myarray 1
.setitem 1 :myarray 2
.setitem 0 :myarray 3
show :myarray
{3 2 1}
PUSH

PUSH stackname thing (library procedure)

Command that adds the thing to the stack whose name is stackname. This variable must have a list as its value; the initial value should be the empty list. New members are added at the front of the list. Later, you can POP thing off the stackname.

stackname: (WORD) A word which is the name of an existing stack to be pushed on.
thing: (THING) Thing to be pushed onto stack.

Example:

make "mystack []
push "mystack 1
push "mystack 2
show :mystack
[2 1]
show pop "mystack
2
show pop "mystack
1
# POP

thing POP stackname  (library procedure)

Outputs the most recent thing that was pushed on, using PUSH, off the stack whose name is stackname and removes that member from the stack.

thing: (THING) Thing to be popped off of stack.

stackname: (WORD) A word which is the name of an existing stack to be popped off.

Example:

make "mystack []
push "mystack 1
push "mystack 2
show :mystack [2 1]
show pop "mystack 2
show pop "mystack 1

* POP
* POP
* POP
* LOGO:0
 QUEUE

**QUEUE queuename thing**  (library procedure)

Command that adds the **thing** to the queue whose name is **queuename**. This variable must have a list as its value; the initial value should be the empty list. New members are added at the back of the list. Later, you can **DEQUEUE** thing from the **queuename**.

**queuename**: (WORD) A word which is the name of an existing queue to be queued on.
**thing**: (THING) Thing to be queued onto queue.

Example:

```
make "myqueue ()
queue "myqueue 1
queue "myqueue 2
show :myqueue
[1 2]
show dequeue "myqueue
1
show dequeue "myqueue
2
```
DEQUEUE

thing DEQUEUE queue

Outputs the least recent (oldest) thing queued on, using QUEUE, off the queue whose name is queue name and removes that member from the queue.

thing: (THING) Thing to be dequeued from queue.

queue: (WORD) A word which is the name of an existing queue to be dequeued from.

Example:

make "myqueue []
queue "myqueue 1
queue "myqueue 2
show :myqueue
[1 2]
show dequeue "myqueue 1
show dequeue "myqueue 2

* DEQUEUE
^ DEQUEUE
& DEQUEUE
* LOGO:0
# $K • PREDICATES (Data)

WORDP  
LISTP  
ARRAYP  
EMPTYP  
EQUALP  
BEFOREP  
EQ  
MEMBERP  
SUBSTRINGP  
NUMBERP  
BACKSLASHEDP

# $K • PREDICATES (Data)  
$ PREDICATES (Data)  
* PREDICATES (Data)  
* LOGO:0
$K\ast$ WORDP

**truth** WORDP **thing**

Outputs TRUE if the input **thing** is a word, FALSE otherwise.

**truth:** (BOOLEAN) Truth of the test for being a word.

**thing:** (THING) Thing to be tested for being a word.

Example:

```lisp
show wordp "Hello
true
show wordp [Hello]
false
show wordp [Hello]
false
```

# $K \ast$ WORDP
$K \ast$ WORDP
$K \ast$ WORDP
$K \ast$ LOGO:0
LISTP

truth LISTP thing

Outputs TRUE if the input thing is a list, FALSE otherwise.

truth: (BOOLEAN) Truth of the test for being a list.

thing: (THING) Thing to be tested for being a list.

Example:

show listp "Hello
false
show listp [Hello]
true
show listp {Hello}
false
# $K • ARRAYP

truth ARRAYP thing

Outputs TRUE if the input thing is an array, FALSE otherwise.

truth: (BOOLEAN) Truth of the test for being a array.

thing: (THING) Thing to be tested for being a array.

Example:

show arrayp "Hello
false
show arrayp [Hello]
false
show arrayp {Hello}
true
truth EMPTYP thing

Outputs TRUE if the input thing is the empty word or the empty list, FALSE otherwise.

truth: (BOOLEAN) Truth of the test for being empty

thing: (THING) Thing to be tested for being empty.

Example:

show emptyp [1 2 3]
false
show emptyp []
true
$\text{EQUALP}$

$\text{truth \ EQUALP \ thing1 \ thing2}$

Outputs TRUE if the inputs $\text{thing1}$ and $\text{thing2}$ are equal. FALSE otherwise. Two numbers are equal if they have the same numeric value. Two non-numeric words are equal if they contain the same characters in the same order. If there is a variable named $\text{CASEIGNORED}$ whose value is TRUE, then an upper case letter is considered the same as the corresponding lower case letter. (This is the case by default.) Two lists are equal if their members are equal. An array is only equal to itself; two separately created arrays are never equal even if their elements are equal. (It is important to be able to know if two expressions have the same array as their value because arrays are mutable; if, for example, two variables have the same array as their values then performing $\text{SETITEM}$ on one of them will also change the other.)

$\text{truth} \ (\text{BOOLEAN}) \ \text{Truth of the test for being equal.}$

$\text{thing1} \ (\text{THING}) \ \text{Thing to be tested for being a equal to thing2.}$

$\text{thing2} \ (\text{THING}) \ \text{Thing to be tested for being a equal to thing1.}$

Example:

```
show equalp 1 1
 true
show equalp 1 2
 false
show equalp [1 2 3] [1 2 3]
 true
show equalp [1 2 3] [3 2 1]
 false
```
truth BEFOREP word1 word2

Outputs TRUE if word1 comes before word2 in ASCII collating sequence (for words of letters, in alphabetical order). Case-sensitivity is determined by the value of CASEIGNOREDP. Note that if the inputs are numbers, the result may not be the same as with LESSP; for example, BEFOREP 3 12 is false because 3 collates before 1.

truth: (BOOLEAN) Truth of the test for being before.

word1: (WORD) Word to be tested for being before word2.
word2: (WORD) Word to be tested for being after word1.

Example:

show beforep "ABC "abd
true
show beforep "abd "ABC
false
truth EQ thing1 thing2

Outputs TRUE if its two inputs thing1 and thing2 are the same object, so that applying a mutator to one will change the other as well. Outputs FALSE otherwise, even if the inputs are equal in value. WARNING: Primitives whose names start with a period are DANGEROUS. Their use by non-experts is not recommended. The use of mutators can lead to circular data structures, infinite loops, or Logo crashes.

truth:(BOOLEAN) Truth of the test for being equal.

thing1:(THING) Thing to be tested for being a equal to thing2.
thing2:(THING) Thing to be tested for being a equal to thing1.

Example:

make "x l
make "y l
show .eq :x :y
false
show .eq :x :x
true

* LOGO:0
truth MEMBERP thing1 thing2

If thing2 is a list or an array, outputs TRUE if thing1 is EQUAL to a member or element of thing2, FALSE otherwise. If thing2 is a word, outputs TRUE if thing1 is EQUAL to a substring of thing2, FALSE otherwise.

Note that this behavior for words is different from other dialects, in which thing1 must be a single character to make MEMBERP true with thing2 a word.

truth: (BOOLEAN) Truth of the test for being member.

thing1: (THING) Thing to be tested for being a member of thing2.
thing2: (THING) Thing to be tested for having a member thing1.

Example:

```
show memberp 1 [1 2 3]
true
show memberp 4 [1 2 3]
false
```
**SUBSTRINGP**

truth SUBSTRINGP thing1 thing2

If thing2 is a list or an array, outputs TRUE if thing1 is EQUALP to a member or element of thing2, FALSE otherwise. If thing2 is a word, outputs TRUE if thing1 is EQUALP to a substring of thing2. FALSE otherwise. Note that this behavior for words is different from other dialects, in which thing1 must be a single character to make MEMBERP true with thing2 a word.

truth: (BOOLEAN) Truth of the test for being a substring.

g1: (THING) Thing to be tested for being a substring of thing2.
g2: (THING) Thing to be tested for having a substring thing1.

Example:

```
symbol memberp ab abc
true
symbol memberp ac abc
false
```
\# \$ K • \textbf{NUMBERP}

\textbf{truth} \textbf{NUMBERP} \textbf{thing}

Outputs \texttt{TRUE} if the input \texttt{thing} is a number, \texttt{FALSE} otherwise.

\texttt{truth}:(\texttt{BOOLEAN}) Truth of the test for being a number

\texttt{thing}:(\texttt{THING}) Thing to be tested for being a number.

Example:

\begin{verbatim}
show numberp 1
true
show numberp [1]
false
\end{verbatim}
# $K • BACKSLASHEDP

**truth BACKSLASHEDP character**

Outputs TRUE if the input character was originally entered into Logo with a backslash (\) before it to prevent special syntactic meaning, FALSE otherwise. (In Europe, outputs TRUE only if the character is a backslashed space, tab, newline, or one of ([][*+/%=<>:*?;~])

**truth:** (BOOLEAN) Truth of the test for being backslashed

**character:** (WORD) Character to be tested for being backslashed.

Example:

```logo
show backslashedp "a
false
```
COUNT
ASCII
RAWASCII
CHAR
MEMBER
LOWERCASE
UPPERCASE
STANDOUT
PARSE
RUNPARSE
TIME
TIMEMILLI
# $K * COUNT

**number** COUNT thing

Outputs the **number** of characters in the input **thing**, if the input **thing** is a word; outputs the **number** of members or elements in the input **thing**, if it is a list or an array. (For an array, this may or may not be the index of the last element, depending on the array's origin.)

**number**:(NUMBER) Count of elements in thing.

**thing**:(THING) Thing to be counted.

Example:

```plaintext
count [1 2 3]
3
count "ab" 2
```

* COUNT
* COUNT
* COUNT
* LOGO:0
# $K ASCII

`number ASCII character`

Outputs the `number` (in the United States, between 0 and 127) that represents the input `character` in the ASCII code.

`number`: (NUMBER) ASCII code of input.
`character`: (WORD) Character to get ASCII code of.

Example:

```awk
show ascii "a
97
show ascii "A
65
show ascii "b
98
```
number RAWSCII character

Outputs the number (in the United States, between 0 and 127) that represents the input character in the ASCII code.

number:(NUMBER) RAWSCII code of input.
character:(WORD) Character to get RAWSCII code of.

Example:

```
show rawascii "a"
  97
show rawascii "A"
  65
show rawascii "b"
  98
```
# $ K • CHAR

character CHAR number

Outputs the character represented in the ASCII code by the input number, which must be an number between 0 and 127.

character:(WORD) Character based on ASCII code input.

number:(NUMBER) ASCII code of character you wish to obtain.

Example:

```
  show char 97
  a
  show char 65
  A
  show char 98
  b
```
# MEMBER

thing MEMBER thing1 thing2

If thing2 is a word or list and if MEMBER with these inputs would output TRUE, outputs the portion of thing2 from the first instance of thing1 to the end. If MEMBER would output FALSE, outputs the empty word or list according to the type of thing2. It is an error for thing2 to be an array.

thing1:(THING) Thing formed by function of inputs.
thing2:(THING) Thing to be tested for being a member of thing2.

Example:

```
show member "b [a b c d] [b c d]
show member "c [a b c d] [c d]
```
newword LOWERCASE word

Outputs a word which is a copy of the input word, but with all uppercase letters changed to the corresponding lowercase letter. (In the United States, letters that were initially read by Logo preceded by a backslash are immune to this conversion.)

newword:(WORD) Lowercase word of its input.

word:(WORD) Word to be lowercased.

Example:

show lowercase "Hello
hello
word UPPERCASE word

Outputs a word which is a copy of the input word, but with all lowercase letters changed to the corresponding uppercase letter. (In the United States, letters that were initially read by Logo preceded by a backslash are immune to this conversion.)

newword:(WORD) Uppercase word of its input.

word:(WORD) Word to be uppercased.

Example:

show uppercase "Hello
HELLO
word STANDOUT thing

(Not supported in MSWLogo yet)

Outputs a word that, when printed, will appear like the input but displayed in standout mode (boldface, reverse video, or whatever your terminal does for standout). The word contains terminal-specific magic characters at the beginning and end; in between is the printed form (as if displayed using TYPE) of the input. The output is always a word, even if the input is of some other type, but it may include spaces and other formatting characters. Note: a word output by STANDOUT while Logo is running on one terminal will probably not have the desired effect if printed on another type of terminal.

newword:(WORD) Standout word of its input.

thing:(WORD) Thing to be Standout.
list PARSE word

Outputs the list that would result if the input word were entered in response to a READLIST operation. That is, PARSE READWORD has the same value as READLIST for the same characters read.

list:(LIST) Parsed input.

word:(WORD) Word to be parsed.

Example:

show parse "Hello
[Hello]
RUNPARSE

list RUNPARSE thing

Outputs the list that would result if the input thing (word or list) were entered as an instruction line; characters such as infix operators and parentheses are separate members of the output. Note that sublists of a runparsed list are not themselves runparsed.

list:(WORD) Runparsed version of its input.

thing:(THING) Word or list to be runparsed.

Example:

show runparse "a<b
[a < b]
# $ K • TIME

list TIME

Outputs the current time on the system as a list of words.

list: (list) A list containing the current system time.

Example:

show time
[Wed Jul 14 23:34:08 1993]
# TIMEMILLI

number TIMEMILLI

Outputs the time since Windows Started in milliseconds, use for precision timing.

number:(INTEGER) Time in milliseconds.

Example:

make "start timemilli
repeat 36 [ellipse 100 200 rt 5]
show timemilli - :start
8189
COMMUNICATION

TRANSMITTERS
RECEIVERS
FILE ACCESS
Communications
KEYBOARD and MOUSE ACCESS
**TRANSMITTERS**

Note: If there is a variable named `PRINTDEPTHLIMIT` with a nonnegative integer value, then complex list and array structures will be printed only to the allowed depth. That is, members of members of... of members will be allowed only so far. The elements or members omitted because they are just past the depth limit are indicated by an ellipsis for each one, so a too-deep list of two elements will print as `[... ...]`.

If there is a variable named `PRINTWIDTHLIMIT` with a nonnegative integer value, then only the first so many elements or members of any array or list will be printed. A single ellipsis replaces all missing objects within the structure. The width limit also applies to the number of characters printed in a word, except that a `PRINTWIDTHLIMIT` between 0 and 9 will be treated as if it were 10 when applied to words. This limit applies not only to the top-level printed object but to any substructures within it.

PRINT
TYPE
SHOW
PRINT

PRINT thing
PR thing
(PRINT thing1 thing2 ...)
(PR thing1 thing2 ...)

Command that prints the input or inputs to the current write stream (initially the terminal). All the inputs are printed on a single line, separated by spaces, ending with a newline. If an input is a list, square brackets are not printed around it, but brackets are printed around sublists. Braces are always printed around arrays.

thing:(thing) A thing you wish to be printed.
thing1:(thing) First thing you wish to be printed.
thing2:(thing) Second thing you wish to be printed.

Example:

print "Hello
Hello
print [Hello how are you]
Hello how are you
TYPE

TYPE thing
(TYPE thing1 thing2 ...)

Command that prints the input or inputs like PRINT, except that no newline character is printed at the end and multiple inputs are not separated by spaces. Note: printing to the terminal is ordinarily "line buffered"; that is, the characters you print using TYPE will not appear on the screen until either a newline character is printed (for example, by PRINT or SHOW) or Logo tries to read from the keyboard (at the request of your program). This buffering makes the program much faster than it would be if each character appeared immediately, and in most cases the effect is not disconcerting.

thing:(thing) A thing you wish to be typed.
thing1:(thing) First thing you wish to be typed.
thing2:(thing) Second thing you wish to be typed.

Example:

type "Hello
type "How
type "Are
print "You
HelloHowAreYou
SHOW thing
(SHOW thing1 thing2 ...)

Command that prints the input or inputs like PRINT, except that if an input is a list it is printed inside square brackets.

thing: (thing) A thing you wish to be shown.
thing1: (thing) First thing you wish to be shown.
thing2: (thing) Second thing you wish to be shown.

Example:

show [1 2 3]
[1 2 3]
print [1 2 3]
1 2 3
# $K$ RECEIVERS

READLIST
READWORD
READCHAR
READCHARS
SHELL

$^a$ RECEIVERS
$^b$ RECEIVERS
$^c$ RECEIVERS
$^* LOGO:0$
READLIST

list READLIST
list RL

Reads a line from the read stream (initially the terminal) and outputs that line as a list. The line is separated into elements as though it were typed in square brackets in an instruction. If the read stream is a file, and the end of file is reached, READLIST outputs the empty word (not the empty list). READLIST process's backslash, vertical bar, and tilde characters in the read stream; the output list will not contain these characters but they will have had their usual effect. READLIST does not, however, treat semicolon as a comment character.

list: (LIST) What was read output to a list.

Example:

show readlist
<Enter (Hello how are you <CR>) in dialog box>
[Hello how are you]
WORD

word READWORD
word RW

Reads a line from the read stream and outputs that line as a word. The output is a single word even if the line contains spaces, brackets, etc. If the read stream is a file, and the end of file is reached, READWORD outputs the empty list (not the empty word). READWORD process's backslash, vertical bar, and tilde characters in the read stream. In the case of a tilde used for line continuation, the output word DOES include the tilde and the newline characters, so that the user program can tell exactly what the user entered. Vertical bars in the line are also preserved in the output. Backslash characters are not preserved in the output, but the character following the backslash has 128 added to its representation. Programs can use BACKSLASHEDP to check for this code. (In Europe, backslashedness is preserved only for certain characters. See BACKSLASHEDP.)

word:(WORD) What was read output to a word.

Example:

show readword
<Enter (Hello<CR>) in dialog box>
Hello
# $ * READCHAR

class READCHAR
class RC

Reads a single character from the read stream and outputs that character as a word. If the read stream is a file, and the end of file is reached, READCHAR outputs the empty list (not the empty word). If the read stream is a terminal, echoing is turned off when READCHAR is invoked, and remains off until READLIST or READWORD are invoked or a Logo prompt is printed. Backslash, vertical bar, and tilda characters have no special meaning in this context.

character:(WORD) What was read output to a single character word.

Example:

show readchar
<Enter (H<CR>) in dialog box>
H
READCHARS

word READCHARS num
word RCS num

(Not supported in MSWLogo when reader is [] (keyboard). See KEYBOARDON)

Reads num characters from the read stream and outputs those characters as a word. If the read stream is a file, and the end of file is reached, READCHARS outputs the empty list (not the empty word). If the read stream is a terminal, echoing is turned off when READCHARS is invoked, and remains off until READLIST or READWORD are invoked or a Logo prompt is printed. Backslash, vertical bar, and tildle characters have no special meaning in this context.

word:(WORD) What was read output to a word.

num:(INTEGER) Number of characters to read.

Example:

openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
show rcs 5
Hello
setread []
close "dummy.fil
**SHELL**

**truth** SHELL command

Outputs the result of running **command** as a shell command. If the command is a literal list in the instruction line, and if you want a backslash character sent to the shell, you must use `\` to get the backslash through Logo's reader intact. The output is a boolean based on successful launch.

**truth**(BOOLEAN) Truth as to the success of shelling the command.

**command**(THING) The thing you wish to shell (usually a list of words).

Example:

```
show shell [notepad c:\\test.txt]
true
```
FILE ACCESS

OPENREAD
OPENWRITE
OPENAPPEND
OPENUPDATE
CLOSE
ALLOPEN
CLOSEALL
ERASEFILE
DRIBBLE
NODRIBBLE
SETREAD
SETWRITE
READER
WRITER
SETREADPOS
SETWRITEPOS
READPOS
WRITEPOS
EOPF
OPENREAD

OPENREAD filename
(OPENREAD filename binarymode)

Command that opens the named file for reading. The read position is initially at the beginning of the file.

filename: (WORD) The name of the file you wish to open for read.
binarymode: (BOOLEAN) True to open file in binary mode False to open file in text mode.

Example:

openwrite "dummy.fil"
setwrite "dummy.fil"
print "Hello"
print [Good Bye]
setwrite []
close "dummy.fil"

openread "dummy.fil"
setread "dummy.fil"
repeat 2 [show readlist]
[Hello]
[Good Bye]
setread []
close "dummy.fil"
OPENWRITE

OPENWRITE filename
(OPENWRITE filename binarymode)

Command that opens the named file for writing. If the file already existed, the old version is deleted and a new, empty file is created.

filename: (WORD) The name of the file you wish to open for write.
binarymode: (BOOLEAN) True to open file in binary mode False to open file in text mode.

Example:

openwrite "dummy.fil"
setwrite "dummy.fil"
print "Hello"
print [Good Bye]
setwrite []
close "dummy.fil"

openread "dummy.fil"
setread "dummy.fil"
repeat 2 [show readlist]
[Hello]
[Good Bye]
setread []
close "dummy.fil"
OPENAPPEND

OPENAPPEND filename
(OPENAPPEND filename binarymode)

Command that opens the named file for writing. If the file already exists, the write position is initially set to the end of the old file, so that newly written data will be appended to it.

filename:(WORD) The name of the file you wish to open for append.
binarymode:(BOOLEAN) True to open file in binary mode False to open file in text mode.

Example:

openwrite "dummy.fil"
setwrite "dummy.fil"
print "Hello"
setwrite []
close "dummy.fil"

openappend "dummy.fil"
setwrite "dummy.fil"
print [Good Bye]
setwrite []
close "dummy.fil"

openread "dummy.fil"
setread "dummy.fil"
repeat 2 [show readlist]
[Hello] [Good Bye]
setread []
close "dummy.fil"
OPENUPDATE filename
(OPENUPDATE filename binarymode)

Command that opens the named file for reading and writing. The read and write position is initially set to the end of the old file, if any. Note: each open file has only one position, for both reading and writing. If a file opened for update is both READER and WRITER at the same time, then SETREADPOS will also affect WRITEPOS and vice versa. Also, if you alternate reading and writing the same file, you must SETREADPOS between a write and a read, and SETWRITEPOS between a read and a write.

filename:(WORD) The name of the file you wish to open for update.
binarymode:(BOOLEAN) True to open file in binary mode False to open file in text mode.

Example:

openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
print [Good Bye]
setwrite []
close "dummy.fil

openupdate "dummy.fil
setread "dummy.fil
show readlist
 [Hello]
setwrite "dummy.fil
setreadpos 7
print [And how are you today]
setwrite []
setread "dummy.fil
setreadpos 0
repeat 3 [show readlist]
 [Hello]
 [And how are you today]
close "dummy.fil
CLOSE filename

Command that closes the named file.

filename: (WORD) The name of the file you wish to close.

Example:

openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
print [Good Bye]
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
repeat 2 [show readlist]

[Hello]
[Good Bye]
setread []
close "dummy.fil
# $K • ALLOPEN

list ALLOPEN

Outputs a list whose members are the names of all files currently open. This list does not include the dribble file, if any.

list(LIST) The list of all open files.

Example:

openwrite "dummy1.fil
openwrite "dummy2.fil
show allopen
[dummy1.fil dummy2.fil]
CLOSEALL  (library procedure)

Command that closes all open files. Abbreviates FOREACH ALLOPEN [CLOSE ?]

Example:

openwrite "dummy1.fil"
openwrite "dummy2.fil"
show allopen
[dummy1.fil dummy2.fil]
closeall
show allopen
[]
ERASEFILE

ERASEFILE filename
ERF filename

Command that erases (deletes, removes) the named file, which should not currently be open.

filename: (WORD) The name of the file you wish to erase.

Example:

openwrite "dummy.fil"
setwrite "dummy.fil"
print "Hello"
setwrite []
close "dummy.fil"

openread "dummy.fil"
setread "dummy.fil"
repeat 2 [show readlist]
[Hello]
setread []
close "dummy.fil"

erasefile "dummy.fil"
openread "dummy.fil"

File system error: I can't open that file
# $ K • DRIBBLE

DRIBBLE filename

Command that creates a new file whose name is the input, like OPENWRITE, and begins recording in that file everything that is read from the keyboard or written to the terminal. That is, this writing is in addition to the writing to WRITER. The intent is to create a transcript of a Logo session, including things like prompt characters and interactions.

filename: (WORD) The name of the file you wish to dribble to.

Example:

dribble "dummy.fil
fd 100
rt 90
nodribble
openread "dummy.fil
setread "dummy.fil
repeat 3 [show readlist]
[fd 100]
[rt 90]
[nodribble]
setread []
close "dummy.fil

* DRIBBLE
$ DRIBBLE
K DRIBBLE
* LOGO:0
NODRIBBLE

Command that stops copying information into the dribble file, and closes the file.

Example:

dribble "dummy.fil
fd 100
rt 90
nodribble
openread "dummy.fil
setread "dummy.fil
repeat 3 [show readlist]
[fd 100]
[rt 90]
[nodribble]
setread []
close "dummy.fil
**SETREAD**

**SETREAD filename**

Command that makes the named file the read stream, used for READLIST, etc. The file must already be open with OPENREAD or OPENUPDATE. If the input is the empty list, then the read stream becomes the terminal, as usual. Changing the read stream does not close the file that was previously the read stream, so it is possible to alternate between files.

filename: (WORD) The name of the file you wish to select for reading.

Example:

```plaintext
openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
print [Good Bye]
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
repeat 2 [show readlist]
[Hello]
[Good Bye]
setread []
close "dummy.fil
```
SETWRITE

SETWRITE filename

Command that makes the named file the write stream, used for PRINT, etc. The file must already be open with OPENWRITE, OPENAPPEND, or OPENUPDATE. If the input is the empty list, then the write stream becomes the terminal, as usual. Changing the write stream does not close the file that was previously the write stream, so it is possible to alternate between files.

filename: (WORD) The name of the file you wish to select for writing.

Example:

close "dummy.fil"
setwrite "dummy.fil"
print "Hello"

Example:

close "dummy.fil"
setwrite "dummy.fil"
print [Good Bye]
setwrite []
close "dummy.fil"

Example:

close "dummy.fil"
setwrite "dummy.fil"
print [Good Bye]
setread []
close "dummy.fil"
# $K* READER

filename READER

Outputs the name of the current read stream file, or the empty list if the read stream is the terminal.

filename: (WORD) The name of the file that is the current reader.

Example:

openread "dummy.fil"
setread "dummy.fil"
show reader
dummy.fil
filename: (WORD) The name of the file that is the current writer.

Example:

openwrite "dummy.fil
setwrite *dummy.fil
show writer
dummy.fil
SETREADPOS

Command that sets the file pointer of the read stream file so that the next READLIST, etc., will begin reading at the charpos-th character in the file, counting from 0. (That is, SETREADPOS 0 will start reading from the beginning of the file.) Meaningless if the read stream is the terminal.

charpos:(INTEGER) The position you wish to read from.

Example:

  openwrite "dummy.fil
  setwrite "dummy.fil
  print "Hello
  print [Good Bye]
  setwrite []
  close "dummy.fil

  openread "dummy.fil
  setread "dummy.fil
  show readlist
  [Hello]
  setreadpos 0
  show readlist
  [Hello]
  setread []
  close "dummy.fil

* SETREADPOS
$ SETREADPOS
% SETREADPOS
* LOGO:0
SETWRITEPOS charpos

Command that sets the file pointer of the write stream file so that the next PRINT, etc., will begin writing at the charpos-th character in the file, counting from 0. (That is, SETWRITEPOS 0 will start writing from the beginning of the file.) Meaningless if the write stream is the terminal.

charpos:(INTEGER) The position you wish to write to.

Example:

openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
setwritepos 0
type "]
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
show readlist
[Jello]
setread []
close "dummy.fil
$K READPOS

charspos READPOS

Outputs the file character position, as a number, of the current read stream file.

charspos: (INTEGER) The current character position the reader is at.

Example:

openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
print [Good Bye]
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
repeat 2 [show readpos show readlist]
0
[Hello]
7
[Good Bye]
setread []
close "dummy.fil

\" READPOS
\$ READPOS
\^ READPOS
\* LOGO:0
WRITEPOS

position WRITEPOS

Outputs the file position, as a number, of the current write stream file.

charpos:(INTEGER) The current character position the writer is at.

Example:

Note: the output had to be put in a list until the writer is returned to screen (]).

openwrite "dummy.fil
setwrite "dummy.fil
make "history []
make "history Iput writepos :history
print "Hello
make "history Iput writepos :history
print [Good Bye]
make "history Iput writepos :history
setwrite []
close "dummy.fil
show :history
[0 7 17]
truth EOFP

Predicate that outputs TRUE if there are no more characters to be read in the read stream file, FALSE otherwise.

truth:(BOOLEAN) Truth as to whether the reader is at the End Of File.

Example:

openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
print [Good Bye]
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
repeat 2 [show readlist show eofp]
[Hello]
false
[Good Bye]
true
setread []
close "dummy.fil
Communications

All commands that start with PORTxxxx are for serial (RS-232) and parallel port communications. All commands that start with INPORTx or OUTPORTx are for "hardware" port communications.

PORTOPEN
PORTCLOSE
PORTMODE
PORTREADARRAY
PORTREADCHAR
PORTWRITEARRAY
PORTWRITECHAR
INPORT
OUTPORT
INPORTB
OUTPORTB
DLLLOAD
DLLCALL
DLLFREE

* Communications
○ Communications
# Communications
+ LOGO:0
PORTOPEN

PORTOPEN port

This command is used to gain access to the serial and parallel ports of your computer. Once the desired port is open you can read (PORTREADCHAR or PORTREADARRAY) or write (PORTWRITECHAR or PORTWRITEARRAY) to it. You can set the characteristics of the port with PORTMODE. Only one port can be open at any given time. Once finished with the port you should close the port with PORTCLOSE. Do not confuse this command with INPORT and OUTPORT commands which are for "hardware" port communications.

port:(WORD) is the name of the port wish to open (e.g. COM1-COM4 and LPT1-LPT3)

Example:

portopen "com1
portclose
PORTCLOSE

This command closes a port that was opened by PORTOPEN.

Example:

portopen *com1
portclose
PORTFLUSH queue

This command is used to flush the ports input or output queue.

queue: (INTEGER) Specifies which queue you want flushed 0 (output) and 1 (input).

Example:

portflush 1
PORTMODE

PORTMODE mode

This command is used to set the mode (speed, parity, data bits, and stop bits) of the port.

Note that the characteristics (such as speed and flow control) can also be set through the Control Panel PORTS icon. Which is the only way you can specify control flow.

mode: (WORD) is mode you wish to set to (*COMn:SPEED,PARITY,DATA,STOP). Same format as the DOS MODE command.

Example:

portmode *com1:9600,n,8,1
PORTREADARRAY

actual PORTREADARRAY count buffer

This command will read the currently open port and write the data into the given buffer array. It will attempt to read count many characters from the port if they are available. It will output the actual number of bytes read.

actual: (INTEGER) is the actual number of bytes read off the port.

count: (INTEGER) is the number of characters to read from the port. You can use a larger number than the array size if you just want to fill the array.

buffer: (ARRAY) is a buffer to which input data is written to. It will be filled with byte size integers.

Example:

portopen "com1"
print se [Sending...] portwritearray 3 listtoarray map [ascii ?] arraytolist
listtoarray "at"
Sending... 2
show portwritechar 13 1
wait 60
make "buff [0 0 0 0 0 0 0 0 0 0]
print se [Receiving...] portreadarray 10 :buff
Receiving... 9
print se [Data Rx... ] map [char ?] remove [] arraytolist :buff
Data Rx... at | |
O K |
portclose
PORTREADCHAR

This command will read one byte from the currently open port and output it as an integer. It will output "-1" if no character was available.

data (INTEGER) is the byte data read from the port (-1 if none available or error).

Example:

portopen *com1
show portwritechar ascii "a"
1
show portwritechar ascii "t"
1
show portwritechar 13
1
wait 60
print "Reading...
Reading...
repeat 10 {make "x portreadchar if not :x = -1 [type char :x]]
print ".
ok
portclose
PORTWRITEARRAY

actual PORTWRITEARRAY count buffer

This command will write to the currently open port with the data in the given buffer array. It will attempt to write count
many characters to the port if possible. It will output the actual number of bytes written.

actual: (INTEGER) is the actual number of bytes written to the port.

count: (INTEGER) is the number of characters to write to the port. You can use a larger number than the array size if
you just want to dump the whole array.

buffer: (ARRAY) is a buffer to which output data is read from. It must contain byte size integers.

Example:

portopen "com1
print se [Sending... ] portwritearray 3 listtoarray map [ascii ?] arraytolist
listtoarray "at
Sending... 2
show portwritechar 13
1
wait 60
make "buff {0 0 0 0 0 0 0 0 0}
print se [Receiving... ] portreadarray 10 :buff
Receiving... 9
print se [Data Rx... ] map [char ?] remove [] arraytolist :buff
Data Rx... at | |
OK |
portclose
PORTWRITECHAR

actual PORTWRITECHAR data

This command will write one data byte to the currently open port and output the actual number of bytes written (0 or 1).

actual: (INTEGER) is the number of bytes written (0 or 1).

data: (INTEGER) is the byte data that is to be written to the port.

Example:

portopen "com1
show portwritechar ascii "a
1
show "y portwritechar ascii "t
1
show "y portwritechar 13
1
wait 60
print "Reading...
Reading...
repeat 10 {make "x portreadchar if not :x = -1 [type char :x]} print ".
at||
OK|

portclose
INPORT

Data INPORT portid

This command will read 2 bytes (16 bits) from the specified portid and output the data as an integer. The portid specifies a "Hardware" Port. Do not confuse this command with PORTOPEN and PORTREADCHAR which are for communications through the Serial and Parallel Ports. See also INPORTB and OUTPORT.

Data:(INTEGER) is the 2 bytes of data read from portid.

Portid:(INTEGER) Specifies which hardware port you wish to read from.

Example:

show inport 198
25730
OUTPORT

OUTPORT portid data

This command will write data as 2 bytes to the specified portid. The portid specifies a "Hardware" Port. Do not confuse this command with PORTOPEN and PORTWRITECHAR which are for communications through the Serial and Parallel Ports. See also OUTPORTB and INPORT.

portid:(INTEGER) Specifies which hardware port you wish to write to.
data:(INTEGER) Is the 2 bytes of data that is to be written to the portid.

Example:

outport 198 0
* INPORTB
* INPORTB
* INPORTB
* LOGO:0

# § K * INPORTB

data INPORTB portid

This command will read 1 byte (character) from the specified portid and output the data as an integer. The portid specifies a "Hardware" Port. Do not confuse this command with PORTOPEN and PORTREADCHAR which are for communications through the Serial and Parallel Ports. See also INPORT and OUTPORT.

data:(INTEGER) Is the byte of data read from portid.

portid:(INTEGER) Specifies which hardware port you wish to read from.

Example:

show inportb 198
130
OUTPORTB portid data

This command will write data as 1 byte to the specified portid. The portid specifies a "Hardware" Port. Do not confuse this command with PORTOPEN and PORTWRITECHAR which are for communications through the Serial and Parallel Ports. See also OUTPORT and INPORT.

portid:(INTEGER) Specifies which hardware port you wish to write to.

data:(INTEGER) Is the byte of data that is to be written to the portid.

Example:

outport 198 0

* OUTPORTB
* OUTPORTB
* OUTPORTB
* LOGO:0
**# $ K • DLLLOAD**

**DLLLOAD dllname**

This command will load the DLL (Dynamic Linked Library) called `dllname` into MSWLogo. Once loaded you can call functions with `DLLCALL` and once you finished with the DLL or need to use another DLL you can free the DLL with `DLLFREE`.

`dllname:` (WORD) Specifies which DLL you wish to load (DLLs do not have to end in ".DLL").

Example:

```
dllload "user.exe
dllfree
```
# $ K * DLLCALL

**status** DLLCALL function

This command will call the specified function and Output as a **type**. It will pass argument **arg1** as a **type1** and pass **arg2** as a **type2** and so on. The **function** must be in the DLL that was loaded with DLLLOAD. The given list must be paired and it must contain at least 4 items (typef function type1 arg1) even if arg1 is not used. This command is meant for experienced users and you should be familiar with the Windows calling conventions to use it. If you do not properly match the argument list the results will be unpredictable (including crashing Windows). This function can handle a wide variety of functions but certainly not all. Since it does allow you to get outside of MSWinLogo you can certainly write your own DLL with a compatible call interface to call more complex interfaces. Note if the argument list is "void" then use [... 'v 0].

Note that on Win32 the argument are in **REVERSE** order.

**status**(TYPEF) Output of function called of types specified in the input.

**function**(LIST) Has the format [**type function** type1 **arg1** type2 **arg2** type3 **arg3** ...] where:

**type**(WORD) Specifies what the type of argument or function is. Where

- v void
- w WORD
- l DWORD
- f DOUBLE
- s LPSTR

**function**(WORD) Is the name of the function to be called.

**arg**(THING): Almost any thing logo supported type than can be appropriately mapped to the above types.

Example:

dllload "user.exe
dllcall [w MessageBox w 0 s MyMessage s [Hello How are you] w 0]
<see MessageBox pop up>
dllfree
# $K$ *DLLFREE*

**DLLFREE**

This command will free the DLL that was loaded with [DLLLOAD](#).

Example:

```plaintext
dllload "user.exe
dllfree
```

---

* DLLFREE
* DLLFREE
* DLLFREE
* LOGO:0
KEYBOARD and MOUSE ACCESS

KEYP
KEYBOARDON
KEYBOARDOFF
KEYBOARDVALUE
MOUSEON
MOUSEOFF
MOUSEPOS
CLEARTEXT
SETCURSOR
CURSOR
SETMARGINS

$ KEYBOARD_and_MOUSE_ACCESS
$ KEYBOARD and MOUSE_ACCESS
$ KEYBOARD and MOUSE_ACCESS
* LOGO:0
KEYP

(KEYP)

(not supported in MSWLogo yet, see KEYBOARDON)

Predicate that outputs TRUE if there are characters waiting to be read from the read stream. If the read stream is a file, this is equivalent to NOT EOF. If the read stream is the terminal, then echoing is turned off and the terminal is set to CBREAK (character at a time instead of line at a time) mode. It remains in this mode until some line-mode reading is requested (e.g., READLIST). The UNIX operating system forgets any pending characters when it switches modes, so the first KEYP invocation will always output FALSE.

# KEYP
# KEYP
# KEYP
# KEYP
# LOGO:0
KEYBOARDON

**KEYBOARDON keydown**

*KEYBOARDON keydown keyup*

This command will enable you to directly trap keyboard events. To obtain what key was involved call **KEYBOARDVALUE** in your keydown or keyup procedure. Note that the "Screen" window must have focus (NOT the commander) to catch the key events. You can force this by **SETFOCUS** [MSWLogo SCREEN] when you issue this command. The second form of this command detects independently the keydown and keyup event. Note that all the "callbacks" for the keyboard are automatically run in a **NOYIELD** mode.

**keydown**: (LIST) is a (short) list of logo commands (or a procedure name) to execute when the key is pushed.

**keyup**: (LIST) is a (short) list of logo commands (or a procedure name) to execute when the key is let go.

Example:

```
keyboardon [print char keyboardvalue]
setfocus [MSWLogo Screen]
<a>
  a
</a>
<b>
  b
</b>
keyboardoff
```
# $K $ KEYBOARDOFF

KEYBOARDOFF

This command will disable trapping of keyboard events.

Example:

keybordon [print char keyboardvalue]  
setfocus [MSWLogo SCREEN]  
<a>  
</a>  
<b>  
</b>  
keybordoff
KEYBOARDVALUE

key KEYBOARDVALUE

This command will output the value of the last key pushed DOWN or let UP.

key:(INTEGER) Is the ASCII value of the last Keyhit event.

Example:

keyboardon [print char keyboardvalue]
setfocus [MSWLogo SCREEN]
<a>
a
<b>
b
keyboardoff

Remember "MSWLogo Screen" must have focus (be selected) when you hit the keys.
MOUSEON leftbuttondown leftbuttonup rightbuttondown rightbuttonup move

This command will enable you to directly trap mouse events. To obtain where the mouse was when a button was pushed or the mouse moved call MOUSEPOS in your button or move procedure. Note that the "MSWLogo Screen" window must have focus (NOT the commander) to catch the mouse events. Note that all the "callbacks" for the mouse are automatically run in a NOYIELD mode.

leftbuttondown:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the Left Button is pushed DOWN.
leftbuttonup:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the Left Button is let UP.
rightbuttondown:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the Right Button is pushed DOWN.
rightbuttonup:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the Right Button is let UP.
move:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the mouse is moved.

Example:

pu mouseon [setpos mousepos pd] [pu] [1] [1] [setpos mousepos]
<move mouse around hold button down to draw>
mouseoff

* MOUSEON
§ MOUSEON
K MOUSEON
* LOGO:0
**MOUSEOFF**

MOUSEOFF

This command will disable trapping of mouse events.

Example:

```
pu
mouseon [setpos mousepos pd] [pu] [] [] [setpos mousepos]
<move mouse around hold button down to draw>
mouseoff
```
# $K * MOUSEPOS

pos MOUSEPOS

This command will output the position of the mouse at the last mouse event. The mouse must first be turned on with call to MOUSEON.

pos:(LIST) is the position ([x y]) of the last mouse event.

Example:

pu
mouseon [setpos mousepos pd] [pu] [] [] [setpos mousepos]
<move mouse around hold button down to draw>
mouseoff
# $ K • CLEARTEXT

CLEARTEXT
CT

Command that clears the text screen of the terminal. In MSWLogo it clears the output/command recall list box.

Example:

print "Hello
Hello
cleartext
SETCURSOR

(not supported in MSWLogo yet)

SETCURSOR vector

Command where the input is a list of two numbers, the x and y coordinates of a screen position (origin in the upper left corner, positive direction is southeast). The screen cursor is moved to the requested position. This command also forces the immediate printing of any buffered characters.
CURSOR

(not supported in MSWLogo yet)

Outputs a list containing the current x and y coordinates of the screen cursor. Logo may get confused about the
current cursor position if, e.g., you type in a long line that wraps around or your program prints escape codes that
affect the terminal strangely.

# § K • CURSOR

® CURSOR
® CURSOR
® CURSOR
* LOGO:0
SETMARGINS vector

(not supported in MSWLogo yet)

Command where the input must be a list of two numbers, as for SETCURSOR. The effect is to clear the screen and then arrange for all further printing to be shifted down and to the right according to the indicated margins. Specifically, every time a newline character is printed (explicitly or implicitly) Logo will type x_margin spaces, and on every invocation of SETCURSOR the margins will be added to the input x and y coordinates. (CURSOR will report the cursor position relative to the margins, so that this shift will be invisible to Logo programs.) The purpose of this command is to accommodate the display of terminal screens in lecture halls with inadequate TV monitors that miss the top and left edges of the screen.

* SETMARGINS
$ SETMARGINS
& SETMARGINS
# SETMARGINS
* LOGO:0
ARITHMETIC

NUMERIC OPERATIONS
PREDICATES (Arithmetic)
RANDOM NUMBERS
PRINT FORMATTING
BITWISE OPERATIONS

* ARITHMETIC
* ARITHMETIC
* ARITHMETIC
* LOGO:0
$\text{NUMERIC\_OPERATIONS}$

$\text{SUM}$
$\text{DIFFERENCE}$
$\text{MINUS}$
$\text{PRODUCT}$
$\text{QUOTIENT}$
$\text{REMAINDER}$
$\text{INT}$
$\text{ROUND}$
$\text{ABS}$
$\text{SIGN}$
$\text{SORT}$
$\text{POWER}$
$\text{EXP}$
$\text{LOG10}$
$\text{LN}$
$\pi$
$\text{TAN}$
$\text{RADTAN}$
$\text{SIN}$
$\text{RADSIN}$
$\text{COS}$
$\text{RADCOS}$
$\text{ARCTAN}$
$\text{RADARCTAN}$
$\text{ARCSIN}$
$\text{RARCSIN}$
$\text{ARCCOS}$
$\text{RADARCCOS}$
# $K • SUM

num SUM num1 num2
num (SUM num1 num2 num3 ...)
num1 + num2

Outputs the sum of its inputs.

num:(NUMBER) Result of sum.

num1:(NUMBER) First number to sum.
num2:(NUMBER) Second number to sum.

Example:

show 2 + 3

5
# $ K • DIFFERENCE

```
um DIFFERENCE num1 num2
num1 - num2
```

Outputs the difference of its inputs. Minus sign means infix difference in ambiguous contexts (when preceded by a complete expression), unless it is preceded by a space and followed by a non space.

```
num:(NUMBER) Result of difference.
num1:(NUMBER) Number to subtract from.
num2:(NUMBER) Number to subtract.
```

Example:

```
show 3 - 2
1
```
# $ K • MINUS

neg MINUS num
- num

Outputs the negative of its input. Minus sign means unary minus if it is immediately preceded by something requiring an input, or preceded by a space and followed by a non space. There is a difference in binding strength between the two forms:

MINUS 3 + 4 means -(3+4)
- 3 + 4 means (-3)+4

neg:(NUMBER) Minus of input.

num:(NUMBER) Number to minus.

Example:

show 2 - -3
5
PRODUCT

num PRODUCT num1 num2
num (PRODUCT num1 num2 num3 ...)
num1 * num2

Outputs the product of its inputs.

num:(NUMBER) Result of product.
num1:(NUMBER) First number of product.
num2:(NUMBER) Second number of product.

Example:

show 2 * 3
6
# § K • QUOTIENT

num QUOTIENT num1 num2
num (QUOTIENT num2)
num1 / num2

Outputs the quotient of its inputs. The quotient of two integers is an integer if and only if the dividend is a multiple of the divisor. (In other words, QUOTIENT 5 2 is 2.5, not 2, but QUOTIENT 4 2 is 2, not 2.0 -- it does the right thing.) With a single input, QUOTIENT outputs the reciprocal of the input.

num1:(NUMBER) Result of quotient.

num1:(NUMBER) Dividend of quotient (this is 1 when not specified).
num2:(NUMBER) Divisor of quotient.

Example:

show 6 / 3
2
show 3 / 2
1.5

* QUOTIENT
• QUOTIENT
§ QUOTIENT
K QUOTIENT
* LOGO:0
### REMAINDER

num REMAINDER num1 num2

Outputs the remainder on dividing num1 by num2; both must be integers and the result is an integer with the same sign as num2.

num: (INTEGER) Result of remainder.

num1: (INTEGER) Dividend of remainder.

num2: (INTEGER) Divisor of remainder.

Example:

```
show remainder 6 4
2
show remainder 6 2
0
```
INT

int INT num

Outputs its input with fractional part removed, i.e., an integer with the same sign as the input, whose absolute value is the largest integer less than or equal to the absolute value of the input.

int:(INTEGER) Result of int.

num:(NUMBER) Number to int.

Note: Inside the computer numbers are represented in two different forms, one for integers and one for numbers with fractional parts. However, on most computers the largest number that can be represented in integer format is smaller than the largest integer that can be represented (even with exact precision) in floating-point (fraction) format. The INT operation will always output a number whose value is mathematically an integer, but if its input is very large the output may not be in integer format. In that case, operations like REMAINDER that requires an integer input will not accept this number.

Example:

show int 8.2
8

show int 8.7
8

* INT
* INT
* INT
* LOGO:0
# $ K • ROUND

int ROUND num

Outputs the nearest integer to the input.

int:(INT)GER Result of rounding.

num:(NUM)BER Number to round.

Example:

text

show round 8.2
8
text
text

show round 8.7
9
# ABS

num ABS num1  (library procedure)

Outputs the absolute value (magnitude) of it's input.

num: (NUMBER) Result of abs.

num1: (NUMBER) Number to take abs of.

Example:

show abs 5
5
show abs -5
5

# ABS
$ ABS
K ABS
* LOGO:0
#SIGN

int SIGN num (library procedure)

Outputs the sign (-1 if negative, 0 if zero, 1 is positive) of its input. This is handy for when you want to take the sign of one number and apply it to another.

int :(INTEGER) Result of sign of num.

num :(NUMBER) Number to take sign of.

Example:

show sign 5
1
show sign 0
0
show sign -98.6
-1
\# $ K \cdot \texttt{SQR T}

\texttt{num SQR T \ num1}

Outputs the square root of the input, which must be nonnegative.

\texttt{num}: (NUMBER) Result of square root of \texttt{num}.

\texttt{num1}: (NUMBER) Number to take sign of.

Example:

\texttt{show sqrt 4}
\texttt{2}
\texttt{show sqrt 9}
\texttt{3}
# $K • POWER

`num POWER num1 num2`

Outputs `num` raised to the power of `num2`. If `num1` is negative, then `num2` must be an integer.

`num:`(NUMBER) Result of `num1` raised to the power of `num2`.

`num1:`(NUMBER) Base to raise.
`num2:`(NUMBER) Power to raise base to.

Example:

`show power 2 3`

8
# $K • EXP

num EXP num1

Outputs e (2.718281828+) to the power of num1.

num1:(NUMBER) Result of e raised to the power of num1

num1:(NUMBER) Power to raise e to.

Example:

show exp 2
7.38905609893065
# $ K • LOG10

**num LOG10 num1**

Outputs the common logarithm of num1. That is, 10 raised to num = num1, where num will be the Output.

**num**: (NUMBER) Result of log10 of num1.

**num1**: (NUMBER) The number to take LOG10 of.

Example:

```
log10 1  0
log10 10  1
log10 100 2
```
# $K • LN

num LN num1

Outputs the natural logarithm of the num1. That is, e (2.718281828+) raised to num = num1, where num will be the Output.

num:(NUMBER) Result of log10 of num1.

num1:(NUMBER) The number to take LN of.

Example:

show ln 1
0
show ln exp 1
1
num Pi (library procedure)

Outputs the value pi (3.141592653589793227020265931059839203954)

num: (NUMBER) The value of Pi.

Example:

show radsin pi/2
1
# TAN

**num TAN angle**

Outputs the tangent of **angle**, which is taken in degrees. See also **RADTAN**.

**num**(NUMBER) The tangent of the input.

**angle**(NUMBER) Angle in degrees to take tangent of.

Example:

```plaintext
show tan 0
0
show tan 90
Error
show tan 45
1
```
# RADIANT

num RADIANT angle

Outputs the tangent of angle, which is taken in radians. See also TAN

num: (NUMBER) The tangent of the input.

angle: (NUMBER) Angle in radians to take tangent of.

Example:

show radian 0
0
show radian pi/2
ERROR
show radian pi/4
1
num SIN angle

Outputs the sine of `angle`, which is taken in degrees. See also `RADSIN`.

`num`:(NUMBER) The sine of the input.

`angle`:(NUMBER) Angle in degrees to take sine of.

Example:

```
show sin 0
 0
show sin 90
 1
show sin 180
 0
```
\* * \* RADSIN

num RADSIN angle

Outputs the sine of angle, which is taken in radians. See also \texttt{SIN}.

\texttt{num}: (NUMBER) The sine of the input.

\texttt{angle}: (NUMBER) Angle in radians to take sine of.

Example:

```plaintext
show radsin 0
  0
show radsin pi/2
  1
show radsin pi
  0
```
# $ K • COS

num COS angle

Outputs the cosine of angle, which is taken in degrees. See also RADCOS.

num: (NUMBER) The cosine of the input.

angle: (NUMBER) Angle in degrees to take cosine of.

Example:

```
show cos 0
1
show cos 90
0
show cos 180
-1
```
RADCOS

num RADCOS angle

Outputs the cosine of angle, which is taken in radians. See also COS.

num: (NUMBER) The cosine of the input.

angle: (NUMBER) Angle in radians to take cosine of.

Example:

show radcos 0
 1
show radcos pi/2
 0
show radcos pi
-1
ARCTAN

angle ARCTAN num
angle (ARCTAN x y)

With one input, outputs the arctangent angle, in degrees, of its input.
With two inputs, outputs the arctangent angle, in degrees, of y/x.
See also RADARCTAN.

angle:(NUMBER) Resulting angle in degrees.
num:(NUMBER) Number to take arctangent of.
x:(NUMBER) Delta X (Run) component to take arctangent of.
y:(NUMBER) Delta Y (Rise) component to take arctangent of.

Example:

show arctan 1
45
# $K • RADARCTAN

angle RADARCTAN num
angle (RADARCTAN x y)

With one input, outputs the arctangent angle, in radians, of its input.
With two inputs, outputs the arctangent angle, in radians, of y/x.
See also ARCTAN.

angle: (NUMBER) Resulting angle in radians.

num: (NUMBER) Number to take arctangent of.

Example:

show (radarctan 1) = pi/4
true

* RADARCTAN
+ RADARCTAN
\* RADARCTAN
* LOGO:0
angle ARCSIN num

Outputs the arcsine angle, in degrees, of its input. See also RADARCSIN.

angle:(NUMBER) Resulting angle in degrees.

num:(NUMBER) Number to take arcsine of.

Example:

show arccsin 0.5
30
show arccsin sin 45
45
RADARCSIN

angle RADARCSIN num

Outputs the arcsine angle, in radians, of its input. See also ARCSIN.

angle : (NUMBER) Resulting angle in radians.

num : (NUMBER) Number to take arcsine of.

Example:

show (radarcsin 1) = pi/2
true
# ARCCOS

angle ARCCOS num

Outputs the arccosine, in degrees, of its input. See also RADARCCOS.

Example:

```
show arccos 0.5
60
show arccos cos 45
45
```
# \* RADARCCOS

angle RADARCCOS num

Outputs the arccosine, in radians, of its input. See also ARCCOS.

angle:(NUMBER) Resulting angle in radians.

num:(NUMBER) Number to take arccosine of.

Example:

show (radarccos (sqrt 2)/2) = pi/4
true
PREDICATES (Arithmetic)

LESSP
GREATERP
## LESSP

**truth LESSP num1 num2**

`num1 < num2`

Outputs TRUE if its first input is strictly less than its second.

**truth**: (BOOLEAN) Truth of the test for being less.

**num1**: (NUMBER) Number to be tested for being less than num2.

**num2**: (NUMBER) Number to be tested for being greater than num1.

**Example:**

```
show 1 < 2
true
show 2 < 1
false
```
# GREATERP

truth GREATERP num1 num2
num1 > num2

Outputs TRUE if its first input is strictly greater than its second.

truth: (BOOLEAN) Truth of the test for being greater.

num1: (NUMBER) Number to be tested for being greater than num2.
num2: (NUMBER) Number to be tested for being less than num1.

Example:

show 2 > 1
true
show 1 > 2
false
RANDOM NUMBERS

RANDOM
RERANDOM
# $K • RANDOM

```
num RANDOM num1
```

Outputs a random nonnegative integer less than its input, which must be an integer.

- **num**: (INTEGER) Random number generated.
- **num1**: (INTEGER) Range for random number to be generated.

Example:

```
repeat 5 [show random 10]
6
8
3
0
9
```
RERANDOM

(RERANDOM seed)

Command that makes the results of RANDOM reproducible. Ordinarily the sequence of random numbers is different each time Logo is used. If you need the same sequence of pseudo-random numbers repeatedly, e.g. to debug a program, say RERANDOM before the first invocation of RANDOM. If you need more than one repeatable sequence, you can give RERANDOM an integer input; each possible input selects a unique sequence of numbers.

seed:(INTEGER) Seed for random number generator.

Example:

rerandom 1234
repeat 2 [show random 10]
 6
 2
rerandom 1234
repeat 2 [show random 10]
 6
 2
# $K$ *PRINT FORMATTING*

FORM
word FORM num width precision

Outputs a word containing a printable representation of num, possibly preceded by spaces (and therefore not a number for purposes of performing arithmetic operations), with at least width characters, including exactly precision digits after the decimal point. (If "precision" is 0 then there will be no decimal point in the output.)

word: (WORD) Formatted number.
num: (NUMBER) Number to be formatted.
width: (INTEGER) Minimum overall width including decimal point.
precision: (INTEGER) Number digits to print after decimal point.

Example:

show form 123.1 10 10
123.1000000000

As a debugging feature, (FORM num -1 format) will print the floating point "num" according to the C printf "format", to allow

to hex :num
op form :num -1 "\%08X \%08X"
end

to allow finding out the exact result of floating point operations. The precise format needed may be machine-dependent.

Example:

show form 99.99 -1 "\%08X \%08X"
0000C28F 000028F5

* FORM
* FORM
* FORM
* LOGO:0
BITWISE OPERATIONS

BITAND
BITOR
BITXOR
BITNOT
ASHIFT
LSHIFT
# BITAND

```
num BITAND num1 num2
num (BITAND num1 num2 num3 ...)
```

Outputs the bitwise AND of its inputs, which must be integers.

```
num:(INTEGER) ANDed result.
num1:(INTEGER) First number to be ANDed.
num2:(INTEGER) Second number to be ANDed.
```

Example:

```
show bitand 5 2
0
show bitand 5 1
1
```
num BITOR num1 num2
num (BITOR num1 num2 num3 ...)

Outputs the bitwise OR of its inputs, which must be integers.

num:(INTEGER) ORed result.

num1:(INTEGER) First number to be ORed.
num2:(INTEGER) Second number to be ORed.

Example:

show bitor 5 2
7
show bitor 5 1
5
BITXOR

num BITXOR num1 num2
num (BITXOR num1 num2 num3 ...)

Outputs the bitwise EXCLUSIVE OR of its inputs, which must be integers.

num:(INTEGER) XORed result.
num1:(INTEGER) First number to be XORed.
num2:(INTEGER) Second number to be XORed.

Example:

```
show bitxor 5 2
7
show bitxor 5 1
4
```
# $ K - BITNOT

```
num BITNOT num1

Outputs the bitwise NOT of its input, which must be an integer.

num: (INTEGER) NOTed result.

num1: (INTEGER) Number to be NOTed.

Example:

show bitnot 1
-2
show bitnot 5
-6
```
\textbf{ASHIFT}

\texttt{num ASHIFT num1 num2}

Outputs \texttt{num1} arithmetic-shifted to the left by \texttt{num2} bits. If \texttt{num2} is negative, the shift is to the right with sign extension. The inputs must be integers.

\texttt{num:}(\texttt{INTEGER}) Arithmetic shifted result.
\texttt{num1:}(\texttt{INTEGER}) The amount to shift by.
\texttt{num2:}(\texttt{INTEGER}) The number to be shifted.

Example:

\begin{verbatim}
show ashift 5 2
20
show ashift 20 -1
10
\end{verbatim}
# $K \textbf{LSHIFT}$

$\text{num LSHIFT num1 num2}$

Outputs $\text{num1}$ logical-shifted to the left by $\text{num2}$ bits. If $\text{num2}$ is negative, the shift is to the right with zero fill. The inputs must be integers.

$\text{num:}(\text{INTEGER})$ Logical shifted result.

$\text{num1:}(\text{INTEGER})$ The amount to shift by.

$\text{num2:}(\text{INTEGER})$ The number to be shifted.

Example:

```lisp
(show lshift 5 2
  20
(show lshift 20 -1
  10
```

---

* LSHIFT
  * LSHIFT
  * LSHIFT
  * LOGO:0
LOGICAL OPERATIONS

AND
OR
NOT
`AND`

truth AND t1 t2
truth (AND t1 t2 t3 ...)

Outputs TRUE if all inputs are TRUE, otherwise FALSE. All inputs must be TRUE or FALSE. (Comparison is case-insensitive regardless of the value of `CASEIGNOREDP`.

That is, "true" or "True" or "TRUE" are all the same.

truth: (BOOLEAN) Truth of ANDed inputs.

tf1: (BOOLEAN) TRUE or FALSE expression.
tf2: (BOOLEAN) TRUE or FALSE expression.

Example:

```
show and "true "false
false
show and "true "true
true
```

---

`AND`

`AND`

`AND`

`AND`

`LOGO:0`
OR

truth OR tf1 tf2
truth (OR tf1 tf2 tf3 ...)

Outputs TRUE if any input is TRUE, otherwise FALSE. All inputs must be TRUE or FALSE. (Comparison is case-insensitive regardless of the value of CASEIGNOREDP. That is, "true" or "True" or "TRUE" are all the same.)

truth: (BOOLEAN) Truth of ORed inputs.
tf1: (BOOLEAN) TRUE or FALSE expression.
tf2: (BOOLEAN) TRUE or FALSE expression.

Example:

show or "true "false
true
show or "false "false
false
# $K • NOT

truth NOT tf

Outputs TRUE if the input is FALSE, and vice versa.

truth:(BOOLEAN) Truth of NOTed input.

tf1:(BOOLEAN) TRUE or FALSE expression.

Example:

not "true
false
MSWLogo (Berkeley) Logo provides traditional and extended Logo turtle graphics. Multiple turtles are supported but dynamic turtles, and collision detection are not supported.

The center of the graphics window is turtle location [0 0]. Positive X is to the right; positive Y is up. Headings (angles) are measured in degrees clockwise from the positive Y axis (12:00). This differs from the common mathematical convention of measuring angles counterclockwise from the positive X axis (3:00). The turtle is represented as an isosceles triangle; the actual turtle position is at the midpoint of the base (the long side).

MSWLogo does take advantage of the hardware and therefore is not completely portable with other ports of ucblogo (Berkeley Logo).

**MULTIPLE TURTLES**
**TURTLE MOTION**
**TURTLE MOTION QUERIES**
**TURTLE AND WINDOW CONTROL**
**USING COLOR**
**TURTLE AND WINDOW QUERIES**
**PEN CONTROL**
**PEN QUERIES**
**MULTIPLE TURTLES**

*SETTURTLE*
*TURTLE*
*BITMAPTURTLE*
*NOBITMAPTURTLE*

# $K$ #

* MULTIPLE_TURTLES
* MULTIPLE_TURTLES
* MULTIPLE_TURTLES
* MULTIPLE_TURTLES
* LOGO:0
SETTURTLE turtle

Selects turtle turtle for control. Each turtle maintains its own Heading (orientation in 3D), Position, and Pen Control, but not pen color. Each turtle can be represented as a BITMAPTURTLE. Note that turtles start at 0 which is the default turtle. There is a limit of 1024 turtles meaning the largest value turtle can be is 1023 (since we started at 0 and not 1). All turtles between 0 and the highest turtle ever selected will be "activate". So don't choose turtle 100 and 200 if you only need 2 turtles. A reset or CLEARSCREEN will automatically select turtle 0 and deactivate all other turtles.

turtle:(INTEGER) Turtle to be selected.

Example:

repeat 8 [setturtle repcount-1 fd 10*repcount]

There are also 3 special turtles used when in PERSPECTIVE mode.

Turtle -1: His position represents where your eye is when viewing a 3D scene (defaults to [400 400 600]). His Orientation represents which direction is UP on the screen (defaults to 0).

Turtle -2: Her position represents where your eye is looking when viewing a 3D scene (defaults to [0 0 0]). Her Orientation does not matter.

Turtle -3: Its position represents where the light source is positioned (defaults to [0 0 1000]). Its Orientation does not matter. This turtle only matters when you're using POLYSTART and POLYEND.

These special turtles are never shown or bitmapped. Each time you enter PERSPECTIVE mode all special turtles are reset to their default position and orientation.

Example 2 (3D):

```
perspective
  cs
  ; Draw a cube from the default vantage point
  repeat 4 [repeat 4 [fd 100 rt 90] fd 100 down 90]  
  cs
  setturtle -1
  setxyz 500 500 500
  setturtle 0
  ; Draw a cube from a new vantage point
  repeat 4 [repeat 4 [fd 100 rt 90] fd 100 down 90]
```

* SETTURTLE
* $ SETTURTLE
* $ SETTURTLE
* $ LOGO:0
turtle TURTLE

Outputs the current turtle under control. To change the turtle under control see SETTURTLE.

Example:

setturtle 8
show turtle
8
Maps the current turtle (See SETTURTLE) to the corresponding Bitmap buffer (See BITCUT). Since Bitmap buffer 0 is mapped to the clipboard then turtle 0 (if mapped to a bitmap with this command) will be mapped to the clipboard. That is, if you select turtle 0 and issue BITMAPTURTLE then the turtle IS the clipboard. Note the clipboard must have a bitmap in it. Try drawing a picture in Paint and copy a section to the clipboard. Then move to MSWLogo and select turtle 0 (the default) and issue this command. Both NOBITMAPTURTLE and CLEARSCREEN will restore the turtle to its normal shape (it does not erase the corresponding BITMAP).

Example:

```
cs
pu
label 1
bk 20
pd
bitcut 20 20
bitmapturtle
fd 100
```
**NOBITMAPTURTLE**

This is the opposite of **BITMAPTURTLE**, it removes mapping of the current turtle from the corresponding Bitmap buffer command.

**Example:**

```plaintext
cs
pu
label 1
bk 20
pd
bitcut 20 20
bitmapturtle
fd 100
nobitmapturtle
```
# $K• TURTLE MOTION

**FORWARD**
**BACK**
**LEFT**
**RIGHT**
**LEFTROLL**
**RIGHTROLL**
**DOWNPITCH**
**UPPITCH**
**SETPOS**
**SETPOSXYZ**
**SETXY**
**SETXYZ**
**SETX**
**SETY**
**SETZ**
**HOME**
**SETHEADING**
**SETROLL**
**SETPITCH**
**SETORIENTATION**
**ELLIPSEARC**
**ELLIPSE**
**ARC**
**CIRCLE**
# $K \bullet \text{FORWARD}

FORWARD dist
FD dist

Moves the turtle forward, in the direction that it's headed, by the specified distance (measured in turtle steps).

dist:(NUMBER) Distance for turtle to travel.

Example:

repeat 4 [forward 100 rt 90]
BACK

BACK dist
BK dist

Moves the turtle backward, i.e., exactly opposite to the direction that it's headed, by the specified distance. (The heading of the turtle does not change.)

dist:(NUMBER) Distance for turtle to travel.

Example:

repeat 4 [back 100 rt 90]
**LEFT**

**LEFT angle**

**LT angle**

Turns the turtle counterclockwise by the specified **angle**, measured in degrees (1/360 of a circle).

**angle**: (NUMBER) Angle, in degrees, for turtle to turn by.

Example:

```
repeat 3 [fd 100 left 120]
```
RIGHT

RIGHT angle
RT angle

Turns the turtle clockwise by the specified angle, measured in degrees (1/360 of a circle).

angle: (NUMBER) Angle, in degrees, for turtle to turn by.

Example:

repeat 3 [fd 100 right 120]
LEFTROLL angle
LR angle

Rolls the turtle (on to his left side) by the specified angle, measured in degrees (1/360 of a circle). This command is designed to run in PERSPECTIVE mode.

angle: (NUMBER) Angle, in degrees, for turtle to roll by.

Example:

g perspective
g leftroll 45
g repeat 3 [fd 100 left 120]
RIGHTROLL angle
RR angle

Rolls the turtle (on its right side) by the specified angle, measured in degrees (1/360 of a circle). This command is designed to run in PERSPECTIVE mode.

angle: (NUMBER) Angle, in degrees, for turtle to roll by.

Example:

perspective
rightroll 45
repeat 3 [fd 100 right 120]
**#SK* DOWNPITCH**

**DOWNPITCH angle**

**DOWN angle**

Pitches the turtles nose downward by the specified angle, measured in degrees (1/360 of a circle). This command is designed to run in PERSPECTIVE mode.

angle (NUMBER) Angle, in degrees, for turtle to pitch by.

Example:

```plaintext
perspective
downpitch 45
repeat 3 [fd 100 right 120]
```
UPPITCH

UPPITCH angle
UP angle

Pitches the turtle nose up by the specified angle, measured in degrees (1/360 of a circle). This command is designed to run in PERSPECTIVE mode.

angle: (NUMBER) Angle, in degrees, for turtle to pitch by.

Example:

perspective
uppitch 45
repeat 3 [fd 100 right 120]
**SETPOS**

**SETPOS pos**

Moves the turtle to an absolute X,Y coordinate. The argument is a list of two numbers, the X and Y coordinates. See also **POS**.

**pos**: (LIST) List of two numbers representing desired X,Y coordinate.

**Example 1**: (draw a square)

```console
cs
setpos [0 100]
setpos [100 100]
setpos [100 0]
setpos [0 0]
```

**Example 2**: (the most common logo question)

```console
make "x 0
make "y 100
setpos [:x :y]
<will fail>
setpos (list :x :y)
<will work>
```

Why, because the first case IS a list that contains 2 words :x and :y. In the second case a list is BUILT containing the value of :x and :y. You can see this more clearly by using the **SHOW** command.

```console
show [:x :y]
[:x :y]
show (list :x :y)
[0 100]
```
# $K * SETPOXYZ

SETPOXYZ pos

Moves the turtle to an absolute X,Y,Z coordinate. The argument is a list of three numbers, the X, Y and Z coordinates. This command is designed to run in PERSPECTIVE mode. See also POSXYZ

pos:(LIST) List of three numbers representing desired X,Y,Z coordinate.

Example:(draw a cube)

```
perspective
; Now draw a cube
setposxyz [0 100 0]
setposxyz [100 100 0]
setposxyz [100 0 0]
setposxyz [0 0 0]
setposxyz [0 0 100]
setposxyz [100 0 100]
setposxyz [100 100 100]
setposxyz [0 100 100]
setposxyz [0 0 100]
setposxyz [0 100 100]
setposxyz [0 100 0]
setposxyz [100 100 0]
setposxyz [100 100 100]
setposxyz [100 0 100]
setposxyz [100 0 0]
```
# $ K • SETXY

SETXY xcor ycor

Moves the turtle to an absolute X,Y coordinate. The two arguments are numbers, the X and Y coordinates. See also POS.

xcor:(NUMBER) The desired X coordinate.
ycor:(NUMBER) The desired Y coordinate.

Example (draw a sine wave):

```
repeat 360 [setxy repcount 100*sin repcount]
```
SETXYZ xcor ycor zcor

Moves the turtle to an absolute 3D position. The three arguments are numbers, the X, Y and Z coordinates. This command is designed to run in PERSPECTIVE mode. See also POSXYZ.

xcor:(NUMBER) The desired X coordinate.
ycor:(NUMBER) The desired Y coordinate.
zcor:(NUMBER) The desired Y coordinate.

Example:(draw some sine waves in 3D)

perspective
for [i 0 360 10] ~
  [for [j 0 360] [setxyz :j :i*sin :j -:i]
   pu
   setxyz 0 0 -:i
   pd
  ]
SETX

SETX xcor

Moves the turtle along the X axis from its current position to a new absolute X coordinate. The argument is the new X coordinate. See also SETY and SETZ.

xcor: (NUMBER) The desired X coordinate.

Example:

setx 100
sety 100
setx 0
sety 0

* SETX
$ SETX
% SETX
* LOGO:0
# $ \cdot \text{SETY}

\text{SETY ycor}

Moves the turtle along the Y axis from its current position to a new absolute Y coordinate. The argument is the new Y coordinate. See also \text{SETX} and \text{SETZ}.

ycor:(NUMBER) The desired Y coordinate.

Example:

\begin{verbatim}
setx 100
sety 100
setx 0
sety 0
\end{verbatim}
SETZ zcor

Moves the turtle along the Z axis from its current position to a new absolute Z coordinate. The argument is the new Z coordinate. This command is designed to run in PERSPECTIVE mode. See also SETX and SETY.

zcor:(NUMBER) The desired Z coordinate.

Example:

```plaintext
perspective
pu setx 100 pd
sez 100
sety 100
setz 0
sety 0
```
HOME

HOME

Moves the turtle to the center of the screen. Equivalent to \texttt{SETPOS [0 0]}.

Example:

\texttt{setxy 100 100}
\texttt{home}
SETHEADING

SETHEADING angle
SETH angle

Turns the turtle to a new absolute heading. The argument is an angle, the heading in degrees clockwise from the positive Y axis. See also HEADING. If you are in PERSPECTIVE mode then, the heading in degrees which is positive from the positive X-Axis to the positive Y-Axis rotating about the Z-Axis.

angle:(NUMBER) Angle, in degrees, to set heading to.

Example:

setheading 45
show heading
45
# SK* SETROLL

SETROLL angle

Rolls the turtle to a new absolute roll. The argument is a `angle`, the roll in degrees which is positive from the positive X-Axis to the negative Z-Axis rotating about the Y-Axis. It is important to understand your Orientation in 3D. This command is designed to run in PERSPECTIVE mode. See also ROLL.

`angle` (NUMBER) Angle, in degrees, to set roll to.

Example:

```
perspective
setroll 45
show roll
45
```
# SETPITCH

SETPITCH angle

Pitches the turtle to a new absolute pitch. The argument is a angle, the pitch in degrees which is positive from the negative Z-Axis to the position Y-Axis rotating about the X-Axis. It is important to Understand your Orientation in 3D. This command is designed to run in PERSPECTIVE mode. See also PITCH.

angle: (NUMBER) Angle, in degrees, to set pitch to.

Example:

```plaintext
perspective
setpitch 45
show pitch
45
```
SETORIENTATION

SETORIENTATION list(library procedure)

Orients the turtle to a new absolute orientation. The argument is a list, the [roll pitch heading] in degrees. It is important to Understand your Orientation in 3D. This command is designed to run in PERSPECTIVE mode. See also ORIENTATION command.

list:(LIST) A list of 3 angles, [roll pitch heading] in degrees, to set orientation to.

Example:

setorientation [180 45 90]
show orientation
[180 45 90]
ELLIPSEARC

ELLIPSEARC angle minor major startangle
ELLIPSA2 angle minor major startangle(library procedure)

ELLIPSEARC does not move the turtle. It draws part of or all of an ellipse based on the turtle heading, turtle position and the given arguments. The ellipse starts at the rear of the turtle heading and sweeps by the amount of angle starting at startangle. The size is based on the minor and major values. The current turtle position will be at the center of the ellipse. ELLIPSEARC will also follow wrap/fence/windows modes. Minor is the shortest distance along the ellipse to the center point. Major is the longest distance along the ellipses the center point. ELLIPSA2 starts the ellipse on the edge rather than from the center and leaves you on the edge where it finishes.

angle (NUMBER) Angle, in degrees, to sweep.
minor:(NUMBER) Minimum distance between the center and the ellipse.
major:(NUMBER) Maximum distance between the center and the ellipse.
startangle:(NUMBER) Angle, in degrees, to start the sweep.

Example:

ellipsearc 360 100 200 0
cs
ellipsearc 90 50 50 0
cs
ellipsearc 90 50 50 90

Example (a barrel? in 3D):

perspective
repeat 72 [ellipsearc 90 100 150 45 rr 5]
ELLIPSE

ELLIPSE minor major  (library procedure)
ELLIPSE2 minor major (library procedure)

ELLIPSE does not move the turtle. It draws an ellipse based on the turtle heading, turtle position and the given arguments. The size is based on the \textit{minor} and \textit{major} values. The current turtle position will be at the center of the ellipse. \texttt{ELLIPSE} will also follow \texttt{wrap}/\texttt{fence}/\texttt{windows} modes. \texttt{Minor} is the shortest distance along the ellipse to the center point. \texttt{Major} is the longest distance along the ellipses the center point. \texttt{ELLIPSE2} starts the ellipse on the edge rather than from the center.

\texttt{minor}: \text{(NUMBER)} Minimum distance between the center and the ellipse.
\texttt{major}: \text{(NUMBER)} Maximum distance between the center and the ellipse.

Example:

\texttt{ellipse 100 200}
\texttt{cs}
\texttt{ellipse 50 50}
\texttt{cs}
\texttt{ellipse 50 50}

Example (towards in 3D):

\texttt{perspective}
\texttt{cs}
\texttt{repeat 72 [ellipse 200 100 rr 5 fd 5]}

# ELLIPSE
$ ELLIPSE
k ELLIPSE
* LOGO:0
# $ K • ARC

ARC angle radius (library procedure)
ARC2 angle radius (library procedure)

ARC does not move the turtle. It draws an arc (part of a circle) based on the turtle heading, turtle position and the given arguments. The arc starts at the rear of the turtle heading and sweeps by the amount of angle. The size is based on the radius. The current turtle position will be at the center of the arc. Arc will also follow wrap/fence/windows modes. ARC 360 radius will of course draw a circle. ARC2 starts the arc on the edge rather than from the center and leaves you on the edge where it finishes.

angle (NUMBER) Angle, in degrees, to sweep.
radius (NUMBER) The distance between the center and the arc.

Example:

arc 360 100
arc 90 50

Example (Water fountain in 3D):

perspective
repeat 36 [r 10 arc2 180 100 arc2 -180 100]
**CIRCLE**

CIRCLE radius (library procedure)
CIRCLE2 radius (library procedure)

CIRCLE does not move the turtle. It draws a circle based on the turtle's position and the given arguments. The size is based on the radius. The current turtle position will be at the center of the circle. CIRCLE will also follow wrap/fence/windows modes. CIRCLE2 starts the circle on the edge rather than from the center.

Example:

circle 100
circle 50

Example (a sphere in 3D):

perspective
repeat 72 [circle 100 n 5]
TURTLE MOTION QUERIES

POS
POSXYZ
XCOR
YCOR
ZCOR
HEADING
ROLL
PITCH
ORIENTATION
TOWARDS
TOWARDSXYZ
DISTANCE
DISTANCEXYZ
PIXEL
SCRUNCH
# $ K • POS

pos POS

Outputs the turtle's current position, as a list of two numbers, the X and Y coordinates.

pos: (LIST) List of two numbers representing current X,Y coordinate.

Example:

setpos [100 100]
show pos
[100 100]
```plaintext
# SK* POSXYZ

pos POSXYZ

Outputs the turtle's current position, as a list of three numbers, the X, Y and Z coordinates. This command is designed to run in PERSPECTIVE mode.

pos:LIST) List of three numbers representing current X,Y,Z coordinate.

Example:

    perspective
    setposxyz [100 100 50]
    show posxyz
    [100 100 50]
```

```
# POSXYZ
$ POSXYZ
^ POSXYZ
* LOGO:0
```
# $ K • XCOR

```
xcoR XCOR  (library procedure)

Outputs a number, the turtle's X coordinate.

xcor:(NUMBER) The current X coordinate.

Example:
```
setx 100
show xcor
100
```
$ YCOR$

YCOR  (library procedure)

Outputs a number, the turtle's Y coordinate.

ycor:(NUMBER) The current Y coordinate.

Example:

sety 100
show ycor
100
ZCOR (library procedure)

Outputs a number, the turtle's Z coordinate. This command is designed to run in PERSPECTIVE mode.

zcor:(NUMBER) The current Z coordinate.

Example:

perspective
setz 100
show zcor
100
angle HEADING

Outputs an angle, the turtle’s heading in degrees. See also SETHEADING.

angle : (NUMBER) Angle, in degrees, of the current heading.

Example:

setheading 90
show heading
90
# ROLL

angle ROLL

Outputs an angle, the turtle's roll in degrees. It is important to understand your Orientation in 3D. See also SETROLL. This command is designed to run in PERSPECTIVE mode.

angle: (NUMBER) Angle, in degrees, of the current roll.

Example:

```
setroll 90
show roll
90
```
angle PITCH

Outputs a angle, the turtle's pitch in degrees. It is important to Understand your Orientation in 3D. See also SETPITCH. This command is designed to run in PERSPECTIVE mode.

angle (NUMBER) Angle, in degrees, of the current pitch.

Example:

```
setpitch 90
show pitch
90
```
**ORIENTATION**

```
list ORIENTATION(library procedure)
```

Outputs a list, the turtle’s [roll pitch heading] each in degrees. It is important to understand your Orientation in 3D. See also SETORIENTATION. This command is designed to run in PERSPECTIVE mode.

`list:(LIST)` A list of 3 angles, [roll pitch heading] in degrees, of the current orientation.

Example:

```
setorientation [180 45 90]
show orientation
[180 45 90]
```
TOWARDS

angle TOWARDS pos

Outputs an angle, the heading in degrees, at which the turtle should be headed so that it would point from its current position towards the position given as the argument.

angle : (NUMBER) Angle, in degrees, to be towards pos.

pos : (LIST) List of two numbers representing desired X,Y coordinate to be towards.

Example:

show towards [100 100]
45
setheading towards [300 400] fd distance [300 400]
list TOWARDSXYZ pos

Outputs a list, containing [roll pitch heading] at which the turtle should be oriented so that it would point from its current position to the position given as the [x y z] argument. It is important to understand your Orientation in 3D. This command is designed to run in PERSPECTIVE mode.

list:(LIST) A list of 3 angles (orientation), [roll pitch heading] in degrees, to be towards pos.

pos:(LIST) List of three numbers representing desired X,Y,Z coordinate to be towards.

Example:

show towardsxyz [100 100 0]
[0 0 45]
setorientation towardsxyz [100 100 100] fd distancexyz [100 100 100]
# DISTANCE

`dist DISTANCE pos` (library procedure)

Outputs a number, the distance the turtle must travel along a straight line to reach the position given as the argument.

`dist:(NUMBER)` Distance, in turtle pixels, to `pos`.

`pos:(LIST)` List of two numbers representing desired X,Y coordinate to find distance to.

Example:

```
show towards [0 100]
100
show towards [300 400]
500
setheading towards [300 400] fd distance [300 400]
```
# *DISTANCEXYZ*

**dist** DISTANCEXYZ pos  (library procedure)

Outputs a number, the distance the turtle must travel along a straight line to reach the xyz position given as the argument. This command is designed to run in **PERSPECTIVE** mode. This command is designed to run in **PERSPECTIVE** mode.

**dist:**(NUMBER) Distance, in turtle pixels, to pos.

pos: (LIST) List of three numbers representing desired X,Y,Z coordinate to find distance to.

Example:

```
show towardsxyz [0 100 0]
100
show towardsxyz [100 100 100]
173.205080756888
setorientation towardsxyz [100 100 100] fd distancexyz [100 100 100]
```
# $ K * PIXEL

colorvector PIXEL

Outputs a list of numbers that represent the Red, Green, and Blue intensity of the pixel currently under the turtle. For an explanation of colorvector see USING COLOR. See also SETPIXEL.

colorvector:(LIST) List of three integers representing Red, Green, Blue intensities each in the range 0-255 that the turtle is over.

Example:

    show pixel
    [255 255 255]
    fd 1
    show pixel
    [255 255 255]
    bk 1
    show pixel
    [0 0 0]

Pixel can be used in PERSPECTIVE mode but the pixel seen will be what ever is in the 2D view.
SCRUNCH

ratio SCRUNCH

Outputs a list containing two numbers, the X and Y scrunch factors, as used by SETSCRUNCH. (But note that SETSCRUNCH takes two numbers as inputs, not one list of numbers.)

ratio: (LIST) List of two numbers representing the current X and Y scrunch factors.

Example:

setscrunch 2.0 1.0
repeat 4 [fd 100 rt 90]
show scrunch
[2 1]
SHOWTURTLE
HIDETURTLE
CLEAN
CLEARSCREEN
WRAP
WINDOW
FENCE
FILL
LABEL
SETPIXEL
SETLABELFONT
LABELFONT
LABELSIZE
TEXTSCREEN
FULLSCREEN
SPLITSCREEN
SETSCRUNCH
REFRESH
NOREFRESH
ZOOM
SCROLLX
SCROLLY
SETFOCUS
GETFOCUS
ICON
UNICON
WINDOWSET
SHOWTURTLE

SHOWTURTLE
ST

Makes the turtle visible.

Example:

hideturtle
showturtle
HIDETURTLE

HIDETURTLE
HT

Makes the turtle invisible. It's a good idea to do this while you're in the middle of a complicated drawing, because hiding the turtle speeds up the drawing substantially.

Example:

hideturtle
showturtle
CLEAN

Erases all lines that the turtle has drawn on the graphics window. The turtle's state (position, heading, pen mode, etc.) is not changed.

Example:

setxy 100 100
clean
CLEARSCREEN

Erases the graphics window and sends the turtle to its initial position and heading. Like HOME and CLEAN together.

Example:

setxy 100 100
clearscreen
WRAP

WRAP

Tells the turtle to enter wrap mode: From now on, if the turtle is asked to move past the boundary of the graphics window, it will "wrap around" and reappear at the opposite edge of the window. The top edge wraps to the bottom edge, while the left edge wraps to the right edge. (So the window is topologically equivalent to a torus.) This is the turtle's initial mode. Compare WINDOW and FENCE.

Example:

window
fd 950
cs
wrap
fd 950
WINDOW

Tells the turtle to enter window mode: From now on, if the turtle is asked to move past the boundary of the graphics window, it will move off screen. The visible graphics window is considered as just part of an infinite graphics plane; the turtle can be anywhere on the plane. (If you lose the turtle, HOME will bring it back to the center of the window.) Compare WRAP and FENCE.

Example:

```plaintext
window
fd 950
cs
wrap
fd 950
```
FENCE

Tells the turtle to enter fence mode: From now on, if the turtle is asked to move past the boundary of the graphics window, it will move as far as it can and then stop at the edge with an "out of bounds" error message. Compare \texttt{WRAP} and \texttt{WINDOW}.

Example:

\begin{verbatim}
window
fd 950
cs
fence
fd 950
turtle out of bounds
\end{verbatim}
### Perspective

**Perspective**

Tells the turtle to enter perspective mode; it adds a third dimension (Z axis) for the turtle to live in. Perspective is a term to describe how we will view a 3-dimensional world on a 2-dimensional surface. You should be familiar with using MSWLogo in 2D before jumping into 3D.

Everything you have learned in 2D applies in 3D (perspective mode). For example if you had code that drew a 2D Picture, say a star, you can run the same code and place that star on six sides of a cube by getting the turtle to the correct 3D coordinate and orientation before you call your 2D code for drawing a star. You get the turtle there like you fly an airplane.

Airplanes have 3 controls to manipulate them through 3D space. Elevators on the tail (pitch), a rudder on the fin (yaw) and ailerons on the wings (roll). MSWLogo has always had a rudder (Right/Left) also known as Yaw. You’ve been flying already on just 1 geometric plane. But now MSWLogo offers the additional controls, ailerons (RightRoll/LeftRoll) and elevators (UpPitch/DownPitch) to maneuver 3D space. Now you have an infinite number of planes (geometric ones, that is) to fly on and each can be at any orientation. Once you’re on the desired plane you can use familiar 2D commands (Forward, Back, Left, Right) to traverse it.

When you start at (X=0,Y=0,Z=0) the +Y axis is still up the screen and the +X axis is to the right of the screen. But now you can go into the screen (away from you) or out of the screen (closer to you). How, just like an airplane flies. When an airplane Rolls to the right it does not change it’s trajectory, does it, neither does MSWLogo. But if you rolled 90 degrees to the right and then take a right turn guess what will happen. You’ll go down real fast, straight down. Think of a real turtle on the floor. Now roll him on to his right side (don’t hurt him now if he doesn’t like it). Now if he takes a right turn towards his right front leg (like he always does) he will want to go into the floor. Let him take the turn and notice how his nose is pointing at the floor. Forward will now go down in the Z axis (away from you).

Also read the section [Understand your Orientation in 3D](#).

There is a simple 3D Intro example called 3DSTEPS.LGO in the Examples directory. Load it, run it and understand it thoroughly before moving on.

Here is a summary of the relationship between commands in 2D mode and 3D mode. Many commands are repeated in both columns because they are effected by being in 3D mode.

<table>
<thead>
<tr>
<th>2D</th>
<th>3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>setx</td>
<td>setx (z stays current)</td>
</tr>
<tr>
<td></td>
<td>setyz</td>
</tr>
<tr>
<td>sety (x stays current)</td>
<td>sebx (y and z stay current)</td>
</tr>
<tr>
<td>setz</td>
<td>sety (x and z stay current)</td>
</tr>
<tr>
<td></td>
<td>setz</td>
</tr>
<tr>
<td>xcor</td>
<td>xcor</td>
</tr>
<tr>
<td>ycor</td>
<td>ycor</td>
</tr>
<tr>
<td>zcor</td>
<td>zcor</td>
</tr>
<tr>
<td>pos</td>
<td>pos (z dropped)</td>
</tr>
<tr>
<td></td>
<td>posxyz</td>
</tr>
<tr>
<td>setpos</td>
<td>setpos (z stays current)</td>
</tr>
<tr>
<td></td>
<td>setposxyz</td>
</tr>
<tr>
<td>towards</td>
<td>towards</td>
</tr>
<tr>
<td></td>
<td>towardsxyz</td>
</tr>
<tr>
<td>distance (new)</td>
<td>distance</td>
</tr>
<tr>
<td></td>
<td>distancexyz</td>
</tr>
</tbody>
</table>
heading
roll
pitch
orientation
setheading
setroll
setpitch
setorientation
right (rt)
right (rt) (on current plane)
rightroll (rr)
uppitch (up)
left (lt)
left (lt) (on current plane)
leftroll (lr)
downpitch (down)
window
perspective
wrap
fence
setturtle 0-1023
setturtle -1 (is your eye view)
setturtle -2 (is where your looking)
ellipsearc
ellipsearc (great for spheres)

Limitations:

There are several commands which will work in 2D but still behave in a 2D fashion. There action will take place at the correct X, Y, Z coordinate. But, the orientation will be limited to a 2D plane. For example you can not draw text with LABEL on 6 sides of cube. But if you wanted to LABEL the 6 vertices of the cube each label can be placed correctly at each vertex using 3D maneuvers. There is nothing to prevent you from building a "3D Block letter" alphabet library made up of 3D vectors though.

LABEL
BITCUT
BITPASTE
BITCOPY
BITBLOCK
FILL
PIXEL
SETPIXEL (new)

Perspective mode does not do hidden line removal. Hidden line removal requires objects be compose of polygons. Perspective mode does not allow you to dynamically move the eye view for (fly through's etc.). This is because MSWLogo does not store your object (your vectors) in 3D for performance reasons. It stores them in a 2D bitmap, as it always has. However you can do "fast" dynamic manipulation in 2 other ways.

One simple way is to keep your scene very simple (relative to the performance of your computer). You simply clear the screen and change the eye view (turtle -1) or the placement of the objects and draw again. Note curved objects (like those based on ellipsearc) generate many vectors. The other option is to record each "scene" in a bitmap and play them back at any rate you wish. This method is shown in the GROW.LGO example. It does not matter whether you're "recording" 2Dscenes or 3Dscenes it's still the same. This will be limited by how much memory your computer has.
FILL

Fills in a region of the graphics window containing the turtle and bounded by lines that have been drawn earlier. This is not portable; it doesn't work for all machines, and may not work exactly the same way on different machines.

Examples:

```
repeat 4 [fd 100 rt 90]
rt 45
pu
fd 20
fill
```

Fill can be used in PERSPECTIVE mode as long as the area to filled is bounded and you only fill what's in plain view. For example if you have a cube don't draw all 6 faces, only 3 at most can be in view, so draw just those faces and fill them.

Example (3D):

```
to square :color
repeat 4 [fd 100 rt 90]
pu
rt 45
fd 50
setfc :color
fill
bk 50
lt 45
pd
end

perspective
rr 45
down 45
square [255 0 0]
rr 90
square [0 255 0]
up 90
square [0 0 255]
```
LABEL text

The input, which may be a word or a list, is printed on the screen. If the object is a list, any sub-lists are delimited by square brackets, but the entire object is not delimited by brackets. You can print any logo object (numbers, lists and strings). Note that the handle of the string (the origin) is the top-left corner of the string. Another thing to be aware of is that the capabilities of the text changes depending on the device (screen or printer), the size, the turtle heading (direction) and the font. It other words sometimes the text can be drawn at the turtle heading and sometimes it cannot. Sometimes what is on the screen will not be exactly what you print.

text(THING) Any thing you wish to label with.

The color of the text is determined by \texttt{SETPENCOLOR}.
The position of the text is determined by the location of the turtle.
The font of the text is determined by \texttt{SETLABELFONT} or \texttt{Set Label Font Command}.
The angle of the text is determined by the heading (direction) of the turtle.
The size of the text can be determined by \texttt{LABELSIZE}.

Example:

label "Hello"

Label can be used in \texttt{PERSPECTIVE} mode but the text will not be on the same plane as the turtle but will be in the correct position. It will still obey the turtles heading on how to orient the text.

Example (Label the 3D axis)

to axis
setlabelfont[['Courier New'] -19 0 0 700 0 0 0 0 3 2 1 49]  
rt 90
sety 200
label "+Y"
sety -200
label "-Y"
sety 0
setx 200
label "+X"
setx -200
label "-X"
setx 0
setz 200
label "+Z"
setz -200
label "-Z"
setz 0
lt 90
end

perspective
cs
setturtle -1
setposxyz [600 600 800]

* "LABEL
* $ LABEL
* K LABEL
* LOGO:0
setturtle 0
axis
# SETPIXEL

**SETPIXEL colorvector**

Set the pixel under the turtle to colorvector. The colorvector is a list of [red green blue] intensities. For an explanation of colorvector see USING COLOR. SETPIXEL is much like doing SETPENCOLOR colorvector FD 1 PU BK 1 PD but much faster. Combined with the command PIXEL you can "Scan" and redraw an image at a new orientation (i.e. rotate a bitmap).

**colorvector:** (LIST) List of three integers representing Red, Green, Blue intensities each in the range 0-255 that you wish to set under the turtle.

Example:

```lisp
pu
setxy -128 -128
for [red 0 255] [for [green 0 255] [setpixel (list:red:green 0) fd 1] bk 256 rt 90 fd 1 lt 90]
```

Setpixel can be used in PERSPECTIVE mode but only the "last" pixel set will be see. In other words, only set pixels of parts of a 3D object that will be in view.

Example (Rotate a 2D image in 3D space):

See Examples/3D/3DBITMAP.LGO example.

```
# SETPIXEL
$ SETPIXEL
K SETPIXEL
* LOGO:0
```
SETLABELFONT

The input, is a list structure that completely describes a font. A font determines what your characters look like on the screen when using the command LABEL. The available fonts depend on your computer. MSWLogo has two ways to specify the font you desire. This command can be used for programs. For interactive use you can use Set Label Font Command (Graphical User Interface) in the Menu. The easiest way to use fonts in MSWLogo is to set the desired font using the Menu and capturing the attributes using LABELFONT command.

Font:(LIST) that contains the following information: [[FaceName] Height Width Orientation Weight Italic Underline StrikeOut CharSet OutPrecision ClipPrecision Quality PitchAndFamily].

Where:

FaceName:(LIST) Specifies the typeface name of the font.

Height:(INTEGER) Specifies the desired height, in logical units, for the font. If this value is greater than zero, it specifies the cell height of the font. If it is less than zero, it specifies the character height of the font.

Width:(INTEGER) Specifies the desired width, in logical units, for the font.

Orientation:(INTEGER) Specifies the desired orientation, in degrees, for the font.

Weight:(INTEGER) Specifies the font weight. This member ranges from 0 to 900 in increments 100. A value of 0 means use default weight.

Italic:(INTEGER) Specifies an italic font if nonzero.

Underline:(INTEGER) Specifies an underlined font if nonzero.

StrikeOut:(INTEGER) Specifies a strikeout font if nonzero.

CharSet:(INTEGER) Specifies the character set of the font.

OutPrecision:(INTEGER) Specifies the Output Precision of the font.

ClipPrecision:(INTEGER) Specifies the Clip Precision of the font.

Quality:(INTEGER) Specifies the Quality of the font.

PitchAndFamily:(INTEGER) Specifies the Pitch and Family of the font.

Note, if you mistype the font name setlabelfont will list what fonts are available.

Example:

setlabelfont [[Times New Roman] -24 0 0 400 0 0 0 0 3 2 1 18] label "Hello

* SETLABELFONT
$ SETLABELFONT
$ SETLABELFONT
* LOGO:0
font LABELFONT

Outputs a list describing the current font.

Font:(LIST) that contains the following information: [FaceName Height Width Orientation Weight Italic Underline StrikeOut CharSet OutPrecision ClipPrecision Quality PitchAndFamily].

See SETLABELFONT for definitions of the members of this LIST.

Example:

setLabelfont [[Times New Roman] -24 0 0 400 0 0 0 3 2 1 18]
show labelfont
[[Times New Roman] -24 0 0 400 0 0 0 3 2 1 18]
**LABELSIZE**

`size LABEL text`  

This command will output the size of the given text. The input, which may be a word or a list is the same as what you would give to `LABEL`. You can use this information to build other forms of `LABEL`. Other forms might be `CENTERLABEL` or `VERTICALLABEL`. You can also use this information to "prepare" a site for text (i.e. frame it or set a background).

`size:(LIST) List of 2 integers [width height] of the text in the current font.`

`text:(THING) Any thing you wish to label with.`

**Example:**

```plaintext```
show labelsize "Hello
[44 24]
```
TEXTSCREEN

TS

(not supported under MSWLogo yet)

Rearranges the size and position of windows to maximize the space available in the text window (the window used for interaction with Logo). The details differ among machines. Compare SPLITSCREEN and FULLSCREEN.
FULLSCREEN

FULLSCREEN
FS

(not supported under MSWLogo yet)

Rearranges the size and position of windows to maximize the space available in the graphics window. The details differ among machines. Compare SPLITSCREEN and TEXTSCREEN.

In the DOS version, switching from fullscreen to splitscreen loses the part of the picture that’s hidden by the text window. Also, since there must be a text window to allow printing (including the printing of the Logo prompt), Logo automatically switches from fullscreen to splitscreen whenever anything is printed. [This design decision follows from the scarcity of memory, so that the extra memory to remember an invisible part of a drawing seems too expensive.]
SPLITSCREEN

SPLITSCREEN
SS

(not supported under MSWLogo yet)

Rearranges the size and position of windows to allow some room for text interaction while also keeping most of the graphics window visible. The details differ among machines. Compare TEXTSCREEN and FULLSCREEN.
SETSCRUNCH xscale yscale

(This is not needed in MSWLogo but available for folks who like to "deliberately misadjust the aspect ratio").

Adjusts the aspect ratio and scaling of the graphics display. After this command is used, all further turtle motion will be adjusted by multiplying the horizontal and vertical extent of the motion by the two numbers given as inputs. For example, after the instruction "SETSCRUNCH 2 1" motion at a heading of 45 degrees will move twice as far horizontally as vertically. If your squares don't come out square, try this. (Alternatively, you can deliberately misadjust the aspect ratio to draw an ellipse.)

For UNIX machines and Macintoshes, both scale factors are initially 1. For DOS machines, the scale factors are initially set according to what the hardware claims the aspect ratio is, but the hardware sometimes lies.

xscale:(NUMBER) X component of aspect ratio.
yscale:(NUMBER) Y component of aspect ratio.

Example:

| setscrunch 2.0 1.0 |
| repeat 4 [fd 100 rt 90] |
| show scrunch     |
| [2 1]             |
REFRESH

REFRESH

(not supported or needed under MSWLogo yet)

Tells Logo to remember the turtle's motions so that they can be reconstructed in case the graphics window is overlaid. The effectiveness of this command may depend on the machine used.
NOREFRESH

(not supported or needed under MSWLogo yet)

Tells Logo not to remember the turtle's motions. This will make drawing faster, but prevents recovery if the window is overlaid.
ZOOM

ZOOM scale

This command allows LOGO to control the scale of the "Screen" window. The argument is the amount to zoom (scale) by. A number greater than 1.0 makes things bigger (e.g. 2.0 makes it 2 times bigger), a number smaller than 1.0 makes things smaller (0.5 makes it 1/2 as big). If an existing image is on the screen when you zoom it will be stretched or squeezed (this takes time by the way) according to the zoom (it may look a little jagged). If you "draw" while zoomed things will not be as jagged. See also the Zoom Menu.

scale:(NUMBER) scale factor to zoom to.

Even though things may appear jagged LOGO remembers everything as if zoom was normal (1.0) and only prints in normal. Once you return to zoom of 1.0 your image will not be stretched or squeezed to fit again. In other words in a zoom of 1.0 lines never get jagged even if you drew it at zoom 0.5 or 2.0.

NOTE: Zoom works best if you choose a zoom that is a "power" of two. For example 2, 4, 8, 1/2 (0.5), 1/4 (0.25), 1/8 (0.125) etc.

Example:

repeat 4 [fd 100 rt 90] zoom 0.5 zoom 2.0

ZOOM
ZOOM
ZOOM
* LOGO:0
SCROLLX

SCROLLX deltax

This command allows LOGO to control the horizontal scroller of the "Screen" window. The argument is the amount to change the scroller by (delta). A positive number scrolls to the right and negative number scrolls to the left. If (delta) is 0 the scroller will reset to its center position.

deltax:(INTEGER) An amount to scroll in X by.

Example:

repeat 10 [scrollx 10]
# $ K • SCROLLY

SCROLLY delta

This command allows LOGO to control the vertical scroller of the "Screen" window. The argument is the amount to change the scroller by (delta). A positive number scrolls down and negative number scrolls to the up. If (delta) is 0 the scroller will reset to it's center position.

delta:(INTEGER) An amount to scroll in Y by.

Example:

repeat 10 [scrolly 10]
SETFOCUS

SETFOCUS caption

This command allows LOGO to control which window is to have focus (is selected). The window desired is specified by its caption (or title). See also SETFOCUS.

caption:(LIST) is the caption on the Window you wish to set focus to.

Note, the "caption" is not the same as the window's "name" used in the WINDOWS functions.

Example:

setfocus [MSWLogo Screen]
caption GETFOCUS

This command will output the caption (title) of the window with the current focus (currently selected). See also SETFOCUS.

caption:(WORD) Will be the caption (title) of window with current focus.

Example:

setfocus [MSWLogo Screen]
show getfocus
[MSWLogo Screen]
ICON

ICON caption

This command allows LOGO to icon a window (if iconable) with the given caption (title). See also UNICON.

caption:(LIST) is the caption on the Window you wish to ICON.

Note, the "caption" is not the same as the window's "name" used in the WINDOWS functions.

Example:

icon "Commander
unicon "Commander
UNICON caption

This command allows LOGO to unicon a window (if iconable) with the given caption (title). See also ICON.

caption:(LIST) is the caption on the Window you wish to UNICON.

Example:

icon "Commander
unicon "Commander
### WINDOWSET

**WINDOWSET caption mode**

This command allows LOGO to control a windows mode (e.g. Hide, Show etc.) with the given **caption** (title). This command is designed for experts, you may loose control of MSWLogo is you are not careful, save your work frequently.

**caption:**(LIST) is the caption on the Window you wish to UNICON.

**mode:**(INTEGER) is the mode you want to set the window to.

**mode 0:** Hides the window and activates another window.

**mode 1:** Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position. An application should specify this flag when displaying the window for the first time.

**mode 2:** Activates the window and displays it as a minimized window.

**mode 3:** Activates the window and displays it as a maximized window.

**mode 4:** Displays the window as a minimized window. The active window remains active.

**mode 5:** Activates the window and displays it in its current size and position.

**mode 6:** Minimizes the specified window and activates the next top-level window in the Z order.

**mode 7:** Displays a window in its most recent size and position. The active window remains active.

**mode 8:** Displays the window in its current state. The active window remains active.

**mode 9:** Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position. An application should specify this flag when restoring a minimized window.

**mode 10:** Reserved.

**Example:**

```logo
to maximize
  windowset "Commander 0
  windowset [MSWLogo Screen] 3
  label [Back in 2 seconds]
  wait 120
  windowset [MSWLogo Screen] 1
  windowset "Commander 1
end
  maximize
```

---

* WINDOWSET
* WINDOWSET
* WINDOWSET
* LOGO:0
USING COLOR

Several commands (SETPENCOLOR, SETFLOODCOLOR, SETSCREENCOLOR) exist in Logo to specify Red Green Blue intensities of color. Each input represents how much Red, Green and Blue you want in the color. Each input has a range of 0-255. By mixing different amounts of colors you can create 16.7 million different colors.

If you’re running with a 256 color Windows Driver for your monitor you will get “real” colors. The colors are stored in a table called a palette. The palette has room for 256 colors (chosen from 16.7 possibilities). You can tell if you are running in 256 color mode (have a palette) by popping up the STATUS window and looking at the palette usage. If it shows "N/A" (Not Applicable) you are not running a 256 (or more) Windows driver. See the CLEARPALETTE command for more details about managing the palette.

If you are running with 16 (or less) color Windows Driver for your monitor Windows will simulate all the colors by mixing (dithering) the 16 (or less) colors. If the SETPENSIZ is too narrow (1) Windows cannot mix (dither) colors. Windows does not mix (dither) colors on Fonts either. If your hardware can support a 256 Windows Driver you should look into loading the appropriate Driver. The Documentation with your Graphics card explains the capabilities and how to load new Drivers. MSWLogo is fun in any mode but it’s even more fun in 256 color mode. You can load 256 color Bitmaps and do some incredible things with them.

Example of setting some common pen colors:

setpencolor [000 000 000] black
setpencolor [255 255 255] white
setpencolor [128 128 128] gray
setpencolor [255 000 000] Red
setpencolor [000 255 000] Green
setpencolor [000 000 255] Blue

Note that when running 256 color mode with a palette that several advantages and disadvantages occur:

Printing a 256 color image on a mono printer will be less pleasing than printing in 16 color mode. This is because no "dithering" (mixing dots) is used in 256 color mode.

Running in 256 color mode is slower and takes more memory and may not be possible on some smaller machines or machines with limited video capabilities.

Running in 256 color does give much more pleasant colors and even allows manipulation of 256 color .BMP files.

For the purposes of compatibility with other logo implementations and simplicity for younger logo programmers MSWLogo will also support single index colors. If you set any color with an 16 color index MSWLogo will output all colors as a 16 color index. If you set any color with an [Red Green Blue] color vector MSWLogo will output all colors as [Red Green Blue] color vectors.

Example:

show pencolor [0 0 0]
setpencolor 5
show pencolor 5
setpencolor [255 0 0]
show pencolor [255 0 0]
The index colors map as follows:

0 -> [ 0 0 0]
1 -> [ 0 0 255]
2 -> [ 0 255 0]
3 -> [ 0 255 255]
4 -> [255 0 0]
5 -> [255 0 255]
6 -> [255 255 0]
7 -> [255 255 255]
8 -> [155 96 59]
9 -> [197 136 18]
10 -> [100 162 64]
11 -> [120 187 187]
12 -> [255 149 119]
13 -> [144 113 208]
14 -> [255 163 0]
15 -> [183 183 183]
### Understand your Orientation in 3D

In 2D life is simple:

- **HEADING** is effected by `LEFT`, `RIGHT`, `SETHEADING`.

But in 3D (`PERSPECTIVE`) mode things are bit more complicated:

- **HEADING** is effected by `LEFT`, `RIGHT`, `SETHEADING`
- **ROLL** is effected by `LEFTROLL`, `RIghtROLL`, `SETROLL`
- **PITCH** is effected by `UPPitch`, `DOWNPitch`, `SETPitch`

However there is more to it than that. Your **ROLL** can also effect your **HEADING**. Your **PITCH** can also effect your **ROLL** and so on. They all interact with one another. The easiest way to understand this is with an example:

```logo
turtle perspective
right 90
(show round roll round pitch round heading)
0 0 90
cs
downpitch 90
rightroll 90
uppitch 90
(show round roll round pitch round heading)
0 0 90
```

I only used rolls and pitches but the heading changed. But it is correct, think about the maneuvers I made and where I should be pointing.

But don’t worry MSWLogo makes it easy, you just need to understand some basic rules so that you don’t get lost in 3D space. The turtle controls **ROLL**, **PITCH**, and **HEADING** do effect one another but in a certain way which is important. The most important is that a change in **HEADING** (with **RIGHT**, **LEFT** or **SETHEADING**) will not effect your **ROLL** or **PITCH**. This is what allows you to use a lot of your existing 2D code in 3D.

There is also another rule to understand, you cannot simply say.

```logo
turtle perspective
setroll 45
setpitch 45
setheading 45
(show round roll round pitch round heading)
45 45 45
```

And expect to be at an absolute orientation of 45, 45, 45.

```logo
turtle cs
right 45
rightroll 30
uppitch 30
; Now try to set your self up to 45,45,45 again
setroll 45
setpitch 45
```

# Understand your_Orientation_in_3D

$ Understand your Orientation in 3D

K Understand your Orientation in 3D

* LOGO:0
setheading 45
(show round roll round pitch round heading)

But don’t worry there is a solution, it’s called SETORIENTATION.

cs
right 45
rightroll 30
uppitch 30
; Now try to set your self up to 45,45,45 again
setorientation [45 45 45]
(show round roll round pitch round heading)

45 45 45

SETORIENTATION understands the interactions between ROLL, PITCH and HEADING. All SETORIENTATION
knows is that it has to do things in the correct order. It basically backs out your current ORIENTATION and applies the
requested one. It does this in a very specific order. Your current ORIENTATION can also be reported or saved using
the ORIENTATION command.

cs
right 45
show orientation
[0 0 45]
make "saveit orientation
right 90
uppitch 180
setorientation :saveit
show orientation
[0 0 45]

There is also another command that knows about your ORIENTATION and that is TOWARDSXYZ. TOWARDSXYZ
will output an orientation that is compatible with SETORIENTATION.

setorientation towardsxyz [100 100 0]
show orientation
[0 0 45]
forward distancexyz [100 100 0]
show posxyz
[100 100 0]

Having said all that, you don’t really need to think in terms of absolute ORIENTATION. ORIENTATION and
SETORIENTATION will be used mostly to save, report and restore your ORIENTATION (just like you do with
HEADING and SETHEADING in 2D). And like explained earlier changes in your heading with RIGHT, LEFT and
SETHEADING only effect what they effected in 2D (your HEADING).

You should also be thinking the same way you do in 2D. You don’t often worry about your absolute HEADING in 2D,
it’s most often a turn relative to your current HEADING. It’s no different in 3D except you have more ways to turn.
Think like you’re an airplane and that you’re in the cockpit of the turtle. Then decide where you want to go next.
# S K • Drawing 3D Solids

3D Solids in MSWLogo uses all the standard 3D commands except a few additional commands to help MSWLogo understand what you’re drawing. In order to do hidden line removal or hidden surface removal you have to define surfaces (not just vectors). Surfaces are simply filled polygons. You just need to tell MSWLogo when you’re starting a surface and when you’re done. If you used only vectors to draw a box, MSWLogo would not know if it’s a 6 Sided closed box (solid cube) or a 4 Sided Box with the top and bottom open. Vectors alone are ambiguous with respect to surfaces.

Logo programmers are used to dealing with filled polygons, you do it all the time so it should not be a strange new concept. But MSWLogo in 3D mode will take care of the correct filling of that polygon with respect to the color you should see based on your eye view, light source, obstructions, and the color of the surface.

Here are the new commands:

Polystart: Start a new polygon (a surface of pencolor)
Polyend: End the definition of the polygon and display it.
Polyview: Shade and view all polygons that have been defined.
Setturtle -3: Use this turtle to position light source.
Setlight: Use to set how objects should be illuminated.

The PEN color when drawing 3D surfaces is the color of the surface. If You drew a BOX in 3D (6 Polygons) that all were drawn the same color, all you’d see is a Filled octagon looking thing. Color is effected by the angle of the surface between your eye (good'ol TURTLE -1) and the light source (new TURTLE -3).

The 3D lighting model in MSWLogo is very powerful and uses some of the most modern techniques used but it is designed for performance. It will not do shadows and reflections etc. You may see Pixel errors sometimes on yours this is normal and an artifact of the algorithm I chose.

POLYSTART
POLYEND
POLYVIEW
SETLIGHT
LIGHT

# Drawing_3D_Solids
$ Drawing 3D Solids
K Drawing 3D Solids
* LOGO:0
POLYSTART

POLYSTART tells MSWLogo that you are about to define a polygon. Each subsequent move will result in the destination coordinate being added as a vertex to the polygon. The definition is completed by issuing the command POLYEND. To view ALL defined polygons as shaded polygons you must issue the command POLYVIEW. Make sure you understand the Polygon Restrictions.

Example:

to square
   polystart
   repeat 4 [fd 100 rt 90]
   polyend
end

   perspective
   repeat 72 [square rr 5]
   polyview
POLYEND

POLYEND tells MSWLogo that you have completed the definition of a polygon that you started when you issued the command POLYSTART. Note that the current coordinate is not added as a vertex when you issue the POLYEND command. But the move that you issued to get to the current coordinate will be included. Make sure you understand the Polygon Restrictions.

Example:

to square
polystart
repeat 4 [fd 100 rt 90]
polyend
end

perspective
repeat 72 [square rr 5]
polyview
POLYVIEW

POLYVIEW asks MSWLogo to erase the screen and view all polygons defined with POLYSTART and POLYEND as shaded polygons. Only shaded polygons will be shown. Polygons will be illuminated according to the position of the light source using SETTURTLE -3 and lighting parameters using with SETLIGHT. Make sure you understand the Polygon Restrictions.

Note that if you need to add other things to your drawing such as LABELs or BITMAPs etc. you must do them after the last POLYVIEW.

Note that you can issue POLYVIEW immediately after any POLYEND to watch polygons get shaded as your program builds your object. But keep in mind POLYVIEW is an expensive operation and will slow down the program considerably if you call it with every POLYEND.

Example:

to square
polystart
repeat 4 [fd 100 rt 90]
polyend
end

perspective
repeat 72 [square rr 5]
polyview
SETLIGHT

Sets how objects will be illuminated. The argument is a list of two numbers ranging from 0.0 to 1.0, the amount of ambient and diffuse light. Ambient defines how brightly the scene is being lit, the larger the number the brighter the light. Diffuse is not a property of light source, but a property of how smooth the surface is (how much it scatters light). A higher value means the objects are smoother and reflective (like a marble) a low value means the objects are rough (like chalk). To position the light use SETTURTLE -3. See also LIGHT.

**type:** (LIST) List of two numbers representing desired Ambient and Diffuse light (each in the range of 0.0 to 1.0).

Example 1:

```plaintext
setlight [0.3 0.6]
show light
[0.3 0.6]
```

Example 2:

```plaintext
to square
polystart
repeat 4 [fd 100 rt 90]
polyend
end
cs
perspective
clearpalette
setpc [255 0 0]
setsc [0 0 0]
repeat 72 [square rr 5]
polyview
repeat 50 [setlight (list (random 100)/100 (random 100)/100) print light wait 60]
```
# $ K * LIGHT

type LIGHT

Outputs the lighting attributes, as a list of two numbers, the Ambient and Diffuse values. See also SETLIGHT.

type:(LIST) List of two numbers representing the current Ambient and Diffuse values.

Example:

setlight [0.3 0.6]
show light
[0.3 0.6]
# $ K * Polygon Restrictions

Polygon restrictions:

1: A polygon's first 3 vertices must form a triangle.
2: A polygon must be convex.
3: A polygon must be planar (no up, down, ri, lr) between polystart/polyend.

All these properties occur very naturally in Logo.

Notes on 1: the triangle can be extremely shallow

This is legal:

polystart
make "savepos posxyz
fd 100
rt 0.00000001
fd 100
setpos :saveposxyz
polyend

This is NOT legal (even though it produces a convex polygon)

polystart
make "savepos posxyz
fd 50
fd 50
rt 90
fd 100
setpos :saveposxyz
polyend

Notes on 2: What does convex mean?

Pick any 2 coordinates within the polygon and connect them with a straight line. If that line goes outside the polygon then it's not convex.

Triangles, Squares, Circles etc. are convex.
The letter "E" as a polygon is not convex.

This is legal

repeat 4 [fd 100 rt 90]

This is NOT legal

repeat 4 [fd 50 rt 90 repeat 3 [fd 25 lt 90] rt 180 fd 50 rt 90]

Notes on 3: What does planar mean?

# Polygon_Restrictions
$ Polygon Restrictions
K Polygon Restrictions
* LOGO:0
The polygon must be flat

This is legal

repeat 4 [fd 100 rt 90]

This is NOT legal

repeat 4 [fd 50 rr 90 rt 90 repeat 3 [fd 25 lt 90] rt 180 lr 90 fd 50 rt 90]

Notes on 2 and 3:

You can work around restriction 2 and 3 breaking the object up into simpler polygons that meet the rules.

You can use CIRCLE, ELLIPSE and ARC.

Example (using circle):

```
perspective
repeat 36 ~
  [ polystart
     circle 100
     polyend
     rr 10
   ]
polyview
```

Example (using arc):

```
perspective
repeat 36 ~
  [ polystart
     arc2 180 100
     RT 90
     FD 200
     RT 90
     polyend
     rr 10
   ]
polyview
```
TURTLE AND WINDOW QUERIES

SHOWNP
SHOWNP

truth SHOWNP

Outputs TRUE if the turtle is shown (visible), FALSE if the turtle is hidden. See SHOWTURTLE and HIDETURTLE.

truth:(BOOLEAN) Truth as to whether the current turtle is shown.

Example:

showturtle
show shownp
true
hideturtle
show shownp
false
PEN CONTROL

The turtle carries a pen that can draw pictures. At any time the pen can be UP (in which case moving the turtle does not change what's on the graphics screen) or DOWN (in which case the turtle leaves a trace). If the pen is down, it can operate in one of three modes: PAINT (so that it draws lines when the turtle moves), ERASE (so that it erases any lines that might have been drawn on or through that path earlier), or REVERSE (so that it inverts the status of each point along the turtle's path).

PENDOWN
PENUP
PENPAINT
PENERASE
PENREVERSE
PENNORMAL
SETPENCOLOR
SETFLOODCOLOR
SETSCREENCOLOR
SETPENSIZE
SETPENPATTERN
SETPEN
CLEARPALETTE
PENDOWN
PD

Sets the pen's position to DOWN, without changing its mode.

Example:

repeat 10 [fd 10 pu fd 10 pd]
 Sets the pen's position to UP, without changing its mode.

Example:

code

repeat 10 [fd 10 pu fd 10 pd]
# $K • PENPAINT

PENPAINT
PPT

Sets the pen's position to DOWN and mode to PAINT. PAINT is the normal mode the turtle draws in.

Example:

penreverse
fd 100
bk 100
penpaint
fd 100
bk 100
$ $ $ K • PENERASE

PENERASE
PE

Sets the pen's position to DOWN and mode to ERASE. To restore the pen to normal use PENPAINT.

Example:

fd 100
penerase
bk 100

* PENERASE
$ PENERASE
$ PENERASE
$ PENERASE
* LOGO:0
# $ K • PENREVERSE

PENREVERSE
PX

Sets the pen's position to DOWN and mode to REVERSE. To restore the pen to normal use PENPAINT.

Example:

penreverse
fd 100
bk 100
penpaint
fd 100
bk 100

* PENREVERSE
$ PENREVERSE
# PENREVERSE
$ LOGO:0
PENNORMAL

PENNORMAL (library procedure)

Sets the pen's state to the startup default of down and paint.

Example:

penreverse
fd 100
pennormal
bk 100
SETPENCOLOR

SETPENCOLOR colorvector
SETPC colorvector

Sets the pen to colorvector. The colorvector is a list of [red green blue] intensities. The pen color effects drawing text (LABEL) and drawing any line with the turtle (such as FORWARD). For an explanation of the arguments see USING COLOR.

colorvector:(LIST) List of three integers representing Red, Greed, Blue intensities each in the range 0-255.

Example:

repeat 255 [setpencolor (list repcount 0 0) fd 100 bk 100]

Example 2:(the most common logo question)

make "red 100
make "green 100
make "blue 100
setpencolor [:red :green :blue]
<will fail>
setpencolor (list :red :green :blue)
<will work>
SETSFLLOODCOLOR

SETSFLLOODCOLOR colorvector
SETFC colorvector

Sets the flood color to colorvector. The colorvector is a list of [red green blue] intensities. The flood color effects FILL and BITBLOCK commands. For an explanation of the arguments see USING COLOR.

colorvector:(LIST) List of three integers representing Red, Green, Blue intensities each in the range 0-255.

Example:

cs
repeat 4 [fd 100 rt 90]
rt 45
pu
fd 20
setfloodcolor [0 255 0]
fill
cs
repeat 4 [fd 100 rt 90]
rt 45
pu
fd 20
setfloodcolor [125 125 125]
fill
**SETSCREENCOLOR**

SETSCREENCOLOR colorvector
SETSC colorvector

Sets the screen color to **colorvector**. The **colorvector** is a list of [red green blue] intensities. The screen color immediately sets the screen background color. For an explanation of the arguments see [USING COLOR](#).

**colorvector**: (LIST) List of three integers representing Red, Greed, Blue intensities each in the range 0-255.

Example:

```
set screencolor [0 0 0]
set pencolor [255 255 255]
repeat 4 [fd 100 rt 90]
```
SETPENSIZE

SETPENSIZE size

Set hardware-dependent pen characteristics. These commands are not guaranteed compatible between implementations on different machines. The "size" is a list of two members, width and height. MSWLogo only uses the width of them. So just set them both to the same value.

size:(LIST) List of 2 integers representing width and height of the pen.

Example:

setpensize [5 5]
SETPENPATTERN

SETPENPATTERN pattern

(not supported under MSWLogo yet)

Set hardware-dependent pen characteristics. These commands are not guaranteed compatible between implementations on different machines.
SETPEN list  (library procedure)

Sets the pen's position, mode, and hardware-dependent characteristics according to the information in the input list, which should be taken from an earlier invocation of PEN.

list:(LIST) A list representing the pen's state as follows [pendown penmode pensize pencolor penpaint].

Example:

```
cs repeat 4 [fd 100 rt 90]
make "savepen pen
setpensize [20 20]
cs repeat 4 [fd 100 rt 90]
setpen :savepen
cs repeat 4 [fd 100 rt 90]
```
# Clearpalette

This command clears the color palette. The color palette is filled by using the \texttt{SETPENCOLOR}, \texttt{SETSCREENCOLOR} and \texttt{SETFLOODCOLOR} commands. The palette is only supported when windows is in 256 color mode (see \texttt{USING COLOR}). Once you run out of colors windows will choose the closest match. For example if the command

\begin{verbatim}
repeat 256 [setpencolor (list repcount 0 0)]
\end{verbatim}

is issued it would fill the palette with 256 shades of red. At this point the palette would now be full. If you now wanted 256 shades of green they would NOT be granted (and matched to red). In order for them to be granted you have to let go of the 256 shades of red. This is done by "clearing the palette".

If you want a wide range of colors then select a wide range into the palette. For example the following would give you 216 colors covering a wide range. Once the few colors, left in the palette, are used, windows will have something reasonable to match further requests to (unlike the example above in which only shades of reds could be matched to).

\begin{verbatim}
repeat 6-
 |
 make "red repcount*40-
 repeat 6-
 |
 make "green repcount*40-
 repeat 6-
 |
 make "blue repcount*40-
 setpencolor (list:red:green:blue)-
 ]~
 ]~
 ]
\end{verbatim}

Note also that loading in .BMP (\texttt{BITLOAD}) files use up colors in the palette. Which again can be cleared using clearpalette. Clearing the screen does NOT clear the palette.

Example:

\begin{verbatim}
repeat 256 [setpencolor (list repcount 0 0)]
status
nostatus
clearpalette
status
nostatus
\end{verbatim}
PEN QUERIES

PENDOWNP
PENMODE
PENCOLOR
FLOODCOLOR
SCREENCOLOR
PENSIZE
PENPATTERN
PEN

# PEN QUERIES
$ PEN QUERIES
% PEN QUERIES
* LOGO:0
PENDOWNP

truth PENDOWNP

Outputs TRUE if the pen is down, FALSE if it's up.

truth:(BOOLEAN) Truth as to whether the current pen is down.

Example:

pu
ifelse pendownp [print [Pen is DOWN]] [print [Pen is UP]]
  Pen is UP
pd
ifelse pendownp [print [Pen is DOWN]] [print [Pen is UP]]
  Pen is DOWN
# PENMODE

mode PENMODE

Outputs one of the words PAINT, ERASE, or REVERSE according to the current pen mode.

mode:(WORD) Current mode of the pen.

Example:

penpaint
show penmode
paint
penerase
show penmode
erase
penreverse
show penmode
reverse

* PENMODE
$ PENMODE
% PENMODE
* LOGO:0
PENCOLOR

Set PENCOLOR

Output pen color information which is a list containing [Red Green Blue] intensities.

PENCOLOR:(LIST) List of three integers representing Red, Greed, Blue intensities each in the range 0-255.

Example:

(setpencil [100 200 50])
(show pencil)
[100 200 50]
(setpencil [0 0 0])
(show pencil)
[0 0 0]
FLOODECOLOR

colorvector FLOODECOLOR

Output flood color information which is a list containing [Red Green Blue] intensities.

colorvector: (LIST) List of three integers representing Red, Green, Blue intensities each in the range 0-255.

Example:

setfloodcolor [100 200 50]
show floodcolor
[100 200 50]
setfloodcolor [0 0 0]
show floodcolor
[0 0 0]

# $K • FLOODECOLOR

$ FLOODECOLOR
K FLOODECOLOR
* LOGO:0
# $K$ * SCREENCOLOR

colorvector SCREENCOLOR.

Output screen color information which is a list containing [Red Green Blue] intensities.

colorvector:(LIST) List of three integers representing Red, Green, Blue intensities each in the range 0-255.

Example:

setscreencolor [100 200 50]
show screencolor
[100 200 50]
setscreencolor [0 0 0]
show screencolor
[0 0 0]
size PENSIZ

Output pen size information which is a list containing [Width Height]. Width is not used in MSWLogo.

size:(LIST) List of 2 integers representing width and height of the pen.

Example:

setpensize [10 20]
show pensize
[20 20]
setpensize [1 1]
show pensize
[1 1]
# $K$ • PENPATTERN

**pattern** PENPATTERN

(not supported under MSWLogo yet)

Output hardware-specific pen information.
list PEN  (library procedure)

Outputs a list containing the pen's position, mode, and hardware-specific characteristics, for use by SETPEN.

list: (LIST) A list representing the pens state as follows [pendown penmode pensize pencolor penpaint].

Example:

cs repeat 4 [fd 100 rt 90]
make "savepen pen
setpensize [20 20]
cs repeat 4 [fd 100 rt 90]
setpen :savepen
cs repeat 4 [fd 100 rt 90]
WORKSPACE MANAGEMENT

PROCEDURE DEFINITION
VARIABLE DEFINITION
PROPERTY LISTS
PREDICATES (Workspace)
QUERIES
INSPECTION
WORKSPACE CONTROL
PROCEDURE DEFINITION

TO
END
DEFINE
TEXT
FULLTEXT
COPYDEF

* PROCEDURE_DEFINITION
* PROCEDURE_DEFINITION
* PROCEDURE_DEFINITION
* LOGO:0
# TO

TO proname :input1 :input2 ...  (special form)

Command that prepares Logo to accept a procedure definition. The procedure will be named proname and there must not already be a procedure by that name. The inputs will be called input1, input2, etc. Any number of inputs is allowed, including none. Names of procedures and inputs are case-insensitive.

proname:(WORD) Name of the procedure being defined (but not preceded by a []).
input1:(WORD) First argument to procedure being defined (preceded by a []).
input2:(WORD) Second argument to procedure being defined (preceded by a []).

Unlike every other Logo procedure, TO takes as its inputs the actual words typed in the instruction line, as if they were all quoted, rather than the results of evaluating expressions to provide the inputs. (That’s what "special form" means.)

This version of Logo allows variable numbers of inputs to a procedure. Every procedure has a MINIMUM, DEFAULT, and MAXIMUM number of inputs. (The latter can be infinite.)

The MINIMUM number of inputs is the number of required inputs, which must come first. A required input is indicated by the

 :inputname

notation.

After all the required inputs can be zero or more optional inputs, represented by the following notation:

[:inputname default.value.expression]

When the procedure is invoked, if actual inputs are not supplied for these optional inputs, the default value expressions are evaluated to set values for the corresponding input names. The inputs are processed from left to right, so a default value expression can be based on earlier inputs. Example:

to proc :inlist [:startvalue first :inlist]

If the procedure is invoked by saying

proc [a b c]

then the variable INLIST will have the value [A B C] and the variable STARTVALUE will have the value A. If the procedure is invoked by saying

(proc [a b c] "x")

then INLIST will have the value [A B C] and STARTVALUE will have the value X.

After all the required and optional input can come a single "rest" input, represented by the following notation:

[:inputname]

This is a rest input rather than an optional input because there is no default value expression. There can be at most one rest input. When the procedure is invoked, the value of this input will be a list containing all the actual inputs provided that were not used for required or optional inputs. Example:

* TO
† TO
‡ TO
* LOGO:0
to proc :in1 [:in2 "foo] [:in3]

If this procedure is invoked by saying

proc "x

then IN1 has the value X, IN2 has the value FOO, and IN3 has the value [] (the empty list). If it's invoked by saying

(proc "a "b "c "d)

then IN1 has the value A, IN2 has the value B, and IN3 has the value [C D].

The MAXIMUM number of inputs for a procedure is infinite if a rest input is given; otherwise, it is the number of required inputs plus the number of optional inputs.

The DEFAULT number of inputs for a procedure, which is the number of inputs that it will accept if its invocation is not enclosed in parentheses, is ordinarily equal to the minimum number. If you want a different default number you can indicate that by putting the desired default number as the last thing on the TO line. example:

to proc :in1 [:in2 "foo] [:in3] 3

This procedure has a minimum of one input, a default of three inputs, and an infinite maximum.

Logo responds to the TO command by entering procedure definition mode. The prompt character changes from "?" to ">" (pops up a dialog box in MSWLogo) and whatever instructions you type become part of the definition until you type a line containing only the word END.

Example:

to echo :times :thing
repeat :times [print :thing]
end
echo 2 "Hello
Hello
Hello
echo 3 "Bye
Bye
Bye
Bye
END  (special form)

This is not really a command. It is to let you to depict the END of a procedure. See also TO.

Example:

to echo :times :thing
repeat :times [print :thing]
end
echo 2 "Hello
Hello
Hello
echo 3 "Bye
Bye
Bye
Bye
## DEFINE

```
DEFINE proname text
```

Command that defines a procedure with name `proname` and text `text`. If there is already a procedure with the same name, the new definition replaces the old one. The text input must be a list whose members are lists. The first member is a list of inputs; it looks like a TO line but without the word TO, without the procedure name, and without the colons before input names. In other words, the members of this first sublist are words for the names of required inputs and lists for the names of optional or rest inputs. The remaining sublists of the text input make up the body of the procedure, with one sublist for each instruction line of the body. (There is no END line in the text input.) It is an error to redefine a primitive procedure unless the variable `REDEFP` has the value TRUE.

- **proname**: (WORD) Name of the procedure being defined.
- **text**: (LIST) Definition of procedure as a list.

### Example:

```
define "abc [[a b] [print :a] [print :b]]
define "Hello "Bye
Hello
Bye
```
text TEXT proname

Outputs the text of the procedure named "proname" in the form expected by DEFINE: a list of lists, the first of which describes the inputs to the procedure and the rest of which are the lines of its body. The text does not reflect formatting information used when the procedure was defined, such as continuation lines and extra spaces.

text:LIST) Definition of procedure as a list.

procname:(WORD) Name of the procedure.

Example:

define "abc [[a b] [print :a] [print :b]]
abc "Hello "Bye
Hello
Bye
show text "abc
[[a b] [print :a] [print :b]]
text FULLTEXT proname

Outputs a representation of the procedure "proname" in which formatting information is preserved. If the procedure was defined with \texttt{TO EDIT}, or \texttt{LOAD}, then the output is a list of words. Each word represents one entire line of the definition in the form output by \texttt{READWORD}, including extra spaces and continuation lines. The last element of the output represents the \texttt{END} line. If the procedure was defined with \texttt{DEFINE}, then the output is a list of lists. If these lists are printed, one per line, the result will look like a definition using \texttt{TO}. Note: the output from \texttt{FULLTEXT} is not suitable for use as input to \texttt{DEFINE}.

\texttt{text:(LIST)} Definition of procedure as a list.

\texttt{proname:(WORD)} Name of the procedure.

Example:

\begin{verbatim}
define "abc [[a b] [print :a] [print :b]]
abc "Hello "Bye
Hello
Bye
show fulltext "abc
[[to abc :a :b] [print :a] [print :b] end]
\end{verbatim}
# $ K  •  COPYDEF

COPYDEF newname oldname

Command that makes "newname" a procedure identical to "oldname". The latter may be a primitive. If "newname" was already defined, its previous definition is lost. If "newname" was already a primitive, the redefinition is not permitted unless the variable REDDEF has the value TRUE. Definitions created by COPYDEF are not saved by SAVE; primitives are never saved, and user-defined procedures created by COPYDEF are buried. (You are likely to be confused if you PO or POT a procedure defined with COPYDEF because its title line will contain the old name. This is why it's buried.)

Note: dialects of Logo differ as to the order of inputs to COPYDEF. This dialect uses "MAKE order," not "NAME order."

newname:(WORD) Name of the new procedure.
oldname:(WORD) Name of the old procedure.

Example:

to welcome
  print "Hello
end
welcome
Hello
copydef "sayhello "welcome
sayhello
Hello

# COPYDEF
$ COPYDEF
& COPYDEF
* COPYDEF
* LOGO:0
VARIABLE DEFINITION

MAKE
NAME
LOCAL
LOCALMAKE
THING
MAKE varname value

Command that assigns the value value to the variable named varname, which must be a word. Variable names are case-insensitive. If a variable with the same name already exists, the value of that variable is changed. If not, a new global variable is created.

varname: (WORD) Name of the variable you wish to assign.
value: (THING) Thing to be assigned to variable.

Example:

make "foo [Hello how are you]
show :foo
[Hello how are you]
NAME

NAME value varname (library procedure)

Command that assigns the value value to the variable named varname, which must be a word. Variable names are case-insensitive. If a variable with the same name already exists, the value of that variable is changed. If not, a new global variable is created. This is the same as MAKE but with the inputs in reverse order.

value:(THING) Thing to be assigned to variable.

varname:(WORD) Name of the variable you wish to assign.

Example:

ame [Hello how are you] "foo
show :foo
[Hello how are you]
LOCAL varname
LOCAL varnamelist
(LIST varname1 varname2 ...)

Command that accepts as inputs one or more words, or a list of words. A variable is created for each of these words, with that word as its name. The variables are local to the currently running procedure. Logo variables follow dynamic scope rules; a variable that is local to a procedure is available to any sub procedure invoked by that procedure. The variables created by LOCAL have no initial value; they must be assigned a value (e.g., with MAKE) before the procedure attempts to read their value.

varname (WORD) Name of the variable you wish to localize.
varnamelist (LIST) List of names (words) of the variables you wish to localize.
varname1 (WORD) First name of the variable you wish to localize.
varname2 (WORD) Second name of the variable you wish to localize.

Example:

to foo
make "bar 1
print :bar
end
foo 1
show :bar 1
to abc
local "xyz
make "xyz 1
print :xyz
end
abc 1
show :xyz
xyz has no value
**LOCALMAKE**

Localmake varname value (library procedure)

Localmake is short hand for the command **LOCAL** varname followed by the command **MAKE** varname value.

varname (WORD) Name of the variable you wish to assign locally.

value (THING) Thing to be assigned to local variable.

Example:

See **LOCAL** for an example.
\# $K \cdot THING

value THING varname
:quoted.varname

Outputs the value of the variable whose name is the input. If there is more than one such variable, the innermost local variable of that name is chosen. The colon notation is an abbreviation not for THING but for the combination thing *

so that :FOO means THING "FOO.

value:(THING) Thing to be output.

varname:(WORD) Name of the variable you wish to output.

Example:

make "foo [Hello how are you]
show thing "foo
[Hello how are you]
show :foo
[Hello how are you]

$ THING
\$ THING
$K THING
*K THING
* LOGO:0
# § K • PROPERTY LISTS

Note: Names of property lists are always case-insensitive. Names of individual properties are case-sensitive or case-insensitive depending on the value of `CASEIGNOREDP`, which is TRUE by default.

**PPROP**
**GPROP**
**REMPROP**
**PLIST**
PPROP plistname proppname value

Command that adds a property to the plistname property list with name proppname and value value.

plistname: (WORD) Name of the property list you wish to add to.
proppname: (WORD) Name of the property you wish to assign.
value: (THING) Thing to be assigned to property.

Example:

pprop "plist1 "p1 1
pprop "plist1 "p2 2
pprop "plist2 "p1 10
pprop "plist2 "p2 20
show gprop "plist1 "p1 1
show gprop "plist1 "p2 2
show gprop "plist2 "p1 10
show gprop "plist2 "p2 20
value GPROP plistname propname

Outputs the value value of the propname property in the plistname property list, or the empty list if there is no such property.

value: (THING) Property value you wish to output.

plistname: (WORD) Name of the property list you wish to access.

propname: (WORD) Name of the property you wish to output.

Example:

pprop "plist1 "p1 1
pprop "plist1 "p2 2
pprop "plist2 "p1 10
pprop "plist2 "p2 20
show gprop "plist1 "p1
show gprop "plist1 "p2
show gprop "plist2 "p1
show gprop "plist2 "p2

* GPROP
$ GPROP
& GPROP
* LOGO:0
REMPROP

REMPROP plistname propname

Command that removes the property named "propname" from the property list named "plistname".

plistname:(WORD) Name of the property list you wish to remove from.
propname:(WORD) Name of the property you wish to remove.

Example:

pprop "plist1 "p1 1
pprop "plist1 "p2 2
show plist "plist1
[p2 2 p1 1]
remprop "plist1 "p1
show plist "plist1
[p2 2]
# $ K • PLIST

list PLIST plistname

Outputs a list whose odd-numbered elements are the names, and whose even-numbered elements are the values, of the properties in the property list named "plistname". The output is a copy of the actual property list; changing properties later will not magically change the list output by PLIST.

list:(LIST) List of Property and Value pairs.

plistname:(WORD) Name of the property list you wish to list.

Example:

pprop "plist1 "pl 1
pprop "plist1 "p2 2
show plist "plist1
[p2 2 pl 1]
# $K$ • PREDICATES (Workspace)

PROCEDUREP
PRIMITIVEP
DEFINEDP
NAMEP
MACROP
truth PROCEDUREP name

Outputs TRUE if the input is the name of a procedure.

truth: (BOOLEAN) Truth as to whether name is a procedure.

name: (WORD) Name of the procedure to test for.

Example:

make "foo 1
to bar
der
to procedurep "foo
false
show procedurep "bar
true
truth PRIMITIVEP name

Outputs TRUE if the input is the name of a primitive procedure (one built into Logo). Note that some of the procedures described in this document are library procedures, not primitives.

truth:(BOOLEAN) Truth as to whether name is a primitive procedure.

name:(WORD) Name of the primitive procedure to test for.

Example:

to fwd :arg
forward :arg
end
show primitivep "fwd
false
show primitivep "forward
true
$ K * DEFINEDP

truth DEFINEDP name

Outputs TRUE if the input is the name of a user-defined procedure, including a library procedure. (However, Logo does not know about a library procedure until that procedure has been invoked.)

truth: (BOOLEAN) Truth as to whether name is defined.

name: (WORD) Name of the defined procedure to test for.

Example:

define "abc [(a b) [print :a] [print :b]]
show definedp "forward
false
show definedp "abc
true
NAMEP

truth NAMEP name

Outputs TRUE if the input is the name of a variable.

truth: (BOOLEAN) Truth as to whether name is a name.

name: (WORD) Name of the name to test for.

Example:

define "abc [[a b] [print :a] [print :b]]
make "xyz 1
show namep "abc
false
show namep "xyz
true
MACROP

truth MACROP name

Outputs TRUE if the input is the name of a user-defined macro, including a library macro. (However, Logo does not know about a library macro until that procedure has been invoked.)

truth : (BOOLEAN) Truth as to whether name is a define macro.

name : (WORD) Name of the defined macro to test for.

Example:

show macro "abc
false
.defmacro "abc [([a b] [print :a] [print :b])
show macro "abc
true
# $K$ • QUERIES

**CONTENTS**
**BURIED**
**PROCEDURES**
**NAMES**
**PLISTS**
**NAMELIST**
**PLLIST**

* QUERIES.2
* QUERIES
* QUERIES
* LOGO:0
`contentslist` CONTENTS

Outputs a `contentslist`, a list of three lists containing names of defined procedures, variables, and property lists respectively. This list includes all unburied named items in the workspace.

`contentslist:(LIST)` List of three lists containing procedures, variables and properties.

Example:

```
to procl
end
make "namel 1
pprop "propl "pl 1
show contents
[[procl] [namel] [propl]]
```
# $K • BURIED

**contentslist** BURIED

Outputs a **contentslist** including all buried named items in the workspace.

**contentslist**(LIST) List of three lists containing buried procedures, variables and properties.

Example:

```
show buried
[[[] [[]]]
pots
show buried
[[[pots] [] []]]
```
# § K • PROCEDURES

**proclist** PROCEDURES

Outputs a list of the names of all unburied user-defined procedures in the workspace. Note that this is a list of names, not a contents list. (However, procedures that require a contents list as input will accept this list.)

**proclist**: (LIST) List of buried procedure names (words).

Example:

to foo
end
to bar
end
show procedures
[bar foo]
 NAMES

contentslist NAMES

Outputs a contentslist consisting of an empty procedure list (indicating no procedure names) followed by a list of all unbunded variable names in the workspace.

contentslist:[LIST] List of 2 lists containing an empty procedure list followed by a list of variable names.

Example:

make "foo 1
make "bar 2
show names

[] [bar foo]
PLISTS

Contentslist PLISTS

Outputs a contentslist consisting of two empty lists (indicating no procedures or variables) followed by a list of all unburied property lists in the workspace.

Contentslist:(LIST) List of 3 lists containing an empty procedure list, empty name list, followed by a list of property list names.

Example:

pprop "propl "pl l
show plists
[[] [] [propl]]
NAMELIST

contentslist NAMELIST varname   (library procedure)
contentslist NAMELIST varnamelist

Outputs a contentslist consisting of an empty list followed by a list of the name or names given as input. This is useful in conjunction with workspace control procedures that require a contents list as input. These names do name have to exist to be use this function.

contentslist:(LIST) List of 2 lists containing an empty procedure list followed by a list of variable names.

varname:(WORD) Name to be placed into the contentslist.
varnamelist:(LIST) List of names to be placed into the contentslist.

namelist [foo bar]
[[ ] [foo bar]]
# § K • PLLIST

contentslist PLLIST pname (library procedure)

Contentslist PLLIST pinamelist

Outputs a contentslist consisting of two empty lists followed by a list of the name or names given as input. This is useful in conjunction with workspace control procedures that require a contents list as input.

Note: All procedures whose input is indicated as "contentslist" will accept a single word (taken as a procedure name), a list of words (taken as names of procedures), or a list of three lists as described under CONTENTS above.

contentslist: (LIST) List of 2 lists containing an empty procedure list followed by a list of variable names.

pname: (WORD) Property list name to be placed into the contents list.

pinamelist: (LIST) List of property list names to be placed into the contentslist.

pllist [foo bar]

[[[]] [foo bar]]
PO
POALL
POPS
PONS
POPLS
PON
POPL
POT
POTS

* INSPECTION
* INSPECTION
* INSPECTION
* LOGO:0
PO contentslist

Command that prints out to the write stream the definitions of all procedures, variables, and property lists named in the input contentslist.

contentslist:(LIST) List of three lists containing procedures, variables and properties.

Example:

to xxx
print "Hello
end
po contents
to xxx
print "Hello
end

# $ K * PO

* PO
$ PO
K PO
* LOGO:0
POALL (library procedure)

Command that prints out all unburied definitions in the workspace. Abbreviates PO CONTENTS.

Example:

to xxx
print "Hello
end
poall
to xxx
print "Hello
end
POPS

POPS  (library procedure)

Command that prints out the definitions of all unburi procedures in the workspace. Abbreviates PO PROCEDURES.

Example:

to xxx
print "Hello
end
pops
to xxx
print "Hello
end

# $ K  * POPS

* POPS
$ POPS
K POPS
* LOGO:0
PONS (library procedure)

Command that prints out the definitions of all unburied variables in the workspace. Abbreviates \textsc{po names}.

Example:

```latex
\texttt{make "foo 1}
\texttt{pons}
\texttt{Make "foo 1}
```
POPLS (library procedure)

Command that prints out the contents of all unburied property lists in the workspace. Abbreviates PO PLISTS.

Example:

pprop "plist1 "pl 1
popls
fprop "plist1 "pl 1
PON varname   (library procedure)
PON varnamelist

Command that prints out the definitions of the named variable(s). Abbreviates PON NAMELIST varname(list).

varname (WORD) Name to be printed out.
varnamelist (LIST) List of names to be printed out.

Example:

make "foo 1
make "bar 2
pon [foo bar]
Make "foo 1
Make "bar 2
POPL * POPL

POPL pname (library procedure)
POPL pnamelist

Command that prints out the definitions of the named property list(s). Abbreviates PO PILLIST pnamelist.

pname:(WORD) Property list name to be printed out.

pnamelist:(LIST) List of property list names to be printed out.

Example:

pprop "plistl "pl l
po pl [plistl]
Fprop "plistl "pl l
POT contentslist

Command that prints out the title lines of the named procedures and the definitions of the named variables and property lists. For property lists, the entire list is shown on one line instead of as a series of PPROP instructions as in PO.

contentslist:(LIST) List of three lists containing procedures, variables and properties.

Example:

to foo
end
; Print Out Titles of Procedures
pot procedures
  to foo
POTS (library procedure)

Command that prints the title lines of all unburied procedures in the workspace. Abbreviates POTT PROCEDURES.

Example:

to foo
end
pots
to foo

* POTS
# POTS
$ POTS
^ POTS
* LOGO:0
WORKSPACE CONTROL

ERASE
ERALL
ERPS
ERN
ERPLS
ERN
ERPI
BURY
BURYALL
BURYNAME
UNBURY
UNBURYALL
UNBURYNAME
TRACE
UNTRACE
STEP
UNSTEP
EDIT
EDALL
EDPS
EDNS
EDPLS
EDN
EDPI
SAVE
SAVEL
LOAD
NOSTATUS
STATUS

\* WORKSPACE CONTROL
\$ WORKSPACE CONTROL
\% WORKSPACE CONTROL
\* LOGO:0
ERASE

ERASE contentslist
ER contentslist

Command that erases from the workspace the procedures, variables, and property lists named in the input. Primitive procedures may not be erased unless the variable REDEFP has the value TRUE.

contentslist: (LIST) List of three lists containing procedures, variables and properties.

Example:

to foo
end
to bar
end
pots
to foo
to bar
erase [foo]
pots
to bar
ERALL (library procedure)

Command that erases all unburied procedures, variables, and property lists from the workspace. Abbreviates ERASE CONTENTS.

Example:

to foo
end
to bar
end
pots
to foo
to bar
erall
pots

* ERALL
* ERALL
* ERALL
* LOGO:0
ERPS (library procedure)

Command that erases all unburied procedures from the workspace. Abbreviates ERASE PROCEDURES.

Example:

to foo
end
to bar
end
pots
to foo
  to bar
erps
pots

* ERPS
$ ERPS
& ERPS
* LOGO:0
ERNS (library procedure)

Command that erases all unburied variables from the workspace. Abbreviates ERASE NAMES.

Example:

make "foo 1
make "bar 2
pons
Make "bar 2
Make "foo 1
erns
pons
ERPLS (library procedure)

Command that erases all unburied property lists from the workspace. Abbreviates `ERASE-PLISTS`.

Example:

```plaintext
pprop "plst1" "pl 1
popls
Fpprop "plst1" "pl 1
erpls
popls
```
ERN

ERN varname (library procedure)
ERN varnamelist

Command that erases from the workspace the variable(s) named in the input. Abbreviates ERASE NAMELIST.

varname (WORD) Name to be erased.
varnamelist (LIST) List of names to be erased.

Example:

make "foo 1
make "bar 2
pons
Make "bar 2
Make "foo 1
ern [foo]
pons
Make "bar 2
ERPL plname (library procedure)
ERPL plnamelist

Command that erases from the workspace the property list(s) named in the input. Abbreviates ERASE PLIST
plname(list).

plname:(WORD) Property list name to be erased.
plnamelist:(LIST) List of property list names to be erased.

Example:

pprop "pplist1 "pl 1
pprop "pplist2 "pl 2
ppopls
Pprop "pplist1 "pl 1
Pprop "pplist2 "pl 2
erpl [pplist2]
popls
Pprop "pplist1 "pl 1
BURY contentslist

Command that buries the procedures, variables, and property lists named in the input. A buried item is not included in the lists output by CONTENTS, PROCEDURES, NAMES, and PLISTS, but is included in the list output by BURIED. By implication, buried things are not printed by POALL or saved by SAVE.

contentslist:(LIST) List of three lists containing procedures, variables and properties.

Example:

to foo
Print [Here I am]
end
to bar
Print [Here I go]
end
pots
to bar
to foo
bury [[foo] [] []]
pots
to bar
foo
Here I am
# $ K • BURYALL

BURYALL (library procedure)

Command that abbreviates BURY CONTENTS.

Example:

to foo
Print [Here I am]
end
to bar
Print [Here I go]
end
pots
to bar
to foo
buryall
pots
foo
Here I am
bar
Here I go
**$K• BURYNAME**

BURYNAME varname (library procedure)
BURYNAME varnamelist

Command that abbreviates **BURY NAMELIST** varname(list).

varname (WORD) Name to be buried.
varnamelist (LIST) List of names to be buried.

Example:

```
make "foo 1
make "bar 2
pons
Make "bar 2
Make "foo 1
buryname [foo]
pons
Make "bar 2
show :foo 1
show :bar 2
```

---

$ BURYNAME
$ BURYNAME
$ BURYNAME
$ BURYNAME
$ LOGO:0
UNBURY

UNBURY contentslist

Command that unburies the procedures, variables, and property lists named in the input. That is, the named items will be returned to view in CONTENTS, etc.

contentslist: (LIST) List of three lists containing procedures, variables and properties.

Example:

make "foo 1
make "bar 2
pons
Make "bar 2
Make "foo 1
buryname [foo]
pons
Make "bar 2
unbury [[] [foo] []]
pons
Make "bar 2
Make "foo 1

# $K • UNBURY

$ UNBURY

K UNBURY

* LOGO:0
UNBURYALL (library procedure)

Command that abbreviates UNBURY BURIED.

Example:

make "foo 1
make "bar 2
pons
Make "bar 2
Make "foo 1
burynam [foo]
pons
Make "bar 2
unburyall
pons
Make "bar 2
Make "foo 1

* UNBURYALL
* UNBURYALL
* UNBURYALL
* LOGO:0
UNBURYNAME

UNBURYNAME varname (library procedure)
UNBURYNAME varnamelist

Command that abbreviates UNBURY NAMLIST varname(list).

varname (WORD) Name to be unburied.
varnamelist (LIST) List of names to be unburied.

Example:

make "foo 1
make "bar 2
pons
Make "bar 2
Make "foo 1
buryname [foo]
pons
Make "bar 2
unbury [foo]
pons
Make "bar 2
Make "foo 1

$ UNBURYNAME
$ UNBURYNAME
$ UNBURYNAME
$ UNBURYNAME
* LOGO:0
TRACE contentslist

Command that marks the named items for tracing. A message is printed whenever a traced procedure is invoked, giving the actual input values, and whenever a traced procedure executes a STOP or OUTPUT command. A message is printed whenever a new value is assigned to a traced variable using MAKE. A message is printed whenever a new property is given to a traced property list using PPROP. See also the UNTRACE command and Trace Button.

Example:

to myprog :a
  print :a
end
myprog "Hello
Hello
trace "myprog
myprog "Hello
  ( myprog "Hello )
Hello
untrace "myprog
myprog "Hello
Hello

* TRACE
$ TRACE
K TRACE
* LOGO:0
UNTRACE

UNTRACE contentslist

Command that turns off tracing for the named items. See also the TRACE command and Trace Button.

contentslist; (LIST) List of three lists containing procedures, variables and properties.

Example:

t to myprog a
print :a
end
myprog "Hello
Hello
trace "myprog
myprog "Hello ( myprog "Hello )
Hello
untrace "myprog
myprog "Hello
Hello
Hello
# $K • **STEP**

**STEP contentslist**

Command that marks the named items for stepping. Whenever a stepped procedure is invoked, each instruction line in the procedure body is printed before being executed, and Logo waits for the user to type a newline at the terminal. A message is printed whenever a stepped variable name is "shadowed" because a local variable of the same name is created either as a procedure input or by the `LOCAL` command. See also the `STEP` command and `Step Button`.

**contentslist:** (LIST) List of three lists containing procedures, variables and properties.

Example:

```logo
to myprog
fd 10
rt 90
fd 20
lt 90
end
myprog
step "myprog
<Each line of myprog will be displayed and will wait for OK to continue to next line>
myprog
unstep "myprog
myprog
```

* **STEP**
* **STEP**
* **STEP**
* **LOGO:** 0
UNSTEP contentslist

Command that turns off stepping for the named items. See also the \texttt{STEP} command and \texttt{Step Button}.

\texttt{contentslist}:(LIST) List of three lists containing procedures, variables and properties.

Example:

to myprog
  fd 10
  rt 90
  fd 20
  lt 90
end
myprog
step "myprog"
<Each line of myprog will be displayed and will wait for OK to continue to next line>
myprog
unstep "myprog"
myprog

\# \$ \& \* UNSTEP

\$ UNSTEP
\& UNSTEP
\* UNSTEP
\* LOGO:0
EDIT contentslist
ED contentslist
(EDIT)
(ED)

Command that edits the definitions of the named item(s), using the Logo editor. See also Editor for details of how the editor works.

contentslist:(LIST) List of three lists containing procedures, variables and properties.

Example:

to "myprog
print "Hello
end
myprog
Hello
edit "myprog
<change Hello to Hello and exit editor>
myprog
Hello
EDALL (library procedure)

Command that abbreviates EDIT CONTENTS.

Example:

to myprog  
print :myarg  
end  
make "myarg "Hello  
myprog  
Hello  
editall  
<change Hello to Hello and exit editor>  
myprog  
Hello
EDPS (library procedure)

Command that abbreviates EDIT PROCEDURES.

Example:

to "myprogl
print "Hello
end
to "myprog2
print "Bye
end
myprogl
Hello
myprog2
Bye

<change Hello to Hello, change Bye to Bye and exit editor>

myprogl
Hello
myprog2
Bye
EDNS (library procedure)

Command that abbreviates EDIT NAMES.

Example:

to myprog
  print :myarg
  end
make "myarg "Hello
myprog
Hello

deitall
<change Hello to Hello and exit editor>
myprog
Hello
EDPLS

EDPLS (library procedure)

Command that abbreviates EDIT PLISTS.

Example:

pprop "plstl "pl 1
popls
Pprop "plstl "pl 1
edpls
<change 1 to 2 and exit editor>
popls
Pprop "plstl "pl 2
EDN

EDN varname (library procedure)
EDN varnamelist

Command that abbreviates EDIT NAMELIST varname(list).

varname (WORD) Name to be edited.
varnamelist (LIST) List of names to be edited.

Example:

to myprog
  print :myarg
end
make "myarg "Hello
myprog
Hello
edn "myarg
<change Hello to Hello and exit editor>
myprog
Hello

* EDN
* EDN
* EDN
* LOGO:0
EDPL

EDPL pname (library procedure)
EDPL plnamelist

Command that abbreviates EDIT PLIST pname(list).

pname:(WORD) Property list name to be edited.
plnamelist:(LIST) List of property list names to be edited.

Example:

pprop "plistl "pl 1
popls
Pprop "plistl "pl 1
edpl "plistl
<change 1 to 2 and exit editor>
popls
Pprop "plistl "pl 2
SAVE

SAVE filename

Command that saves the definitions of all unburied procedures, variables, and property lists in the named file.

filename: (WORD) Name of file to save Logo’s workspace to.

Save is equivalent to:

to save :filename
local "oldwriter
make "oldwriter writer
openwrite :filename
setwrite :filename
poall
setwrite :oldwriter
close :filename
end

Example:

to myprogl
print "Hello1
end
to myprog2
print "Hello2
end
pots
to myprogl
to myprog2
save "myprogs.lgo
erall
pots
load "myprogs.lgo
pots
to myprogl
to myprog2
SAVEL contentslist filename (library procedure)

Command that saves the definitions of the procedures, variables, and property lists specified by "contentslist" to the file named "filename".

contentslist: (LIST) List of three lists containing procedures, variables and properties.
filename: (WORD) Name of file to save the contentslist to.

Example:

to myprog1
print "Hello1"
end
to myprog2
print "Hello2"
end
pots
to myprog1
to myprog2
savel [([myprog1] [] [])] "myprogs.lgo
erall
pots
load "myprogs.lgo
pots
to myprog1
LOAD

LOAD filename

Command that reads instructions from the named file and executes them. The file can include procedure definitions with TO, and these are accepted even if a procedure by the same name already exists. If the file assigns a list value to a variable named \texttt{STARTUP}, then that list is run as an instruction list after the file is loaded.

It is important to understand that the file is executed, not copied, into the workspace:

- If Logo sees a \texttt{TO} then it will define that procedure (the code within a definition is not executed).
- If Logo sees a \texttt{MAKE} it will make that name.
- If Logo sees code is outside of a procedure definition it will be executed in the order seen.

\texttt{filename}:(WORD) Name of file to execute.

Example:

to myprog1
print "Hello1
end
to myprog2
print "Hello2
end
pots
to myprog1
to myprog2
save "myprogs.1go
errall
pots
load "myprogs.1go
pots
to myprog1
to myprog2

\* LOAD
\$ LOAD
\* LOAD
\* LOGO:0
NOSTATUS

This command has the same effect as hitting the NOSTATUS BUTTON. That is, it kills the popup status window. See also STATUS command.

Example:

status
nostatus
STATUS

This command has the same effect as hitting the STATUS BUTTON. That is, it pops up the status window. See also NOSTATUS command.

Example:

status
nostatus
CONTROL STRUCTURES

Note: in the following descriptions, an "instruction list" can be a list or a word. In the latter case, the word is parsed into list form before it is run. Thus, RUN READWORD or RUN READLIST will work. The former is slightly preferable because it allows for a continued line (with ~) that includes a comment (with :) on the first line.

CONTROL COMMANDS
TEMPLATE-BASED ITERATION
CONTROL COMMANDS

RUN
RUNRESULT
REPEAT
REPCOUNT
IF
IFELSE
TEST
IFTRUE
IFFALSE
TRUE
FALSE
STOP
OUTPUT
CATCH
THROW
ERROR
PAUSE
CONTINUE
YIELD
NOYIELD
EVENTCHECK
SETCURSORWAIT
HALT
WAIT
SETTIMER
CLEARTIMER
BYE
MAYBEOUTPUT
IGNORE
-
FOR
DO.WHILE
WHILE
DO.UNTIL
UNTIL
GOTO
TAG
# $K • RUN

**RUN instructionlist**

Command or operation that runs the Logo instructions in the input list; outputs if the list contains an expression that outputs.

**Instructionlist:** (LIST) List of Logo instructions to run.

Example:

```logo
make "thingstodo [print]
make "thingstodo lput "Hello :thingstodo
run :thingstodo
Hello
```

RUNRESULT instructionlist

Runs the instructions in the input; outputs an empty list if those instructions produce no output, or a list whose only member is the output from running the input instructionlist. Useful for inventing command-or-operation control structures:

local "result
make "result runresult [something]
if emptyp :result [stop]
output first :result

Instructionlist:(LIST) List of Logo instructions to run with a result.

Example:

make "thingstodo [first [1 2 3]]
make "answer runresult :thingstodo
show :answer
[1]
# $ K • REPEAT

REPEAT num instructionlist

Command that runs the instructionlist repeatedly, num times. See also REPCount.

num: (INTEGER) Number of times to repeat instructionlist.
Instructionlist: (LIST) List of Logo instructions to run repeatedly.

Example:

repeat 3 [print (list "This "is "loop repcount")]
This is loop 1
This is loop 2
This is loop 3
# $ K $ * REPCOUNT

**count REPCOUNT**

This operation may be used only within the range of a **REPEAT** command. It outputs the number of repetitions that have been done, including the current one. That is, it outputs 1 the first time through, 2 the second time, and so on.

**count**: (INTEGER) Current value of the REPEAT counter.

Example:

```
repeat 3 [print (list "This "Is "loop repcount)])
This Is loop 1
This Is loop 2
This Is loop 3
```
thing IF tf instructionlist
thing (IF tf instructionlist1 instructionlist2)

Command or operation where:
If the first input has the value TRUE, then IF runs the second input.
If the first input has the value FALSE, then IF does nothing.
If given a third input, IF acts like IFFALSE.
It is an error if the first input is not either TRUE or FALSE.

For compatibility with earlier versions of Logo, if an IF instruction is not enclosed in parentheses, but the first thing on
the instruction line after the second input expression is a literal list (i.e., a list in square brackets), the IF is treated as if
it were IFFALSE, but a warning message is given. If this aberrant IF appears in a procedure body, the warning is given
only the first time the procedure is invoked in each Logo session.

thing:(THING) Outputs whatever thing instructionlist, instructionlist1 or instructionlist2 outputs if executed.

tf:(BOOLEAN) Expression to test for truth.
instructionlist:(LIST) List of Logo instructions to run if tf is TRUE.
instructionlist1:(LIST) List of Logo instructions to run if tf is TRUE.
instructionlist2:(LIST) List of Logo instructions to run if tf is FALSE.

Example:
if l=1 [print [Yes it is true]]
Yes it is true

Example (where ifelse outputs):
to max :a :b
output (if :a > :b [:a] [:b])
end
show max 1 2
2
# $ K * IFELSE

thing IFELSE if instructionlist1 instructionlist2

Command or operation where if the first input has the value TRUE, then IFELSE runs the second input. If the first input has the value FALSE, then IFELSE runs the third input. IFELSE outputs a value if the instructionlist contains an expression that outputs a value.

thing:(THING) Outputs what ever thing instructionlist1 or instructionlist2 outputs.

tf:(BOOLEAN) Expression to test for truth.
instructionlist:(LIST) List of Logo instructions to run if tf is TRUE.
instructionlist1:(LIST) List of Logo instructions to run if tf is TRUE.
instructionlist2:(LIST) List of Logo instructions to run if tf is FALSE.

Example:

ifelse l=1 [print [Yes it is true]] [print [No it is false]]
Yes it is true
ifelse l=0 [print [Yes it is true]] [print [No it is false]]
No it is false

Example (where ifelse outputs)

to max :a :b
output ifelse :a > :b [:a] [:b]
end

show max 1 2
2
# $K TEST

TEST if

Command that remembers its input, which must be TRUE or FALSE, for use by later IFTRUE or IFFALSE instructions. The effect of TEST is local to the procedure in which it is used; any corresponding IFTRUE or IFFALSE must be in the same procedure or a subprocedure.

if:(BOOLEAN) Expression to test for truth.

Example:

to mytest :arg
test l=arg
print [Do this]
print [Do that]
iftrue [print [arg was the number one]]
iffalse [print [arg was NOT the number one]]
end
mytest 1
Do this
Do that
arg was the number one
IFTRUE

IFTRUE instruction list
IFT instruction list

Command that runs its input if the most recent TEST instruction had a TRUE input. The TEST must have been in the same procedure or a superprocedure.

instructionlist: (LIST) List of Logo instructions to run if earlier TEST was TRUE.

Example:

to mytest :arg
test 1=:arg
print [Do this]
print [Do that]
iftrue [print [arg was the number one]]
iffalse [print [arg was NOT the number one]]
end
mytest 1
Do this
Do that
arg was the number one
IFFALSE

IFFALSE instructionlist
IFFI instructionlist

Command that runs its input if the most recent TEST instruction had a FALSE input. The TEST must have been in the same procedure or a superprocedure.

instructionlist: (LIST) List of Logo instructions to run if earlier TEST was FALSE.

Example:

to mytest :arg
test 1=:arg
print [Do this]
print [Do that]
iftrue [print [arg was the number one]]
iffalse [print [arg was NOT the number one]]
end
mytest 0
Do this
Do that
arg was NOT the number one
# $K^* TRUE

TRUE (special form)

This is a special word to indicate a positive condition.

Example:

```
show l=1
true
if "true [print [True is always true]]
True is always true
```

* TRUE
$ TRUE
$ TRUE
* LOGO:0
FALSE

FALSE  (special form)

This is a special word to indicate a negative condition.

Example:

> show 1=0

false

ifelse "false [print [We can not get here]] [print [False is always false]]

False is always false
STOP

Command that ends the running of the procedure in which it appears. Control is returned to the context in which that procedure was invoked. The stopped procedure does not output a value.

Example:

to myprog :arg
print [Before Stop]
if 1=arg [stop]
print [After Stop]
end

myprog 1
Before Stop
myprog 2
Before Stop
After Stop

° STOP
§ STOP
★ STOP
* LOGO:0
OUTPUT

OUTPUT value

Command that ends the running of the procedure in which it appears. That procedure outputs the value "value" to the context in which it was invoked. Don't be confused: OUTPUT itself is a command, but the procedure that invokes OUTPUT is an operation.

value:(THING) Any Logo thing to output.

Example:

to myprog
output [This is the output]
end
show myprog
[This is the output]
CATCH tag instructionlist
value CATCH tag instructionlist

Command or operation that runs its second input. Outputs if that instructionlist outputs. If, while running the
instructionlist, a THROW instruction is executed with a tag equal to the first input (case-insensitive comparison), then
the running of the instructionlist is terminated immediately. In this case the CATCH outputs if a value input is given to
THROW. The tag must be a word.

If the tag is the word ERROR, then any error condition that arises during the running of the instructionlist has the effect
of THROW "ERROR" instead of printing an error message and returning to toplevel. The CATCH does not output if an
error is caught. Also, during the running of the instructionlist, the variable ERRACT is temporarily unbound. (If there is
an error while ERRACT has a value, that value is taken as an instructionlist to be run after printing the error message.
Typically the value of ERRACT, if any, is the list [PAUSE].)

value: (THING) Value the THROW threw back.
tag: (WORD) Name of tag to THROW to.
instructionlist: (LIST) List of Logo instructions to run while under protection of CATCH.

Example:

to myprog2
  print [Before throw]
  throw "tag1
  print [We never get here because we THROW back]
  end
to myprog1
  catch "tag1 [myprog2]
  print [I'm back]
  end
myprog1
Before throw
I'm back
**THROW**

**THROW tag**

(THROW tag value)

Command that must be used within the scope of a CATCH with an equal tag. Ends the running of the instruction list of the CATCH. If THROW is used with only one input, the corresponding CATCH does not output a value. If THROW is used with two inputs, the second provides an output for the CATCH.

**value:** (THING) Value to THROW back to for CATCH to output.

**tag:** (WORD) Name of tag to THROW to.

THROW *TOLEVEL can be used to terminate all running procedures and interactive pauses, and return to the toplevel instruction prompt. Typing the system interrupt character (normally ^C) has the same effect (or HALT button in MSWLogo).

THROW *ERROR can be used to generate an error condition. If the error is not caught, it prints a message (THROW *ERROR) with the usual indication of where the error (in this case the THROW) occurred. If a second input is used along with a tag of ERROR, that second input is used as the text of the error message instead of the standard message. Also, in this case, the location indicated for the error will be, not the location of the THROW, but the location where the procedure containing the THROW was invoked. This allows user-defined procedures to generate error messages as if they were primitives. Note: in this case the corresponding CATCH *ERROR, if any, does not output, since the second input to THROW is not considered a return value.

THROW *SYSTEM immediately leaves Logo, returning to the operating system, without printing the usual parting message and without deleting any editor temporary file written by EDIT.

Example (There may be a bug in throw here):

to myprog2
  print [Before throw]
  (throw "tag1 [We need to get back])
  print [We never get here because we THROW back]
end

to myprog1
  show catch "tag1 [myprog2]
  print [I'm back]
end

myprog1
Before throw
[We need to get back]
I'm back
errorlist ERROR

Outputs a list describing the error just caught, if any. If there was not an error caught since the last use of ERROR, the empty list will be output. The error list contains four members: an integer code corresponding to the type of error, the text of the error message, the name of the procedure in which the error occurred, and the instruction line on which the error occurred. (See also the list of ERROR CODES)

errorlist: (LIST) List containing the code, the message, the name of procedure and instruction.

Example:

to myprog
fd 1000
end
fence
catch "error [myprog]
show error
[3 [turtle out of bounds] myprog [fd 1000]]
PAUSE

Command or operation that enters an interactive pause. The user is prompted for instructions, as at toplevel, but with a prompt that includes the name of the procedure in which PAUSE was invoked. Local variables of that procedure are available during the pause. PAUSE outputs if the pause is ended by a \texttt{CONTINUE} with an input.

If the variable \texttt{ERRACT} exists, and an error condition occurs, the contents of that variable are run as an instructionlist. Typically \texttt{ERRACT} is given the value \texttt{PAUSE} so that an interactive pause will be entered on the event of an error. This allows the user to check values of local variables at the time of the error.

\texttt{value:(THING) Value the CONTINUE sent back.}

Example:

to myprog
repeat 180 [rt 2 if 90=repcount [pause]]
print "Done"
end
myprog

Pausing...myprog

\texttt{<Enter SHOW HEADING in Pause-Box and hit OK>}

180

\texttt{<Enter CO in Pause Box and hit OK>}

Done
# $ K  • CONTINUE

CONTINUE value
CO value
(CONTINUE)
(CO)

Command that ends the current interactive pause, returning to the context of the PAUSE invocation that began it. If CONTINUE is given an input, that value is used as the output from the PAUSE. If not, the PAUSE does not output.

value: (THING) Value to send back to for PAUSE to output.

Exceptionally, the CONTINUE command can be used without its default input and without parentheses provided that nothing follows it on the instruction line.

Example:

to myprog
repeat 180 [rt 2 if 90=repcount {pause}]
print "Done"
end
myprog
Pausing...myprog
<Enter SHOW HEADING in Pause-Box and hit OK>
180
<Enter CONTINUE in Pause Box and hit OK>
Done
YIELD

YIELD tells LOGO to let other programs use the computer while LOGO is working. NOYIELD tells LOGO to not let other programs use the computer while LOGO is working. The default is to Yield. Note that the commander window itself is like another program. That is, if you have issued a NoYield, the commander will lose control (That means the Hall Button won't work) until a YIELD is issued or LOGO is idle again. The reason the no yielding is available is that LOGO runs faster if it can keep the computer all to itself. See also NoYield command.

You can achieve the best of both worlds (performance and yielding) by doing (and understanding) the following. Let's say that you have some large multi-nested loop in your code. The most inner loop is where most of the work is done. However, there is no need to YIELD all through the inner loop.

Example:

We have 3 cases:

Case 1 (User in control for 10,000 operations, lower performance):

```logol
yield
repeat 100 -
  [~
    repeat 100 -
      [~
        (work to be done)~
      ]~
  ]
```

Case 2 (User out of control for 10,000 operations, good performance):

```logol
noyield
repeat 100 -
  [~
    repeat 100 -
      [~
        (work to be done)~
      ]~
  ]
```

Case 3 (User out of control for 100 operations, still good performance):

```logol
* YIELD
$ YIELD
& YIELD
* LOGO:0
```
repeat 100~
  [~
  noyield~
repeat 100 ~
  [~
    (work to be done)~
    ]~
yield~
}
NOYIELD

NOYIELD

NOYIELD tells LOGO to not let other programs use the computer while LOGO is working. The default is to Yield. See also Yield command.

Example:

See Yield command example.
EVENTCHECK

The EVENTCHECK command is like call Yield followed by NoYield command immediately after. It basically checks to see if any events are waiting to be processed. Some commands that issue "callbacks" run in a Non-Yielding state. If the callback code is short and quick you won't see any problems. But if it's long the user will have no control during this period. If you sprinkle a few of these around the user will not loose control for such a "long" period of time.

This command can be used as an alternative to placing YIELD and NOYIELD commands around compute intensive loops. You can encapsulate the whole procedure in a NOYIELD and YIELD pair then you can place EVENTCHECK's appropriately in your loops.

Example:

mouseon [repeat 72 [repeat 4 [fd 100 rt 90] rt 5]] [] [] [][]
<now click the mouse in MSWLogo Screen>
<note that while it is drawing you cannot HALT it, because mouse callbacks do not yield>
mouseon [repeat 72 [repeat 4 [fd 100 rt 90] rt 5 eventcheck]] [] [] [] []
<now click the mouse in MSWLogo Screen>
<note that while it is drawing you CAN now HALT>
SETCURSORWAIT

This will set the cursor to the familiar hourglass shape. Its purpose is to indicate two things. One is that the operation about to take place will take some time. And second that during that time the user does not have control of Windows (not yielding). This function only works when not yielding. Once yielding the appropriate cursor will be used on the next event involving the cursor (like moving the mouse). This means that you must issue NOYIELD before each SETCURSORWAIT. If you decide to YIELD momentarily during a computation you must SETCURSORWAIT again. In other words if you wish to use SETCURSORWAIT it should always be paired up with a NOYIELD (just before it).

Example:

repeat 100-
  [-
    noyield-
    setcursorwait-
    repeat 100 -
      [-
        (work to be done)-
      ]-
    setcursornowait-
    yield-
  ]-
}
HALT

The halt command is like hitting the HALT BUTTON but is under Logo control.

Example:

repeat 1000 [fd 100 bk 100 rt 1]
Hit HALT BUTTON while it is drawing
Stopping...
WAIT

WAIT time

Command that delays further execution for time 1/60ths of a second. Also causes any buffered characters destined for the terminal to be printed immediately. WAIT 0 can be used to achieve this buffer flushing without waiting.

time: (INTEGER) Time to wait in 1/60ths of second.

Example:

wait 60
**SETTIMER**

SETTIMER id delay callback

This command sets up a timer identified by id (1-31) to call the logo commands in callback in every delay (milliseconds) of time. It will continue to "fire" every delay until you issue a **CLEARTIMER** id. You can generate as much trouble as you can power with this command.

id:(INTEGER) Id of Timer (1-31) to set.
delay:(INTEGER) Timer interval to delay between firing.
callback:(LIST) Is a list of instructions to execute each time the timer fires.

You can clear the timer at any time. This means that you can clear the timer in the callback (handler) itself (a single fire). You can also issue a timer at any time, including in the callback (handler). Halting will clear all timers. The behavior of timers from 1-16 are slightly different than 17-31. Timers 1-16 will not allow the callback code to Yield. Where as timers 17-31 will Yield.

You must be sure that you are handling (servicing) the timer request faster than they are coming in. If you do not, you will eventually "Stack-Overflow" and MSWLogo will shutdown.

You must also be sure that the callback code is error free. If not, you may generate more "OK" boxes then you have ever seen and eventually "Stack OverFlow".

It's a good idea to save your code frequently to disk when developing code that uses timers. These are not really bugs, you're just taping directly into Windows and you have to be careful.

You can "block" all timers from firing (interrupting) in any code, including the callback (handler), by issuing the **NOYIELD** command. You can "restore" firing by issuing the **YIELD** command. Blocking does not loose any "events" (firings), they are queued up. So don't "block" to long.

Each timer can have different callbacks or the same callbacks.

Example:

```
settimer 1 200 [setpen color (list random 256 random 256 random 256)]
<note MSWLogo returns here and the timer is firing (changing the pen randomly)>
repeat 72 [repeat 4 [fd 100 rt 90] rt 5]
cleartimer 1
```

* SETTIMER
* SETTIMER
* SETTIMER
* LOGO:0
# $K• CLEARTIMER

CLEARTIMER id

Command that clears the timer set by SETTIMER and identified by id (1-31).

id: (INTEGER) Id of Timer (1-31) to clear.

Example:

settimer 1 200 [setpencolor (list random 256 random 256 random 256)]
<note MSWLogo returns here and the timer is firing (changing the pen randomly)>
repeat 72 [repeat 4 [fd 100 rt 90] rt 5]
cleartimer 1

* CLEARTIMER
$ CLEARTIMER
% CLEARTIMER
* LOGO:0
# $K • BYE

BYE

Command that exits from Logo; returns to the operating system.

Example:

bye
.MAYBEOUTPUT

.MAYBEOUTPUT value (special form)

Works like OUTPUT except that the expression that provides the input value might not output a value, in which case the effect is like STOP. This is intended for use in control structure definitions, for cases in which you don't know whether some expression produces a value.

value:(THING) Any Logo thing to maybe output.

Example:

to invoke :function [:inputs] 2
.maybeoutput apply :function :inputs
end

(invoke "print "a "b "c)
 a b c
print (invoke "word "a "b "c)
a bc

This is an alternative to RUNRESULT. It's fast and easy to use, at the cost of being an exception to Logo's evaluation rules. (Ordinarily, it should be an error if the expression that's supposed to provide an input to something doesn't have a value.)
### IGNORE

**value** (library procedure)

Command that does nothing. Used when an expression is evaluated for a side effect and its actual value is unimportant.

**value:** (THING) Any Logo thing to evaluate.

Example:

```logo
to myprog :arg
  print :arg
  output count :arg
end
myprog "Hello
Hello
I don't know what to do with 5
ignore myprog "Hello
Hello
```

---

* IGNORE
* IGNORE
* IGNORE
* IGNORE
* LOGO:0
newlist ` list (library procedure)

Outputs a list equal to its input but with certain substitutions. If a member of the input list is the word `,` (comma) then the following member should be an instructionlist that produces an output when run. That output value replaces the comma and the instructionlist. If a member of the input list is the word `@` (comma atsign) then the following member should be an instructionlist that outputs a list when run. The members of that list replace the `,` and the instructionlist.

**newlist:**(LIST) Processed output list.

**list:**(LIST) Input list to be processed.

Example:

```
show `[foo baz ,[bf [a b c]] garply ,@[bf [a b c]]]
[foo baz [b c] garply b c]
```
FOR

FOR forcontrol instructionlist (library procedure)

Command in which the first input must be a list containing three or four members: (1) a word, which will be used as the name of a local variable; (2) a word or list that will be evaluated as by Run to determine a number, the starting value of the variable; (3) a word or list that will be evaluated to determine a number, the limit value of the variable; (4) an optional word or list that will be evaluated to determine the step size. If the fourth element is missing, the step size will be 1 or -1 depending on whether the limit value is greater than or less than the starting value, respectively.

The second input is an instructionlist. The effect of FOR is to run that instructionlist repeatedly, assigning a new value to the control variable (the one named by the first element of the forcontrol list) each time. First the starting value is assigned to the control variable. Then the value is compared to the limit value. FOR is complete when the sign of (current - limit) is the same as the sign of the step size. (If no explicit step size is provided, the instructionlist is always run at least once. An explicit step size can lead to a zero-try FOR, e.g., FOR [1 1 0] ...) Otherwise, the instructionlist is run, then the step is added to the current value of the control variable and FOR returns to the comparison step.

forcontrol:(LIST) List containing the name (word), followed by 3 numbers, start value, end value, and increment.

instructionlist:(LIST) List of Logo instructions to execute for each loop.

Example:

for [i 2 7 1.5] [print :i]
  2
  3.5
  5
  6.5

* FOR
* FOR
* FOR
* LOGO:0
DO.WHILE instructionlist tfexpressionlist (library procedure)

Command that repeatedly evaluates the instructionlist as long as the evaluated tfexpressionlist remains TRUE. Evaluates the first input first, so the instructionlist is always run at least once. The tfexpressionlist must be an expressionlist whose value when evaluated is TRUE or FALSE.

instructionlist:(LIST) List of Logo instructions to do while expression is TRUE.
tfexpressionlist:(BOOLEAN) Expression list that outputs the truth as whether the while loop should continue.

Example:

make "i 0
do.while [make "i :i+1 print :i] [:i<3]  
  
1 
2 
3
# $ K * WHILE

**WHILE**

WHILE **tfexpressionlist instructionlist**  
(library procedure)

Command that repeatedly evaluates the **instructionlist** as long as the evaluated **tfexpression** remains TRUE. Evaluates the first input first, so the **instructionlist** may never be run at all. The **tfexpressionlist** must be an expressionlist whose value when evaluated is TRUE or FALSE.

**tfexpressionlist**: (BOOLEAN) Expression list that outputs the truth as whether the while loop should continue.  
**instructionlist**: (LIST) List of Logo instructions to execute while expression is TRUE.

Example:

```
make "i 0
while [:i<3] [make "i :i+1 print :i]
 1
 2
 3
```

* WHILE
¥ WHILE
¥ WHILE
¥ LOGO:0
# $K • DO.UNTIL

DO.UNTIL instructionlist tfexpressionlist    (library procedure)

Command that repeatedly evaluates the instructionlist as long as the evaluated tfexpressionlist remains FALSE. Evaluates the first input first, so the instructionlist is always run at least once. The tfexpressionlist must be an expressionlist whose value when evaluated is TRUE or FALSE.

instructionlist:(LIST) List of Logo instructions to do until expression is FALSE.
tfexpression:(BOOLEAN) Expression that outputs the truth as whether the do loop should continue.

Example:

make "i 0
do.until [make "$i :i+1 print :i] [:i>3]
1
2
3
4
UNTIL

UNTIL tfexpressionlist instructionlist  (library procedure)

Command that repeatedly evaluates the instructionlist as long as the evaluated tfexpressionlist remains FALSE. Evaluates the first input first, so the "instructionlist" may never be run at all. The tfexpressionlist must be an expressionlist whose value when evaluated is TRUE or FALSE.

tfexpression:(BOOLEAN) Expression that outputs the truth as whether the loop should continue.
instructionlist:(LIST) List of Logo instructions to do until expression is FALSE.

Example:

make "i 0
until [:i>3] [make "i :i+1 print :i]
1
2
3
4
GOTO

GOTO tag

This command can be used only inside a procedure. The input must be a WORD. A **Tag** of the same name must appear within the same procedure. Processing will continue following the tag. It is best to avoid GOTO's but in some instances such as "state machines" they can be very powerful.

**Tag** (WORD) Name of a tag to goto to with in procedure.

Example:

```logo
to states
tag "State1
  print "State1
  goto "State3
tag "State2
  print "State2
  goto "State4
tag "State3
  print "State3
  goto "State2
tag "State4
  print "State4
  stop
end
```

**States**

State1
State3
State2
State4
TAG

TAG tag

This command can be used only inside a procedure. It describes where a GOTO can jump to. The input must be a WORD.

tag:(WORD) Name of a tag to goto to within procedure.

Example:

See GOTO Example.
**TEMPLATE-BASED ITERATION**

The procedures in this section are iteration tools based on the idea of a "template." This is a generalization of an instruction list or an expression list in which "slots" are provided for the tool to insert varying data. Three different forms of template can be used.

The most commonly used form for a template is "explicit-slot" form, or "question mark" form.

Example:

```
show map [? * ?] [2 3 4 5]
[4 9 16 25]
```

In this example, the MAP tool evaluated the template [? * ?] repeatedly, with each of the members of the data list [2 3 4 5] substituted in turn for the question marks. The same value was used for every question mark in a given evaluation. Some tools allow for more than one datum to be substituted in parallel; in these cases the slots are indicated by ?1 for the first datum, ?2 for the second, and so on.

Example:

```
show (map [word ?1 ?2 ?1] [a b c] [d e f])
[ada be cfc]
```

If the template wishes to compute the datum number, the form (? 1) is equivalent to ?1, so (? ?1) means the datum whose number is given in datum number 1. Some tools allow additional slot designations, as shown in the individual descriptions.

The second form of template is the "named-procedure" form. If the template is a word rather than a list, it is taken as the name of a procedure. That procedure must accept a number of inputs equal to the number of parallel data slots provided by the template; the procedure is applied to all the available data in order. That is, if data ?1 through ?3 are available, the template "PROC is equivalent to [PROC ?1 ?2 ?3]."

Example:

```
show (map "word [a b c] [d e f])
[ad be cf]
```

to dotprod :a :b ; vector dot product
up apply "sum (map "product :a :b)
end

The third form of template is "named-slot" or "lambda" form. This form is indicated by a template list containing more than one element, whose first element is itself a list. The first element is taken as a list of names; local variables are created with those names and given the available data in order as their values. The number of names must equal the number of available data. This form is needed primarily when one iteration tool must be used within the template list of another, and the ? notation would be ambiguous in the inner template.

Example:

```
to matmul :ml :m2 [:tm2 transpose :m2] ; multiply two matrices
end
```

These iteration tools are extended versions of the ones in Appendix B of the book Computer Science Logo Style,
Volume 3: Advanced Topics by Brian Harvey [MIT Press, 1987]. The extensions are primarily to allow for variable numbers of inputs.

**APPLY**
**INVOKE**
**FOREACH**
**MAP**
**MAP.SE**
**FILTER**
**FIND**
**REDUCE**
**CROSSMAP**
**CASCADE**
**CASCADE.2**
**TRANSFER**
# APPLY

output APPLY template inputlist
 APPLY template inputlist

Command or operation that runs the template, filling its slots with the members of inputlist. The number of members in inputlist must be an acceptable number of slots for template. It is illegal to apply the primitive TO as a template, but anything else is okay. APPLY outputs what template outputs, if anything.

output: (THING) Any thing that template might output.

inputlist: (LIST) List of things to be applied to.
template: (WORD) Template to be applied.

Example:

show apply "sum [1 2 3]
  6

$ APPLY
$ APPLY
$ APPLY
$ APPLY
* LOGO:0
# $ K * INVOLVE

output INVOLVE template input  (library procedure)
INVOLVE template input
output (INVOLVE template input1 input2 ...)
(INVOLVE template input1 input2 ...)

Command or operation that is like APPLY except that the inputs are provided as separate expressions rather than in a list.

output: (THING) Any thing that template might output.

input: (LIST) Thing to be applied to.
Input1: (LIST) First thing to be applied to.
Input2: (LIST) Second thing to be applied to.
template: (WORD) Template to be applied.

Example:

show (invoke "sum 1 2 3)

6
# $ K • FOREACH

FOREACH datalist template list  (library procedure)
(FOREACH datalist1 datalist2 ... template list)

Command that evaluates the template list repeatedly, once for each element of the datalist. If more than one data list are given, each of them must be the same length. (The data inputs can be words, in which case the template is evaluated once for each character.

datalist:(LIST) List of things to be operated upon.
datalist1:(LIST) First List of things to be operated upon.
datalist2:(LIST) Second List of things to be operated upon.
template list:(LIST) List of templates to be executed.

In a template, the symbol ?REST represents the portion of the data input to the right of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then ?REST would be replaced by [C D E]. If multiple parallel slots are used, then (?REST 1) goes with ?1, etc.

In a template, the symbol # represents the position in the data input of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then # would be replaced by 2.

Example:

foreach [a b c d] [print (se "index # "value ? "rest ?rest)])
  index 1 value a rest b c d
  index 2 value b rest c d
  index 3 value c rest d
  index 4 value d rest

* FOREACH
* FOREACH
* FOREACH
* LOGO:0
mapped MAP template1ist data    (library procedure)
mapped (MAP template1ist data1 data2 ...)  

Outputs a word or list, depending on the type of the data input, of the same length as that data input. (If more than one data input are given, the output is of the same type as data1.) Each element of the output is the result of evaluating the template list, filling the slots with the corresponding element(s) of the data input(s). (All data inputs must be the same length.) In the case of a word output, the results of the template evaluation must be words, and they are concatenated with WORD.

mapped:(THING) Mapped output that is of the same types as the first data input.

data:(LIST) List of data elements to be mapped.
data1:(LIST) First List of data elements to be mapped.
data2:(LIST) Second List of data elements to be mapped.
template1ist:(LIST) List of templates to apply mapping.

In a template, the symbol ?REST represents the portion of the data input to the right of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then ?REST would be replaced by [C D E]. If multiple parallel slots are used, then (?REST 1) goes with ?1, etc.

In a template, the symbol # represents the position in the data input of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then # would be replaced by 2.

Example:

show (map "list [a b c] [d e f])
[[a d] [b e] [c f]]
mapped MAP_SE templatelist data (library procedure)
mapped (MAP_SE templatelist data1 data2 ...)

Outputs a list formed by evaluating the templatelist repeatedly and concatenating the results using SENTENCE. That is, the members of the output are the members of the results of the evaluations. The output list might, therefore, be of a different length from that of the data input(s). (If the result of an evaluation is the empty list, it contributes nothing to the final output.) The data inputs may be words or lists.

mapped:(THING) Mapped output that is of the same types as the first data input.

data:(LIST) List of data elements to be mapped.
data1:(LIST) First List of data elements to be mapped.
data2:(LIST) Second List of data elements to be mapped.
templatelist:(LIST) List of templates to apply mapping.

In a template, the symbol ?REST represents the portion of the data input to the right of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then ?REST would be replaced by [C D E]. If multiple parallel slots are used, then (?REST 1) goes with ?1, etc.

In a template, the symbol # represents the position in the data input of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then # would be replaced by 2.

Example:

show (map.se "list [a b c] [d e f])
[a d b e c f]
FILTER

filtered FILTER tftemplatealist data  (library procedure)

Outputs a word or list, depending on the type of the data input, containing a subset of the members (for a list) or characters (for a word) of the input. The template is evaluated once for each member or character of the data, and it must produce a TRUE or FALSE value. If the value is TRUE, then the corresponding input constituent is included in the output.

filtered:(THING) Filtered output of input data.

data:(LIST) List of data elements to be filtered.
tftemplatealist:(WORD) List of True-False templates to filter data with.

In a template, the symbol ?REST represents the portion of the data input to the right of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then ?REST would be replaced by [C D E].

In a template, the symbol # represents the position in the data input of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then # would be replaced by 2.

Example:

to vowelp :arg
if :arg="a" [output "true"]
if :arg="e" [output "true"]
if :arg="i" [output "true"]
if :arg="o" [output "true"]
if :arg="u" [output "true"]
output "false"
end
print filter "vowelp *elephant eea"
# $ k • FIND

found FIND ttemplate list data (library procedure)

Outputs the first constituent of the data input (the first member of a list, or the first character of a word) for which the value produced by evaluating the template with that constituent in its slot is TRUE. If there is no such constituent, the empty list is output.

found:(THING) Found output of input data.

data:(LIST) List of data elements to search in.
ttemplate list:(WORD) List of True-False templates to find data with.

In a template, the symbol ?REST represents the portion of the data input to the right of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then ?REST would be replaced by [C D E].

In a template, the symbol # represents the position in the data input of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then # would be replaced by 2.

Example:

to findl :arg
if :arg=1 [output "true]
output "false
end
show (find "findl [2 4 3 0])
[]
show (find "findl [2 1 3 0])
1
REDUCE

reduced REDUCE template data (library procedure)

Outputs the result of applying the template to accumulate the elements of the data input. The template must be a two-slot function. Typically it is an associative function name like *SUM*. If the data input has only one constituent (member in a list or character in a word), the output is that constituent. Otherwise, the template is first applied with ?1 filled with the next-to-last constituent and ?2 with the last constituent. Then, if there are more constituents, the template is applied with ?1 filled with the next constituent to the left and ?2 with the result from the previous evaluation. This process continues until all constituents have been used. The data input may not be empty.

found:(THING) Reduced output of input data.
data:(LIST) List of data elements to reduce.
template:(WORD) Template to perform reduction.

Note: If the template is, like SUM, the name of a procedure that is capable of accepting arbitrarily many inputs, it is more efficient to use APPLY instead of REDUCE. The latter is good for associative procedures that have been written to accept exactly two inputs.

Example:

to max :a :b
output ifelse :a > :b [:a] [:b]
end

print reduce "max [2 3 8 7 9 0]"
9

Alternatively, REDUCE can be used to write MAX as a procedure that accepts any number of inputs, as SUM does:

to max [:inputs] 2
if emptyp :inputs [[throw *error [not enough inputs to max]]]
output reduce [ifelse ?1 > ?2 [?1] [?2]] :inputs
end
print (max 2 3 8 7 9 0)
9

* REDUCE
* REDUCE
* REDUCE
* LOGO:0
# $ K * CROSSMAP

crossed CROSSMAP template data (library procedure)
crossed (CROSSMAP template data1 data2 ...)

Outputs a list containing the results of template evaluations. Each data list contributes to a slot in the template; the number of slots is equal to the number of data list inputs. As a special case, if only one data list input is given, that list is taken as a list of data lists, and each of its members contributes values to a slot. CROSSMAP differs from MAP in that instead of taking members from the data inputs in parallel, it takes all possible combinations of members of data inputs, which need not be the same length.

crossed:(LIST) A list of cross mapped input data.

data:(LIST) List of data elements to crossmap.
data1:(LIST) First list of data elements to crossmap.
data2:(LIST) Second list of data elements to crossmap.
template:(WORD) Template to perform cross mapping.

Example:

show (crosmap [word 2l 32] [a b c] [1 2 3 4])
[a1 a2 a3 a4 b1 b2 b3 b4 c1 c2 c3 c4]

For compatibility with the version in CSLS, CROSSMAP templates may use the notation :1 instead of ?1 to indicate slots.


# CASCADE

cascaded CASCADE endtest template startvalue  (library procedure)
cascaded (CASCADE endtest template1 startvalue1 template2 startvalue2 ...)
cascaded (CASCADE endtest template1 startvalue1 template2 startvalue2 ... finaltemplate)

Outputs the result of applying a template (or several templates, see TEMPLATE-BASED ITERATION) repeatedly, with a given value filling the slot the first time, and the result of each application filling the slot for the following application.

In the simplest case, CASCADE has three inputs. The template is a one-slot expression template. That template is evaluated some number of times (perhaps zero). On the first evaluation, the slot is filled with the startvalue input; on subsequent evaluations, the slot is filled with the result of the previous evaluation. The number of evaluations is determined by the endtest input. This can be either a nonnegative integer, in which case the template is evaluated that many times, or a predicate expression template, in which case it is evaluated (with the same slot filler that will be used for the evaluation of the second input) repeatedly, and the CASCADE evaluation continues as long as the predicate value is FALSE. (In other words, the predicate template indicates the condition for stopping.)

cascaded:(THING) Cascaded output of input data.
endtest:(LIST) End Test template.
template:(WORD) Template to perform cascade.
startvalue:(THING) Starting value for initial iteration.
template1:(WORD) First parallel template to perform cascade.
startvalue1:(THING) First parallel starting value for initial iteration.
template2:(WORD) Second parallel template to perform cascade.
startvalue2:(THING) Second parallel starting value for initial iteration.
finaltemplate:(WORD) Final template to perform cascade.

If the template is evaluated zero times, the output from CASCADE is the third (startvalue) input. Otherwise, the output is the value produced by the last template evaluation.

CASCADE templates may include the symbol # to represent the number of times the template has been evaluated. This slot is filled with 1 for the first evaluation, 2 for the second, and so on.

Example:

show cascade 5 [1put # ?] []
[1 2 3 4 5]
show cascade [vowelp first?] [bf?] "spring"
show cascade 5 [# * ?] 1
120

Several cascaded results can be computed in parallel by providing additional template-startvalue pairs as inputs to CASCADE. In this case, all templates (including the endtest template, if used) are multi-slot, with the number of slots equal to the number of pairs of inputs. In each round of evaluations, ?2 represents the result of evaluating the second template in the previous round. If the total number of inputs (including the first endtest input) is odd, then the output from CASCADE is the final value of the first template. If the total number of inputs is even, then the last input is a template that is evaluated once, after the end test is satisfied, to determine the output from CASCADE.

Example:

to fibonacci :n

# CASCADE
$ CASCADE
% CASCADE
* LOGO:0
output (cascade :n (?1 + ?2) 1 [&?1] 0)
end

to piglatin :word
output (cascade [vowelp first?] ~
[word bf ? first?] ~
:word ~
[word ? "ay])
end
cascaded CASCADE.2 endtest template1 startvalue1 template2 startvalue2 (library procedure)

Outputs the result of invoking CASCADE with the same inputs. The only difference is that the default number of inputs is five instead of three.

cascaded:(THING) Cascaded output of input data.

endtest:(LIST) End Test template.
template1:(WORD) First parallel template to perform cascade.
startvalue1:(THING) First parallel starting value for initial iteration.
template2:(WORD) Second parallel template to perform cascade.
startvalue2:(THING) Second parallel starting value for initial iteration.

Example:

to fibonacci :n
output cascade.2 :n [?1 + ?2] 1 [?1] 0
end
# § K • TRANSFER

**outbasket** TRANSFER endtest template inbasket

(library procedure)

Outputs the result of repeated evaluation of the template input. The template is evaluated once for each member of the list inbasket. TRANSFER maintains an outbasket that is initially the empty list. After each evaluation of the template, the resulting value becomes the new outbasket.

**outbasket**: (THING) Transferred output of input data.

**endtest**: (LIST) End test template.

**template**: (WORD) Template to perform transfer.

**inbasket**: (THING) Starting value for initial iteration.

In the template, the symbol ?IN represents the current element from the inbasket; the symbol ?OUT represents the entire current outbasket. Other slot symbols should not be used.

If the first (endtest) input is an empty list, evaluation continues until all inbasket members have been used. If not, the first input must be a predicate expression template, and evaluation continues until either that template’s value is TRUE or the inbasket is used up.

Example (for each word in the last input, if that word is already part of the result, forget it; if not, append that word to the result so far. The result is initially empty.):

```
```
iseq

sequencelist iseq from to (library procedure)

Author: Erich Neuwirth <neuwirth@smc.univie.ac.at>

Outputs a consecutive ordered list of integers starting with from and ending with to. See also RSEQ.

sequencelist:(LIST) List of integers ranging from from to to.

from:(INTEGER) Start range of the sequence to output.
to:(INTEGER) End range of the sequence to output.

show iseq 3 7
[3 4 5 6 7]
show iseq 7 3
[7 6 5 4 3]
RSEQ

sequencelist RSEQ from to count  (library procedure)

Author: Erich Neuwirth <neuwirth@smc.univie.ac.at>

Outputs an ordered list of count equally spaced rational numbers starting with from and ending with to. See also RSEQ.

sequencelist: (LIST) List of numbers ranging from from to to.

from: (NUMBER) Start range of the sequence to output.
to: (NUMBER) End range of the sequence to output.
count: (INTEGER) Number of elements to generate to output list.

show rseq 3 5 9
[3 3.25 3.5 3.75 4 4.25 4.5 4.75 5]
show rseq 3 5 5
[3 3.5 4 4.5 5]
MACRO COMMANDS

MACRO
.DEMACRO
.MACRO

.MACRO proname :input1 :input2 ... (special form)

A macro is a special kind of procedure whose output is evaluated as Logo instructions in the context of the macro's caller. .MACRO is like TO except that the new procedure becomes a macro.

proname:(WORD) Name of the macro procedure being defined (but not preceded by a []).
input1:(WORD) First argument to macro procedure being defined (preceded by a [{}]).
input2:(WORD) Second argument to macro procedure being defined (preceded by a [{}]).

Macros are useful for inventing new control structures comparable to REPEAT, IF, and so on. Such control structures can almost, but not quite, be duplicated by ordinary Logo procedures. For example, here is an ordinary procedure version of REPEAT:

to my.repeat :num :instructions
if :num=0 [stop]
run :instructions
my.repeat :num-1 :instructions
end

This version works fine for most purposes, e.g.,

my.repeat 5 [print "hello"

But it doesn't work if the instructions to be carried out include OUTPUT, STOP, or LOCAL. For example, consider this procedure:

to example
print [Guess my secret word. You get three guesses.]
repeat 3 [type "|?? | ~]
if readword = "secret" [pr "Right! stop!"
print [Sorry, the word was "secret"!]
end

This procedure works as written, but if MY.REPEAT is used instead of REPEAT, it won't work because the STOP will stop MY.REPEAT instead of stopping EXAMPLE as desired.

The solution is to make MY.REPEAT a macro. Instead of carrying out the computation, a macro must return a list containing Logo instructions. The contents of that list are evaluated as if they appeared in place of the call to the macro. Here's a macro version of REPEAT:

.macro my.repeat :num :instructions
if :num=0 [output []]
output sentence :instructions ~
(list *my.repeat :num-1 :instructions)
end

Every macro is an operation -- it must always output something. Even in the base case, MY.REPEAT outputs an empty instruction list. To show how MY.REPEAT works, let's take the example

my.repeat 5 [print "hello"]
For this example, MY.REPEAT will output the instruction list

[print "hello my.repeat 4 [print "hello"]

Logo then executes these instructions in place of the original invocation of MY.REPEAT; this prints "hello" once and invokes another repetition.

The technique just shown, although fairly easy to understand, has the defect of slowness because each repetition has to construct an instruction list for evaluation. Another approach is to make my.repeat a macro that works just like the non-macro version unless the instructions to be repeated includes OUTPUT or STOP.

.macro my.repeat :num :instructions
catch "repeat.catchtag -
[op repeat.done runresult [repeat1 :num :instructions]]
op []
end

to repeat1 :num :instructions
if :num=0 [throw "repeat.catchtag]
run :instructions
.maybeoutput repeat1 :num-1 :instructions
end
to repeat.done :repeat.result
if emptyp :repeat.result [op [stop]]
op list "output quoted first :repeat.result
end

If the instructions do not include STOP or OUTPUT, then REPEAT1 will reach its base case and invoke THROW. As a result, my.repeat's last instruction line will output an empty list, so the second evaluation of the macro result will do nothing. But if a STOP or OUTPUT happens, then REPEAT.DONE will output a STOP or OUTPUT instruction that will be re-executed in the caller's context.

The macro-defining commands have names starting with a dot because macros are an advanced feature of Logo; it's easy to get in trouble by defining a macro that doesn't terminate, or by failing to construct the instruction list properly.

Lisp users should note that Logo macros are NOT special forms. That is, the inputs to the macro are evaluated normally, as they would be for any other Logo procedure. It's only the output from the macro that's handled unusually.

Here's another example:

.macro localmake :name :value
output (list ("local-
word ":name-\n"apply-
"make-
(list :name :value))
end

It's used this way:

to try
localmake "garply "hello
print :garply
end

LOCALMAKE outputs the list

[local "garply apply "make [garply hello]]
The reason for the use of `APPLY` is to avoid having to decide whether the second input to `MAKE` requires a quotation mark before it. (In this case it would -- `MAKE "GARPLY HELLO` -- but the quotation mark would be wrong if the value were a list.)

It's often convenient to use the `.` function to construct the instruction list:

```
.macro localmake :name :value
  op ` local , word "":name apply "make [[:name],[value]]"
.end
```

On the other hand, `.` is slow, since its tree recursive and written in Logo.
.DEFMACRO

.DEFMACRO procname text

.DEFMACRO is like DEFINE with the same exceptions as .MACRO.

procname:(WORD) Name of the macro procedure being defined.
text:(LIST) Definition of macro procedure as a list.

Example:

See DEFINE and .MACRO.
**ERROR PROCESSING**

If an error occurs, Logo takes the following steps. First, if there is an available variable named ERRACT, Logo takes its value as an instructionalist and runs the instructions. The operation ERROR may be used within the instructions (once) to examine the error condition. If the instructionalist invokes PAUSE, the error message is printed before the pause happens. Certain errors are "recoverable"; for one of those errors, if the instructionalist outputs a value, that value is used in place of the expression that caused the error. (If ERRACT invokes PAUSE and the user then invokes CONTINUE with an input, that input becomes the output from PAUSE and therefore the output from the ERRACT instructionalist.)

It is possible for an ERRACT instructionalist to produce an inappropriate value or no value where one is needed. As a result, the same error condition could recur forever because of this mechanism. To avoid that danger, if the same error condition occurs twice in a row from an ERRACT instructionalist without user interaction, the message "Erract loop" is printed and control returns to toplevel. "Without user interaction" means that if ERRACT invokes PAUSE and the user provides an incorrect value, this loop prevention mechanism does not take effect and the user gets to try again.

During the running of the ERRACT instructionalist, ERRACT is locally unbound, so an error in the ERRACT instructions themselves will not cause a loop. In particular, an error during a pause will not cause a pause-within-a-pause unless the user reassigns the value [PAUSE] to ERRACT during the pause. But such an error will not return to toplevel; it will remain within the original pause loop.

If there is no available ERRACT value, Logo handles the error by generating an internal THROW "ERROR. (A user program can also generate an error condition deliberately by invoking THROW.) If this throw is not caught by a CATCH "ERROR in the user program, it is eventually caught either by the toplevel instruction loop or by a pause loop, which prints the error message. An invocation of CATCH "ERROR in a user program locally unbinds ERRACT, so the effect is that whichever of ERRACT and CATCH "ERROR is more local will take precedence.

If a floating point overflow occurs during an arithmetic operation, or a two-input mathematical function (like POWER) is invoked with an illegal combination of inputs, the "doesn't like" message refers to the second operand, but should be taken as meaning the combination.

**ERROR CODES**

---

* ERROR_PROCESSING
* ERROR_PROCESSING
* ERROR_PROCESSING
* LOGO:0
ERROR CODES

Here are the numeric codes that appear as the first element of the list output by ERROR when an error is caught, with the corresponding messages. Some messages may have two different codes depending on whether the error is recoverable (that is, a substitute value can be provided through the ERRACT mechanism) in the specific context. Some messages are warnings rather than errors; these will not be caught. The first two are so bad that Logo exits immediately.

0  Fatal internal error (can’t be caught)
1  Out of memory (can’t be caught)
2  PROC doesn’t like DATUM as input (not recoverable)
3  PROC didn’t output to PROC
4  Not enough inputs to PROC
5  PROC doesn’t like DATUM as input (recoverable)
6  Too much inside ( ‘ )’s
7  I don’t know what to do with DATUM
8  ’ )’ not found
9  VAR has no value
10  Unexpected ’)
11  I don’t know how to PROC (recoverable)
12  Can’t find catch tag for THROWTAG
13  PROC is already defined
14  Stopped
15  Already dribbling
16  File system error
17  Assuming you mean IFELSE, not IF (warning only)
18  VAR shadowed by local in procedure call (warning only)
19  Throw “Error
20  PROC is a primitive
21  Can’t use TO inside a procedure
22  I don’t know how to PROC (not recoverable)
23  IFTRUE/IFFALSE without TEST
24  Unexpected ’)
25  Unexpected ’)
26  Couldn’t initialize graphics
27  Macro returned VALUE instead of a list
28  I don’t know what to do with VALUE
29  Can only use STOP or OUTPUT inside a procedure

* ERROR_CODES
* ERROR_CODES
* ERROR_CODES
* LOGO:0
SPECIAL VARIABLES

Logo takes special action if any of the following variable names exists. They follow the normal scoping rules, so a procedure can locally set one of them to limit the scope of its effect. Initially, no variables exist except CASEIGNORED, which is TRUE and buried.

SPECIAL COMMANDS

# $ K • SPECIAL_VARIABLES

$ SPECIAL_VARIABLES

K SPECIAL_VARIABLES

* LOGO:0
SPECIAL COMMANDS

CASEIGNOREDP
ERRACT
PRINTDEPTHLIMIT
PRINTWIDTHLIMIT
REDEEP
STARTUP
MACHINE
# $K CASEIGNOREDP

CASEIGNOREDP

If TRUE, indicates that lower case and upper case letters should be considered equal by EQUALP, BEFOREP, MEMBERP, etc. Logo initially makes this variable TRUE, and buries it.

Example:

make "caseignoredp "false
show equalp "a "A
false
make "caseignoredp "true
show equalp "a "A
true
### ERRACT

**ERRACT**

An instruction list that will be run if there is an error. Typically has the value `[PAUSE]` to allow interactive debugging.

Example:

```plaintext
fence
fd 1000
Turtle out of bounds
make "erract [print [You really blew it]]
fd 1000
You really blew it
```
PRINTDEPTHLIMIT

If a nonnegative integer, indicates the maximum depth of sublist structure that will be printed by PRINT, etc.

Example:

print [[1 [2 [3 [4 [5]]]]]]
[1 [2 [3 [4 [5]]]]]
make "printdepthlimit 4
print [[1 [2 [3 [4 [5]]]]]]
[1 [2 [3 [... .]]]]
PRINTWIDTHLIMIT

If a nonnegative integer, indicates the maximum number of elements in any one list that will be printed by PRINT, etc.

print [1 2 3 4 5 6]
  1 2 3 4 5 6
make "printwidthlimit 4
print [1 2 3 4 5 6]
  1 2 3 4 ...

^PRINTWIDTHLIMIT
$PRINTWIDTHLIMIT
^KPRINTWIDTHLIMIT
^LOGO:0
REDEFP

If TRUE, allows primitives to be erased (ERASE) or redefined (COPYDEF).

Example:

erase "fd
fd is a primitive
make "redefp "true
erase "fd
fd
I don't know how to fd
STARTUP

If assigned a list value in a file loaded by LOAD, that value is run as an instructionlist after the loading.

Example:

to myprog
print "Hello"
end
make "startup [myprog]
save "myprog.lg"
myprog.lg
erall
load "myprog.lg"
Hello
list MACHINE

This command outputs the characteristics of the machine in a list. Where the list has the following format [Windows WindowsVersion BitMapWidth BitMapHeight Palette]. Where palette is a 1 if true and 0 if false. This can be used to write code that is compatible between multiple platforms.

list (LIST) List of parameters describing the current machine.

Windows 3.0 = 30
Windows 3.1 = 31
NT/Windows 95 = 40

Example:

show machine
[1 31 1000 1000 0]
HELP COMMANDS

HELP
WINHELP
HELP

HELP
(HELP keyword)

This command has two forms (with and without a keyword argument). Without a keyword Logo will enter Windows help on LOGO at the top level. With a keyword argument Logo will search the Windows help for the keyword. You may enter the full keyword or a partial keyword.

keyword (WORD) A keyword that you would normally enter in the help search engine.

Example:

help "introduction ;(Enter help on introduction)
help "intro ;(Enter help search engine)

For context sensitive help, see the EDIT command.

Note also that Windows Help allows you to Copy text from Help to the Clipboard. Since the Editor within Logo also supports the Clipboard this means that you can copy examples within Help into the Editor, save them, and then execute them. To do this:

Click on EDIT in the Help menu.
Select Copy.
Using the mouse select the desired text (example code).
Click on COPY button (it is now in the Clipboard).
Enter the Logo Editor.
Set the cursor where you want the example inserted.
Click on EDIT in the Edit menu.
Select paste (it's now in the editor).
WINHELP

WINHELP filename
(WINHELP filename keyword)

This command has two forms (with and without a keyword argument). Without a keyword Logo will enter Windows help on the desired FILENAME at the top level. With a keyword argument Logo will search the Windows help FILENAME for the keyword. You must enter the full keyword.

filename: (WORD) A path to a help file you wish to open.
keyword: (WORD) A keyword that you would normally enter in the help search engine.

Example:

(winhelp "c:\myhelp *introduction") ;(Enter help on introduction)
DIRECTORY COMMANDS

DIR
FILES
DIRECTORIES
CHDIR
POPDIR
MKDIR
RMDIR
DIR

This command prints out files and directories in the current directory. It prints out the FILES list and DIRECTORIES list.

Example:

dir
File LOGO.EXE
File LOGO.HLP
File MYPROG.LG
Directory .
Directory ..
Directory LOGOLIB
Directory EXAMPLES
#K FILES

filelist FILES

Outputs a filelist, a list of files in the current directory.

filelist: (LIST) List of files.

Example:

show files
[logo.doc logo.exe LOGICEX.BMP LICENSE.TXT MSWLOGO.TXT mcistrwh.hlp LOGO.CNT]
# $K$ DIRECTORIES

directorylist DIRECTORIES

Outputs a directorylist, a list of directories in the current directory.

directorylist:(LIST) List of directories.

Example:

show directories
[EXAMPLES LOGLIB . . ]
CHDIR directory

This command is exactly like the DOS command CHDIR (cd). The parameter must be a word (the name of the directory you want work in). You can use the DIR command to list both procedures and directories. See also POPDIR and MKDIR.

directory:(WORD) A path to a directory you wish to change to.

Example:

```plaintext
chdir examples
Pushed to C:\LOGO\EXAMPLES
popdir
Popped to C:\LOGO
```
**POPDIR**

This command pops you up (1 level) out of a directory. It is equivalent to CHDIR "." command. See also CHDIR and MKDIR.

Example:

```plaintext
chdir "examples
Pushed to C:\LOGO\EXAMPLES
popdir
Popped to C:\LOGO
```
MKDIR

MKDIR directory

This command makes (creates) a directory and then changes (CHDIR) you into it. The parameter must be a word (the name of the directory you want to make). You can use the DIR command to list the files and directories in the current directory. Once done with the directory you can pop (POPDIR) back out and change to another.

directory:(WORD) A path to a directory you wish to make.

Example:

mkdir "junk"
Now in newly created directory junk
popdir
Popped to C:\LOGO

^ MKDIR
$ MKDIR
K MKDIR
* LOGO:0
RMDIR directory

This command removes (deletes) a directory. The parameter must be a word (a name of an existing directory). You cannot remove a directory if you are (CHDIR) into it. Nor can you remove it, if it contains any files or other directories (DIR must not return anything while changed to the directory you want to remove). So to be sure you can remove it, do a DIR and check if it’s there, then change (CHDIR) to it, do DIR again and confirm it’s empty, pop (POPDIR) back out and then remove (RMDIR) it. See also MkDIR.

directory: (WORD) A path to a directory you wish to remove.

Example:

mkdir "junk"
Now in newly created directory junk
popdir
Popped to C:\LOGO
remdir "junk"
Logo directory junk removed
# § K • WINDOWS FUNCTIONS

This section describes how the LOGO programmer can create powerful Graphical User Interfaces (GUIs). Most any GUI has an Application Programming Interface (API), which is what this section documents for Logo regarding Windows. Both the GUI and the API maintain a parent-child relationship. That is, Windows and controls appear (graphically) in a nested fashion (or owned by one another). The nested appearance is maintained in the code you develop to present it.

For each "somethingCREATE" command a parent must be specified which identifies the relationship. When a parent of any kind is "somethingDELETED" so will all its "child" Windows and controls. Every Window function (command) specifies a "name". On the "somethingCREATE" commands it is there to label the "something" so that it can later be referenced. On most other commands it is used as a "handle" to identify the Window or control you wish to communicate with. And third it is also used to reference the "parent" of the Window or control that is to "own" this new "something".

When controls are in windows the coordinates used in all the Windows functions are in Windows coordinate system (NOT the turtle coordinate system). That is the Y axis is upside down (Positive numbers are down). The origin is NOT the middle of the screen it is the upper left hand corner of the window.

When controls are in the graphics window the coordinates will be in the turtle coordinate system.

WINDOW COMMANDS
DIALOG COMMANDS
LISTBOX COMMANDS
COMBOBOX COMMANDS
SCROLLBAR COMMANDS
BUTTON COMMANDS
STATIC COMMANDS
GROUPBOX COMMANDS
CHECKBOX COMMANDS
RADIODOWN COMMANDS
DEBUG COMMANDS
Modal vs. Modeless Windows
Predefined Windows
WINDOWS Example
WINDOW COMMANDS

WINDOWCREATE
WINDOWDELETE
WINDOWCREATE

WINDOWCREATE parent name title xpos ypos width height setup

This command will create a WINDOW. A WINDOW is used as the frame work to which you add other window objects or controls (sometimes called widgets). You can add things such as buttons, scrollbars, listboxes, etc. to the WINDOW. (see Modal vs. Modeless Windows)

parent:(WORD) Is the name of the WINDOW that is to own this new WINDOW. If this is the first window created use "root" as the parent name.
name:(WORD) Is used to identify this WINDOW (perhaps as the parent of another future window or control) and MUST be unique.
title:(LIST) Is used as the title (caption) of the WINDOW.
xpos:(INTEGER) Is the X position you wish to place the upper left corner of the new WINDOW.
ypos:(INTEGER) Is the Y position you wish to place the upper left corner of the new WINDOW.
width:(INTEGER) Is the width of the new WINDOW.
height:(INTEGER) Is the height of the new WINDOW.
setup:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the window is created. The commands to be executed should most likely be other somethingCREATE functions to add controls to the window. You may give an empty list for this argument and add controls later.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
windowdelete "mywindow
# $K $ WINDOWDELETE

WINDOWDELETE name

This command will delete (close) the WINDOW with the given name. All the child windows and controls will be also deleted. If name is not found all windows and all controls in the graphics window.

name (WORD) is used to identify the WINDOW you want destroyed.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
windowdelete "mywindow
DIALOG COMMANDS

DIALOGCREATE
DIALOGDELETE

* DIALOG_COMMANDS
* DIALOG_COMMANDS
* DIALOG_COMMANDS
* LOGO:0
**DIALOGCREATE**

**DIALOGCREATE parent name title xpos ypos width height setup**

This command will create a DIALOG window. A DIALOG window is used as the frame work to which you add other window objects or controls (sometimes called widget's). You can add things such as buttons, scrollbars, listboxes, etc. to the DIALOG window. This function is similar to WINDOWCREATE except it will not return to the caller until the Window is closed (see Modal vs. Modeless Windows).

- **parent**: (WORD) Is the name of the DIALOG window that is to own this new DIALOG window. If this is the first window created use 'root' as the parent name.
- **name**: (WORD) is used to identify this DIALOG window (perhaps as the parent of another future window or control) and MUST be unique.
- **title**: (LIST) is used as the title (caption) of the DIALOG window.
- **xpos**: (INTEGER) is the X position you wish to place the upper left corner of the new DIALOG window.
- **ypos**: (INTEGER) is the Y position you wish to place the upper left corner of the new DIALOG window.
- **width**: (INTEGER) is the width of the new DIALOG window.
- **height**: (INTEGER) is the height of the new DIALOG window.
- **setup**: (LIST) is a (short) list of logo commands (or a procedure name) to execute when the DIALOG window is created. The commands to be executed should most likely be other somethingCREATE functions to add controls to the window. The reason is, is that DIALOGCREATE will not return until the window is closed (the caller looses control and cannot add controls). So be sure to add an (OK, END, CLOSE, CANCEL or whatever) button that will call DIALOGDELETE on this window.

Example:

```
to mysetup
    buttoncreate "mydialog "myok "OK 25 25 50 50 [dialogdelete "mydialog]
    end
    dialogcreate "main "mydialog "mytitle 0 0 100 100 [mysetup]
```

---

* DIALOGCREATE
* DIALOGCREATE
* DIALOGCREATE
* LOGO:0
DIALGDELETE name

This command will delete (close) the DIALOG window with the given name. Note all the child windows and controls will be also deleted.

name (WORD) is used to identify the DIALOG window you want destroyed.

Example:

to mysetup
buttoncreate "mydialog "myok "OK 25 25 50 50 [dialogdelete "mydialog]
end
dialogcreate "main "mydialog "mytitle 0 0 100 100 [mysetup]
LISTBOX COMMANDS

LISTBOXCREATE
LISTBOXDELETE
LISTBOXGETSELECT
LISTBOXADDSTRING
LISTBOXDELETESTRING

* LISTBOX_COMMANDS
* LISTBOX_COMMANDS
* LISTBOX_COMMANDS
* LISTBOX_COMMANDS
* LOGO:0
LISTBOXCREATE parent name xpos ypos width height

This command will create a LISTBOX control. A LISTBOX control is used to give the user a selection of items.

parent:(WORD) Is the name of the DIALOG window that is to own this new LISTBOX control.
name:(WORD) Is used to identify this LISTBOX control.
xpos:(INTEGER) Is the X position you wish to place the upper left corner of the new LISTBOX control.
ypos:(INTEGER) Is the Y position you wish to place the upper left corner of the new LISTBOX control.
width:(INTEGER) Is the width of the new LISTBOX control.
height:(INTEGER) Is the height of the new LISTBOX control.

Example (in a new window):

windowcreate "main "mywindow "mytitle 0 0 100 100 []
listboxcreate "mywindow "mylist 25 25 50 50
windowdelete "mywindow

Example (in the graphics window):

listboxcreate "main "mylist 25 25 100 100
windowdelete "main

# § K • LISTBOXCREATE

LISTBOXCREATE parent name xpos ypos width height

This command will create a LISTBOX control. A LISTBOX control is used to give the user a selection of items.

parent:(WORD) Is the name of the DIALOG window that is to own this new LISTBOX control.
name:(WORD) Is used to identify this LISTBOX control.
xpos:(INTEGER) Is the X position you wish to place the upper left corner of the new LISTBOX control.
ypos:(INTEGER) Is the Y position you wish to place the upper left corner of the new LISTBOX control.
width:(INTEGER) Is the width of the new LISTBOX control.
height:(INTEGER) Is the height of the new LISTBOX control.

Example (in a new window):

windowcreate "main "mywindow "mytitle 0 0 100 100 []
listboxcreate "mywindow "mylist 25 25 50 50
windowdelete "mywindow

Example (in the graphics window):

listboxcreate "main "mylist 25 25 100 100
windowdelete "main

* LISTBOXCREATE
§ LISTBOXCREATE
K LISTBOXCREATE
* LOGO:0
LISTBOXDELETE

LISTBOXDELETE name

This command will delete (close) the LISTBOX control with the given name.

name (WORD) is used to identify the LISTBOX control you want destroyed.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
listboxcreate "mywindow "mylist 25 25 50 50
listboxdelete "mylist
windowdelete "mywindow
LISTBOXGETSELECT

text LISTBOXGETSELECT name

This command will solicit (ask) the LISTBOX control for the selected item and output a copy of it.

text:(LIST) Represents the selected item of the LISTBOX control.

name:(WORD) is used to identify the LISTBOX you wish to solicit.

Example:

to dodraw
    cs
    if equalp [TRIANGLE] listboxgetselect "mylist [repeat 3 [fd 100 rt 120]]"
    if equalp [SQUARE] listboxgetselect "mylist [repeat 4 [fd 100 rt 90]]"
    end
    windowcreate "main "mywindow "mytitle 0 0 100 100 []
    listboxcreate "mywindow "mylist 25 0 50 50
    listboxaddstring "mylist [TRIANGLE]
    listboxaddstring "mylist [SQUARE]
    buttoncreate "mywindow "mydraw "Draw 25 50 50 25 [dodraw]
    <select figure from listbox and then click on Draw button>
    windowdelete "mywindow"
# $K • LISTBOXADDSTRING

LISTBOXADDSTRING name item

This command will add the item to the LISTBOX control with the given name.

name (WORD) is used to identify the LISTBOX control you wish to add a string to. item (WORD) is the item you wish to insert into the LISTBOX control.

Example:

to dodraw
    cs
    if equalp [TRIANGLE] listboxgetselect "mylist [repeat 3 [fd 100 rt 120]]"
    if equalp [SQUARE] listboxgetselect "mylist [repeat 4 [fd 100 rt 90]]"
    end

    windowcreate "main "mywindow "mytitle 0 0 100 100 []
    listboxcreate "mywindow "mylist 25 0 50 50
    listboxaddstring "mylist [TRIANGLE]
    listboxaddstring "mylist [SQUARE]
    buttoncreate "mywindow "mydraw "Draw 25 50 50 25 [dodraw]
    <select figure from listbox and then click on Draw button>
    windowdelete "mywindow
LISTBOXDELETESTRING

LISTBOXDELETESTRING name index

This command will delete an item at "index" of the LISTBOX control.

name (WORD) is used to identify the LISTBOX control you want to delete a string from.
index: (INTEGER) Index of item you wish deleted (starting at 0).

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
listboxcreate "mywindow "mylist 25 0 50 50
listboxaddstring "mylist [TRIANGLE]
listboxaddstring "mylist [SQUARE]
listboxaddstring "mylist [HEXAGON]
listboxdeletestring "mylist 1
windowdelete "mywindow
COMBOBOX COMMANDS

COMBOBOXCREATE
COMBOBOXDELETE
COMBOBOXGETTEXT
COMBOBOXGETTEXT
COMBOBOXADDSTRING
COMBOBOXDELETESTRING
COMBOBOXCREATE  parent name xpos ypos width height

This command will create a COMBOBOX control. A COMBOBOX control is used to give the user a selection of items and allow the user to enter a selection not listed. A COMBOBOX is two controls in one (A LISTBOX control and an EDIT control). If you wish to create an EDIT control (a COMBOBOX without a LISTBOX) just set the height to a size in which the LISTBOX doesn't fit.

parent:(WORD) Is the name of the DIALOG window that is to own this new COMBOBOX control.
name:(WORD) Is used to identify this COMBOBOX control and MUST be unique.
xpos:(INTEGER) Is the X position you wish to place the upper left corner of the new COMBOBOX control.
ypos:(INTEGER) Is the Y position you wish to place the upper left corner of the new COMBOBOX control.
width:(INTEGER) Is the width of the new COMBOBOX control.
height:(INTEGER) Is the height of the new COMBOBOX control.

Example (in a new window):

windowcreate "main "mywindow "mytitle 0 0 100 100 []
comboxcreate "mywindow "mycombo 25 25 50 50
windowdelete "mywindow

Example (in the graphics window):

comboxcreate "main "mycombo 25 25 100 100
windowdelete "main

# $ K • COMBOBOXCREATE

$ COMBOBOXCREATE
K COMBOBOXCREATE
* LOGO:0
COMBOBOXDELETE

COMBOBOXDELETE name

This command will delete (close) the COMBOBOX control with the given name.

name (WORD) is used to identify the COMBOBOX you want destroyed.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
comboxboxcreate "mywindow "mycombo 25 25 50 50
comboxboxdelete "mycombo
windowdelete "mywindow
text COMBOBOXGETTEXT name

This command will solicit (ask) the COMBOBOX, for the contents of the EDIT control portion of the COMBOBOX (which may or may not be a selected item).

text(LIST) Represents the EDIT control contents in the COMBOBOX control.

name(WORD) is used to identify the COMBOBOX control you wish to solicit.

Example:

to dodraw
  cs
  make "sides first comboboxgettext "mycombo
  repeat :sides [fd 50 rt 360.0/:sides]
end
windowcreate "main "mywindow "mytitle 0 0 100 100 []
comboboxcreate "mywindow "mycombo 25 0 50 50
comboboxaddstring "mycombo [3]
comboboxaddstring "mycombo [4]
comboboxaddstring "mycombo [5]
comboboxaddstring "mycombo [6]
buttoncreate "mywindow "mydraw "Draw 25 50 50 25 [dodraw]
<select or enter number of sides from combobox and then click on Draw button>
windowdelete "mywindow
COMBOBOXSETTEXT

This command will set the contents of the (EDIT control component of) COMBOBOX with "text".

name:(WORD) is used to identify the COMBOBOX control you wish to SETTEXT to.
text:(WORD) is the item you wish to insert into the (EDIT control component of) COMBOBOX control.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
comboxcreate "mywindow "mycombo 25 0 50 50
comboxaddstring "mycombo [3]
comboxaddstring "mycombo [4]
comboxaddstring "mycombo [5]
comboxaddstring "mycombo [6]
comboxsettext "mycombo [3]
windowdelete "mywindow
COMBOBOXADDSTRING

COMBOBOXADDSTRING name item

This command will add the "item" to the COMBOBOX with the given "name". Note that items in the LISTBOX are automatically sorted as they are inserted.

name:(WORD) is used to identify the COMBOBOX you wish to add to.

item:(LIST) is the item you wish to insert into the (LISTBOX component of) COMBOBOX.

Example:

to dodraw
  cs
  make "sides first comboboxgettext "mycombo
  repeat :sides [fd 50 rt 360.0/:sides] end
  windowcreate "main "mywindow "mytitle 0 0 100 100 []
  comboboxcreate "mywindow "mycombo 25 0 50 50
  comboboxaddstring "mycombo [3]
  comboboxaddstring "mycombo [4]
  comboboxaddstring "mycombo [5]
  comboboxaddstring "mycombo [6]
  buttoncreate "mywindow "mydraw "Draw 25 50 50 25 [dodraw]
  <select or enter number of sides from combobox and then click on Draw button>
  windowdelete "mywindow
COMBOBOXDELETESTRING

COMBOBOXDELETESTRING name index

This command will delete an item at "index" of the COMBOBOX with the given "name".

name: (WORD) is used to identify the COMBOBOX you want to delete string from.

index: (INTEGER) Index of item you wish deleted (starting at 0).

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
comboboxcreate "mywindow "mycombo 25 0 50 50
comboboxaddstring "mycombo [3]
comboboxaddstring "mycombo [4]
comboboxaddstring "mycombo [5]
comboboxaddstring "mycombo [6]
comboboxdeletestring "mycombo 1
windowdelete "mywindow
SCROLLBAR COMMANDS

SCROLLBARCREATE
SCROLLBARDELETE
SCROLLBARSET
SCROLLBARGET
SCROLLBARCREATE parent name xpos ypos width height callback

This command will create a SCROLLBAR control. A SCROLLBAR control is used to solicit, from the user, a variable value (although you can map its function to anything you desire). It is also common to link it to a STATIC control to inform the user of its setting (but this is not required). You must also not forget to set the SCROLLBAR range and initial value using SCROLLBARSET.

The orientation (vertical or horizontal) of the SCROLLBAR is determined by the longest dimension. That is, if X > Y then horizontal is chosen otherwise vertical is chosen.

parent:(WORD) is the name of the window that is to own this new SCROLLBAR control.
name:(WORD) is used to identify this SCROLLBAR control and MUST be unique.
xpos:(INTEGER) is the X position you wish to place the upper left corner of the new SCROLLBAR control.
ypos:(INTEGER) is the Y position you wish to place the upper left corner of the new SCROLLBAR control.
width:(INTEGER) is the width of the new SCROLLBAR control.
height:(INTEGER) is the height of the new SCROLLBAR control.
callback:(LIST) is a (short) list of logo commands (or a procedure name) to execute when the user adjusts the SCROLLBAR. It is common to call a procedure that informs the user of what the SCROLLBAR state is.

Example (in a new window):

windowcreate "main "mywindow "mytitle 0 0 100 100 []
scrollbarcreate "mywindow "myscroll 25 25 50 25 [setheading scrollbarget "myscroll]
scrollbaset "myscroll 0 360 0
<try moving scrollbar and notice what happens to the turtle>
windowdelete "mywindow

Example (in the graphics window):

scrollbarcreate "main "myscroll 25 25 100 25 [setheading scrollbarget "myscroll]
scrollbaset "myscroll 0 360 0
<try moving scrollbar and notice what happens to the turtle>
windowdelete "main

^ SCROLLBARCREATE
$ SCROLLBARCREATE
k SCROLLBARCREATE
* LOGO:0
SCROLLBARDELETE

SCROLLBARDELETE name

This command will delete (close) the SCROLLBAR control with the given name.

name (WORD) is used to identify the SCROLLBAR control you want destroyed.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
scrollbarcreate "mywindow "myscroll 25 25 50 25 []
scrollbardelete "myscroll
windowdelete "mywindow
SCROLLBARSET

SCROLLBARSET name lorange hirange position

This command will set the output range and current position of the SCROLLBAR control. You can issue a SCROLLBARSET as many times as you want.

name:(WORD) is used to identify the SCROLLBAR control you want to set.
lorange:(INTEGER) is used as the low range of the output values of the SCROLLBAR control.
hirange:(INTEGER) is used as the high range of the output values of the SCROLLBAR control.
position:(INTEGER) is used to set the current position (and output value) of the SCROLLBAR control.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
scrollbarcreate "mywindow "myscroll 25 25 50 25 [setheading scrollbarget "myscroll]
scrollbarset "myscroll 0 360 0
<try moving scrollbar and notice what happens to the turtle>
windowdelete "mywindow
# $ K • SCROLLBARGET

**pos** SCROLLBARGET name

This command will output the position of the SCROLLBAR control of the given name.

**pos**:(INTEGER) is the position of the scrollbar (always within the SET range).

**name**:(WORD) is used to identify the SCROLLBAR control you wish to solicit.

Example:

```plaintext
windowcreate "main "mywindow "mytitle 0 0 100 100 []
scrollbarcreate "mywindow "myscroll 25 25 50 25 [setheading scrollbarget "myscroll]
scrollbarget "myscroll 0 360 0
<try moving scrollbar and notice what happens to the turtle>
windowdelete "mywindow
```

* SCROLLBARGET
* SCROLLBARGET
$ SCROLLBARGET
K SCROLLBARGET
* LOGO:0
# $K$ BUTTON COMMANDS

BUTTONCREATE
BUTTONDELETE
BUTTONUPDATE
$\text{BUTTONCREATE}$

**BUTTONCREATE** parent name label xpos ypos width height callback

This command will create a BUTTON control. A BUTTON control is used to allow the user to trigger events. The only function of the BUTTON control is to execute the "callback" list.

- **parent**: (WORD) is the name of the window that is to own this new BUTTON control.
- **name**: (WORD) is used to identify this BUTTON control.
- **label**: (LIST) is used as the label of the BUTTON control.
- **xpos**: (INTEGER) is the X position you wish to place the upper left corner of the new BUTTON control.
- **ypos**: (INTEGER) is the Y position you wish to place the upper left corner of the new BUTTON control.
- **width**: (INTEGER) is the width of the new BUTTON control.
- **height**: (INTEGER) is the height of the new BUTTON control.
- **callback**: (LIST) is a (short) list of logo commands (or a procedure name) to execute when the user clicks on the BUTTON.

**Example:**

```logo
windowcreate "main "mywindow "mytitle 0 0 100 100 []
buttoncreate "mywindow "myleft "Left 25 25 25 25 [fd 2 lt 1]
buttoncreate "mywindow "myright "Right 50 25 25 25 [fd 2 rt 1]
<click left or right repeatedly and watch the turtle>
windowdelete "mywindow
```

**Example (in the graphics window):**

```logo
buttoncreate "main "myleft "Left 50 50 50 25 [fd 2 lt 1]
buttoncreate "main "myright "Right 100 50 50 25 [fd 2 rt 1]
<click left or right repeatedly and watch the turtle>
windowdelete "main
```

* BUTTONCREATE
* BUTTONCREATE
* BUTTONCREATE
* LOGO:0
BUTTONDELETE

BUTTONDELETE name

This command will delete (close) the BUTTON control with the given name.

name (WORD) is used to identify the BUTTON you want destroyed.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
buttoncreate "mywindow "mybutton "PUSH 25 25 25 25 [print "pushed]";
buttondelete "mybutton"
windowdelete "mywindow"
BUTTONUPDATE

BUTTONUPDATE name text

This command will replace the contents of the BUTTON control with "text".

name: (WORD) is used to identify the BUTTON control you want to update. text: (LIST) is used as the new contents of the BUTTON control.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
make "Pushed 0
buttoncreate "mywindow "mybutton :Pushed 25 25 25 25 -
    [make "Pushed :Pushed + 1 Buttonupdate "mybutton :Pushed]
buttondelete "mybutton
windowdelete "mywindow
$K$ $K$ STATIC COMMANDS

STATICCREATE
STATICDELETE
STATICUPDATE

* STATIC_COMMANDS
$ STATIC_COMMANDS
$ STATIC_COMMANDS
$ STATIC_COMMANDS
LOGO:0
STATIC CREATE

STATICCREATE parent name text xpos ypos width height

This command will create a STATIC control. A STATIC control is used to simply display text. The name can be a bit misleading. In that a STATIC control can be very dynamic by using the STATICUPDATE command.

parent:(WORD) Is the name of the DIALOG window that is to own this new STATIC control.
name:(WORD) Is used to identify this STATIC control and MUST be unique.
text:(LIST) Is used as the (perhaps initial) contents of the STATIC control.
xpos:(INTEGER) Is the X position you wish to place the upper left corner of the new STATIC control.
ypos:(INTEGER) Is the Y position you wish to place the upper left corner of the new STATIC control.
width:(INTEGER) Is the width of the new STATIC control.
height:(INTEGER) Is the height of the new STATIC control.

Example (in a new window):

windowcreate "main "mywindow "mytitle 0 0 100 100 []
staticcreate "mywindow "mystatic [Heading=0] 25 25 50 25
repeat 72 [rt 5 staticupdate "mystatic se [Heading=] heading wait 60]
windowdelete "mywindow

Example (in the graphics window):

staticcreate "main "mystatic [Heading=0] 25 25 100 50
repeat 72 [rt 5 staticupdate "mystatic se [Heading=] heading wait 60]
windowdelete "main
STATICDELETE

This command will delete (close) the STATIC control with the given name.

name (WORD) is used to identify the STATIC control you want destroyed.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
staticcreate "mywindow "mystatic [This is Static] 25 25 50 25
staticcreate "mystatic
windowdelete "mywindow
**STATICUPDATE**

STATICUPDATE name text

This command will replace the contents of the STATIC control with "text".

name: (WORD) is used to identify the STATIC control you want to update.  
text: (LIST) is used as the new contents of the STATIC control.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100 []
staticcreate "mywindow "mystatic [Heading=0] 25 25 50 25
repeat 72 [rt 5 staticupdate "mystatic se [Heading=] heading wait 60]
windowdelete "mywindow
```
GROUPBOX COMMANDS

GROUPBOXCREATE
GROUPBOXDELETE
GROUPBOXCREATE parent name xpos ypos width height

This command will create a GROUPBOX control. A GROUPBOX control is a unique control compared with most other Windows functions. It is unique because all it does is group RadioButtons (RADIOBUTTONCREATE) and CheckBoxes (CHECKBOXCREATE) both graphically and logically. RadioButtons and CheckBoxes must belong to a GROUPBOX. Also realize that RadioButtons and CheckBoxes placed in the GROUPBOX still use the parents origin NOT the GROUPBOX origin.

parent:(WORD) Is the name of the DIALOG window that is to own this new GROUPBOX control.
name:(WORD) Is used to identify this GROUPBOX control and MUST be unique.
xpos:(INTEGER) Is the X position you wish to place the upper left corner of the new GROUPBOX control.
ypos:(INTEGER) Is the Y position you wish to place the upper left corner of the new GROUPBOX control.
width:(INTEGER) Is the width of the new GROUPBOX control.
height:(INTEGER) Is the height of the new GROUPBOX control.

Example:

to checkonthings
if else checkbox "myhideturtle [ht] [st]
end

windowcreate "main "mywindow "mytitle 0 0 100 100 []
groupboxcreate "mywindow "mygroupbox 10 10 80 40
checkboxcreate "mywindow "mygroupbox "myhideturtle [Hide Turtle] 20 20 60 20
buttoncreate "mywindow "mybutton "GO 40 50 25 25 [checkonthings]
windowdelete "mywindow
GROUPBOXDELETE

GROUPBOXDELETE name

This command will delete (close) the GROUPBOX control with the given name.

name (WORD) is used to identify the GROUPBOX you want destroyed.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
groupboxcreate "mywindow "mygroupbox 10 10 80 40
groupboxdelete "mygroupbox
windowdelete "mywindow
CHECKBOX COMMANDS

CHECKBOXCREATE
CHECKBOXDELETE
CHECKBOXGET
CHECKBOXSET

* CHECKBOX_COMMANDS
* CHECKBOX_COMMANDS
* CHECKBOX_COMMANDS
* LOGO:0
CHECKBOXCREATE parent group name label xpos ypos width height

This command will create a CHECKBOX control. A CHECKBOX control is used to give the user a selection of a two state (True or False) item. A CHECKBOX must also be associated with a GROUPBOX. (GROUPBOXCREATE)

parent:(WORD) Is the name of the DIALOG window that is to own this new CHECKBOX control. 
group:(WORD) Is the name of the GROUPBOX control that is to be associated with this new CHECKBOX control. 
name:(WORD) Is used to identify this COMBOBOX control and MUST be unique. 
label:(LIST) Is used as the label of this new CHECKBOX control. 
xpos:(INTEGER) Is the X position you wish to place the upper left corner of the new CHECKBOX control. 
ypos:(INTEGER) Is the Y position you wish to place the upper left corner of the new CHECKBOX control. 
width:(INTEGER) Is the width of the new CHECKBOX control. 
height:(INTEGER) Is the height of the new CHECKBOX control.

Example (in a new window):

to checkonthings
  ifelse checkboxget "myhideturtle [ht] [st]
  end
end

windowcreate "main "mywindow "mytitle 0 0 100 100 []
groupboxcreate "mywindow "mygroupbox 10 10 80 40
checkboxcreate "mywindow "mygroupbox "myhideturtle [Hide Turtle] 20 20 60 20
buttoncreate "mywindow "mybutton "GO 40 50 25 25 [checkonthings]
windowdelete "mywindow

Example (in the graphics window):

to checkonthings
  ifelse checkboxget "myhideturtle [ht] [st]
  end
end

groupboxcreate "main "mygroupbox 110 90 140 60
checkboxcreate "main "mygroupbox "myhideturtle [Hide Turtle] 120 80 120 40
buttoncreate "main "mybutton "GO 160 20 50 50 [checkonthings]
windowdelete "main

# CHECKBOXCREATE
$ CHECKBOXCREATE
% CHECKBOXCREATE
* LOGO:0
# $ k • CHECKBOXDELETE

CHECKBOXDELETE name

This command will delete (close) the CHECKBOX control with the given name.

name (WORD) is used to identify the CHECKBOX you want destroyed.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
groupboxcreate "mywindow "mygroupbox 10 10 80 40
checkboxcreate "mywindow "mygroupbox "mycheckbox [Check Me] 20 20 60 20
checkboxdelete "mycheckbox
windowdelete "mywindow
truth CHECKBOXGET name

This command will solicit (ask) the CHECKBOX, for its state (True or False).

truth: (BOOLEAN) Represents the state (True or False) of the CHECKBOX control.

name: (WORD) is used to identify the COMBOBOX control you wish to solicit.

Example:

to checkothings
ifelse checkboxget "myhideturtle [ht] [st]
end
windowcreate "main "mywindow "mytitle 0 0 100 100 []
groupboxcreate "mywindow "mygroupbox 10 10 80 40
checkboxcreate "mywindow "mygroupbox "myhideturtle [Hide Turtle] 20 20 60 20
buttoncreate "mywindow "mybutton "GO 40 50 25 25 [checkothings]
windowdelete "mywindow
CHECKBOXSET name state

This command will set the state of the CHECKBOX with state (True or False).

name (WORD) is used to identify the CHECKBOX control you wish to SET to.
state (WORD) is the state you wish to set the CHECKBOX control to.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
groupboxcreate "mywindow "mygroupbox 10 10 80 40
checkboxcreate "mywindow "mygroupbox "mycheckbox [Check Me] 20 20 60 20
checkboxset "mycheckbox "true
checkboxset "mycheckbox "false
windowdelete "mywindow

# $ K * CHECKBOXSET

$ CHECKBOXSET

K CHECKBOXSET

* LOGO:0
RADIOBUTTON COMMANDS

RADIOBUTTONCREATE
RADIOBUTTONDELETE
RADIOBUTTONGET
RADIOBUTTONSET

* RADIOBUTTON_COMMANDS
+ RADIOBUTTON_COMMANDS
- RADIOBUTTON_COMMANDS
- LOGO:0
# $ K • RADIOBUTTONCREATE

RADIOBUTTONCREATE parent group name label xpos ypos width height

This command will create a RADIOBUTTON control. A RADIOBUTTON control is used to give the user a selection of a two state (True or False) item. But along with this the user will be restricted to only have one RADIOBUTTON set True within a GROUPBOX at any given time. A RADIOBUTTON must also be associated with a GROUPBOX (GROUPBOXCREATE).

parent:(WORD) Is the name of the DIALOG window that is to own this new RADIOBUTTON control.
group:(WORD) Is the name of the GROUPBOX control that is to be associated with this new RADIOBUTTON control.
name:(WORD) Is used to identify this RADIOBUTTON control and MUST be unique.
label:(LIST) Is used as the label of this new RADIOBUTTON control.
xpos:(INTEGER) Is the X position you wish to place the upper left corner of the new RADIOBUTTON control.
ypos:(INTEGER) Is the Y position you wish to place the upper left corner of the new RADIOBUTTON control.
width:(INTEGER) Is the width of the new RADIOBUTTON control.
height:(INTEGER) Is the height of the new RADIOBUTTON control.

Example (in a new window):

to checkonthings
ifelse radiobuttonget "myhideturtle [ht] [st]
end

windowcreate "main "mywindow "mytitle 0 0 100 100 []
groupboxcreate "mywindow "mygroupbox 10 10 80 60
radiobuttoncreate "mywindow "mygroupbox "myhideturtle [Hide Turtle] 20 20 60 20
radiobuttoncreate "mywindow "mygroupbox "myshowturtle [Show Turtle] 20 40 60 20
radiobuttonset "myhideturtle "true
radiobuttonset "myshowturtle "false
buttoncreate "mywindow "mybutton "GO 40 70 25 20 [checkonthings]
windowdelete "mywindow

Example (in the graphics window):

to checkonthings
ifelse radiobuttonget "myhideturtle [ht] [st]
end

groupboxcreate "main "mygroupbox 110 130 140 100
radiobuttoncreate "main "mygroupbox "myhideturtle [Hide Turtle] 120 120 120 40
radiobuttoncreate "main "mygroupbox "myshowturtle [Show Turtle] 120 80 120 40
radiobuttonset "myhideturtle "true
radiobuttonset "myshowturtle "false
buttoncreate "main "mybutton "GO 160 20 50 50 [checkonthings]
;windowdelete "main

* RADIOBUTTONCREATE
1 RADIOBUTTONCREATE
2 RADIOBUTTONCREATE
3 RADIOBUTTONCREATE
4 LOGO:0
# $K • RADIOBUTTONDELETE

RADIOBUTTONDELETE name

This command will delete (close) the RADIOBUTTON control with the given name.

name (WORD) is used to identify the RADIOBUTTON you want destroyed.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100 []
groupboxcreate "mywindow "mygroupbox 10 10 80 60
radiobuttoncreate "mywindow "mygroupbox "myradiobutton [Switch Me] 20 20 60 20
radiobuttondelete "myradiobutton
windowdelete "mywindow
truth RADIOBUTTONGET name

This command will solicit (ask) the RADIOBUTTON, for its state (True or False).

truth:(BOOLEAN) Represents the state (True or False) of the RADIOBUTTON control.

name:(WORD) is used to identify the RADIOBUTTON control you wish to solicit.

Example:

to checkonthings
 ifelse radiobuttonget "myhideturtle [ht] [st]
 end
windowcreate "main "mywindow "mytitle 0 0 100 100 []
groupboxcreate "mywindow "mygroupbox "myhideturtle [Hide Turtle] 10 10 80 60
radiobuttoncreate "mywindow "mygroupbox "myhideturtle [Hide Turtle] 20 20 60 20
radiobuttoncreate "mywindow "mygroupbox "myshowturtle [Show Turtle] 20 40 60 20
radiobuttonset "myhideturtle "true
radiobuttonset "myshowturtle "false
buttoncreate "mywindow "mybutton "GO 40 70 25 20 [checkonthings]
windowdelete "mywindow
# $ K • RADIOBUTTONSET

RADIOBUTTONSET name state

This command will set the state of the RADIOBUTTON with state (True or False). Note that even though the user can only have one RADIOBUTTON set True at any given time this can be violated through this command. If you choose to use this command you must maintain a correct state. That is, if you set a RADIOBUTTON True make sure you set all the other RADIOBUTTONs within the GROUPBOX to False.

name: (WORD) is used to identify the RADIOBUTTON control you wish to SET to.
state: (BOOLEAN) is the state (True or False) you wish to set the RADIOBUTTON control to.

Example:

to checkonthings
ifelse radioobuttonget "myhideturtle [ht] [st]
end

windowcreate "main "mywindow "mytitle 0 0 100 100 []
groupboxcreate "mywindow "mygroupbox "myhideturtle [Hide Turtle] 20 20 60 20
radioobuttoncreate "mywindow "mygroupbox "myhideturtle [Hide Turtle] 20 20 60 20
radioobuttoncreate "mywindow "mygroupbox "myshowturtle [Show Turtle] 20 40 60 20
radioobuttonset "myhideturtle "true
radioobuttonset "myshowturtle "false
buttoncreate "mywindow "mybutton "GO 40 70 25 20 [checkonthings]
windowdelete "mywindow

* RADIOBUTTONSET
* RADIOBUTTONSET
& RADIOBUTTONSET
* LOGO:0
DEBUG COMMANDS

DEBUGWINDOWS
# DEBUGWINDOWS

**DEBUGWINDOWS**

**(DEBUGWINDOWS name)**

This command will print the tree (window hierarchy) starting at the window called `name` if supplied otherwise it will list all Windows and controls.

`name:(WORD)` is used to identify the root Window you wish print.

```plaintext
windowcreate "main "mywindow "mytitle 0 0 100 100 []
listboxcreate "mywindow "mylist 25 0 50 50
buttoncreate "mywindow "mydraw "Draw 25 50 50 25 [Print "Click"
debugwindows
Window mywindow
    Button mydraw
    ListBox mylist
windowdelete "mywindow
```
Modal vs. Modeless Windows

Windows programming supports two modes, Modal and Modeless. The Modal mode (DIALOGCREATE) is similar to a non-windows programming model (the application is in control). Similar, in that, in midstream of processing you, as the programmer, decide to prompt the user for information (e.g., readlist). That is, processing is halted until the information is acquired and other components of the application are inaccessible to the user. For example prompting the user for a file name to open a document is Modal.

In the Modeless mode (WINDOWCREATE) the tables are turned, the Window (user) is in control. For example the commander in LOGO is Modeless. This takes some getting used to but is a very important idea. The program is now idle while. The application executes when the user triggers an event (such as pushing a button).

Full custom Windows are available in Modeless and Modal mode. Besides READLIST and READCHAR the following section describes Built-in Modal windows that are available.
$K$ * Predefined Windows

Predefined windows allows you access to several very common dialogs that are available under Windows.

MESSAGEBOX
YESNOBOX
SELECTBOX
QUESTIONBOX
DIALOGFILEOPEN
DIALOGFILESAVE
WINDOWFILEEDIT

※ Predefined_Windows
※ Predefined_Windows
※ Predefined_Windows
※ Predefined_Windows
※ LOGO:0
# K • MESSAGEBOX

MESSAGEBOX banner body

This command will stop processing and popup a message window using **banner** as the title bar and **body** as the message. Processing will not continue until the user clicks on the button. Note the LOGO commander is disabled until OK is clicked.

**banner**: (LIST) is used label the banner of the window.
**body**: (LIST) is used to fill the message box with the given text. The box will automatically be sized.

Example:

```
messagebox [This is the banner] [This is the body]
```
**SELECTBOX**

**choice** SELECTBOX banner choicelist

This command will stop processing and popup a message using *banner* as the title bar and *choicelist* as the choices. Processing will not continue until the user clicks on the **OK** or **CANCEL** button. Note the LOGO commander is disabled until a button is clicked. Once a **OK** is clicked this command will output an index corresponding to the current element selected (1 for choice 1, 2 for choice 2) and 0 if the user clicks on **CANCEL**.

**choice**:(INTEGER) Is 0 if the user chooses cancel and if the user clicks **OK** then 1 for choice 1, 2 for choice 2, and etc.

**banner**:(LIST) Is used label the banner of the window.

**choicelist**:(LIST) Is a list of items that the user can choose from.

Example:

```
show selectbox [Choose Color] [Red Green Blue]
<select green and hit **OK**>
2
show selectbox [Which are colors] [[Red Green Blue] [One Two Three] None]
1
```

---

# SELECTBOX

# SELECTBOX

# SELECTBOX

# SELECTBOX

# LOGO:0
YESNOBOX

yesorno YESNOBOX banner body

This command will stop processing and popup a message using `banner` as the title bar and `body` as the message. Processing will not continue until the user clicks on the Yes or No button. Note the LOGO commander is disabled until a button is clicked. Once a button is clicked this command will output TRUE for Yes and FALSE for No.

`yesorno:` (BOOLEAN) Is TRUE if Yes was clicked and FALSE if No was clicked.

`banner:` (LIST) Is used label the banner of the window.
`body:` (LIST) Is used to fill the message box with the given text. The box will automatically be sized.

Example:

`show yesnobox [Question] [Do you like logo?]`
  `true`
# $K • QUESTIONBOX

data QUESTIONBOX banner body

This command will stop processing and popup a message using banner as the title bar and body as the message. Processing will not continue until the user clicks on the Ok or Cancel button. Note the LOGO commander is disabled until a button is clicked. Once a button is clicked this command will output what the user typed for a response.

data:(LIST) is a list containing the users response. The empty list is output if the user clicks Cancel.

banner:(LIST) is used to label the banner of the window.

body:(LIST) is used to fill the message box with the given text. The box will NOT automatically be sized so keep it short.

Example:

show questionbox [Question] [Do you like logo?]

[Yes I do.]
DIALOGFILEOPEN

fullpath DIALOGFILEOPEN filename

This command will pop up a standard "FILE OPEN" windows dialog box. It does NOT open a file it just allows the user to select the desired file using a Graphical User Interface. The filename argument is used to set the defaults of name, directory and/or extension. The output is the fully specified file name that the user selected. In the case that the user selects cancel the output will be an empty list ({}).

fullpath:(LIST) The fully specified filename (includes drive and directory).

filename:(WORD) Specified the default directory, filename and/or extension of the filename. ("**" are permitted as the filename or extension).

Example:

show dialogfileopen "c:\\logo\\examples\\*.me
This will list all files ending in ".me" in the specified directory
[c:\\logo\\examples\\read.me]
fullpath DIALOGFILESAVE filename

This command will pop up a standard "File SaveAs" windows dialog box. It does NOT save a file just allows the user to select the desired file using a Graphical User Interface. The filename argument is used to set the defaults of name, directory and/or extension. The output is the fully specified file name the user selected. In the case that the user selects cancel the output will be an empty list ([]).

fullpath:(LIST) The fully specified filename (includes drive and directory).

filename:(WORD) Specified the default directory, filename and/or extension of the filename. ("**" are permitted as the filename or extension).

Example:

show dialogfilesave "c:\\logo\\examples\\*.me"
<This will list all files ending in ".me" in the specified directory>
[c:\\logo\\examples\\read.me]
WINDOWFILEEDIT

filename callback

This command will pop up a standard editor on the given filename. When the user exits the editor the contents of callback will be executed.

filename: (WORD) Specifies the file to be edited.
callback: (LIST) Is a (short) list of logo commands (or a procedure name) to execute when the user exits the editor.

Example:

windowfileedit dialogfileopen "c:\logo\examples\*.me [Print "Done"]
## $K$ WINDOWS Example

Note: That for the modeless case "setup" can be called after the WINDOWCREATE returns. And that for the modal case "setup" has to be called DURING the DIALOGCREATE because it does not return until the window is closed.

to win
; For modeless example use this line
windowcreate "root" "d1 [Draw Pictures] 0 0 150 110 [ ] setup ;Create main window
; For modal example use this line
; dialogcreate "root" "d1 [Draw Pictures] 0 0 150 110 [setup] ;Create main window
end
to setup
staticcreate "d1 "st4 [Select Shape] 5 10 50 10 ; Label the List box
listboxcreate "d1 "ll 5 25 80 40 ;Create List box with 3 Items owned by d1
listboxaddstring "ll "SQUARE
listboxaddstring "ll "TRIANGLE
listboxaddstring "ll "HEXAGON

staticcreate "d1 "st11 [Red] 100 10 40 10 ;Label the scrollbar
scrollbarcreate "d1 "s1 100 25 10 50 [myred] ;Create scroll bar, call myred when clicked
scrollbarset "s1 1 255 125 myred ;Init

buttoncreate "d1 "b1 "END 5 80 40 10 [myend] ;Create button to call myend
buttoncreate "d1 "b3 "CLEAR 55 80 35 10 [cs] ;Create button to clear
buttoncreate "d1 "b2 "DRAW 100 80 35 10 [drawthing] ;Create button to call drawthing
end

; execute this routine when DRAW button pushed
to drawthing
setpencolor (list scrollbarget "s1 0 0") ;Ask scrollbar what to setpencolor to

; Draw appropriate shape according to the listbox
if equalp [HEXAGON] listboxgetselect "ll [repeat 6 [fd 100 rt 60]]
if equalp [SQUARE] listboxgetselect "ll [repeat 4 [fd 100 rt 90]]
if equalp [TRIANGLE] listboxgetselect "ll [repeat 3 [fd 100 rt 120]]
end

; execute this routine when END button is pushed
to myend
; For modeless example use this
windowdelete "d1
; For modal example use this
; dialogdelete "d1
end

; execute this routine when RED scroll bar is adjusted
to myred
staticupdate "still sentence [Red] scrollbarget "al ;Update static label of position
end
**BITMAP FUNCTIONS**

Bitmap functions allow you to manipulate sub-images within the primary image.

**BITMAP COMMANDS**
# $K$ BITMAP COMMANDS

BITCUT
BITCOPY
BITPASTE
BITPASTETOINDEX
BITFIT
SETBITINDEX
BITINDEX
SETBITMODE
BITMODE
BITBLOCK
BITLOAD
BITSAVE
BITSIZE
GIFLOAD
GIFSAVE
GIFSIZEx
SEITAIVEAREA
ACTIVEAREA
BITCUT width height

This command will "cut" out part of the image and put it into Logo's memory (Clipboard if the BITINDEX is 0). Later at anytime you can "paste" (BITPASTE) it back into the image. LOGO will cut starting at the turtles' position with a width of the first argument and a height of the second argument. See also SETBITINDEX.

width:(INTEGER) Specifies the width of the bitmap to cut.
height:(INTEGER) Specifies the height of the bitmap to cut.

Note: A you might think these functions are off by 1 pixel in width and height when you go to use them, but it really is not. The reason is if you draw a square with say "repeat 4 [fd 2 rt 90]" how many pixels tall is it? It's 3 Pixels tall not 2. The reason for the is complicated, but in short, the way logo draws is that it has an imaginary line going down the center of each pixel. If a pixel is intersected (hit), then it is lit up. The distance from the center of pixels representing the left side of the square to the center on the right of the square is the correct distance (2 pixels). But we have to either turn the whole pixel on or off (there is no in between). Logo turns them on to show the line. This adds 1/2 a pixel to each side making the square 3x3 pixels in size. Bitmaps sizes are not from "center pixel to center pixel" like lines, they are whole pixel to whole pixel (inclusive).

Example:

setpensize [2 2]
repeat 72 [repeat 4 [fd 100 rt 90] setpencolor (list repcount*3 0 0) rt 5]
pu
setxy -50 -50
bitcut 100 100
cs
pu
repeat 36 [fd 150 bitpaste bk 150 rt 10]
BITCOPY

BITCOPY width height

This command will "copy" part of the image and put it into Logo's memory (Clipboard if the BITINDEX is 0). Later at anytime you can "paste" (BITPASTE) it back into the image. LOGO will copy starting at the turtles' position with a width of the first argument and a height of the second argument. See also SETBITINDEX.

width:(INTEGER) Specifies the width of the bitmap to copy.
height:(INTEGER) Specifies the height of the bitmap to copy.

Example:

setpensize [2 2]
repeat 72 [repeat 4 [fd 100 rt 90] setpencolor (list repcount*3 0 0) rt 5]
pu
setxy -50 -50
bitcopy 100 100
cs
pu
repeat 36 [fd 150 bitpaste bk 150 rt 10]
**# S K #** BITPASTETOINDEX

BITPASTETOINDEX index x y

This command will "paste" back to the coordinate x, y of the bitmap at index, the image that was "cut" (BITCUT) into the current index (or in the Clipboard if index is 0). See also SETBITINDEX and SETBITMODE. Don't confuse the indexes here, there is the current index and the index you wish to paste to. The coordinate x, y specifies were to place the lower left-hand corner of the image to be pasted relative to the lower left-hand corner of the destination image.

Why would you want such a command? Because they sky is the limit using this command for fast clean (flicker free) animation. For example, let's say you have a ball rolling across a background. You have to erase the ball, put the background back and draw the ball in it's new position. The problem is the user will see you erase the ball and will momentarily see a screen with no ball at all. This causes flicker, if you do everything in memory (which this command allows you to do) and only draw (paste) to the screen, the ball in position 1 immediately followed by the ball in position 2 (with position 1 all cleaned up) the user won't see flicker.

Note, the destination index must already have an image to paste to.

index:(INTEGER) Specifies which bitmap you wish to paste to.

x:(INTEGER) Specifies the x coordinate to place with in the destination index.
y:(INTEGER) Specifies the y coordinate to place with in the destination index.

Example:

```
; Draw a red block 100x100 in Bitmap Buffer 1
setbitindex 1
setfloodcolor [255 0 0]
bitblock 100 100
bitcopy 100 100
; Draw a green block 50x50 in Bitmap Buffer 2
cs
setbitindex 2
setfloodcolor [0 255 0]
bitblock 50 50
bitcopy 50 50
; Reserve 100x100 space in Bitmap Buffer 3
cs
setbitindex 3
bitcopy 100 100
; Copy 1 to 3
setbitindex 1
bitpastetoindex 3 0 0
; Copy 2 to 3
setbitindex 2
bitpastetoindex 3 25 25
; Paste 3 to Screen
setbitindex 3
bitpaste
```
BITPASTE

BITPASTE

This command will "paste" back into the image what was "cut" (BITCUT) (or what is in the Clipboard if BITINDEX is 0). LOGO will always "paste" at the location of the turtle with the turtle being the lower left corner. See also SETBITINDEX and SETBITMODE.

Example:

setpensize [2 2]
repeat 72 [repeat 4 [fd 100 rt 90] setpencolor (list repcount*3 0 0) rt 5]
pu
setxy -50 -50
bitcut 100 100
cs
pu
repeat 36 [fd 150 bitpaste bk 150 rt 10]

* BITPASTE
* BITPASTE
* BITPASTE
* LOGO:0
BITFIT width height

This command will "fit" the currently "cut" (BITCUT or BITCOPY) image into the specified dimensions. Later at anytime you can "paste" (BITPASTE) it back into the image. LOGO will fit the "cut" image to a width of the first argument and a height of the second argument. The original "cut" image is replaced by this newly "fit" image. You can permanently "scale" your image with bitfit. Whereas zoom only views it temporarily at a different scale.

width:(INTEGER) Specifies the width of the bitmap to fit to.
height:(INTEGER) Specifies the height of the bitmap to fit to.

Example:

setpensize [2 2]
repeat 72 [repeat 4 [fd 100 rt 90] setpencolor (list repcount*3 0 0) rt 5]
pu
setxy -50 -50
bitcut 100 100
cs
bitpaste
cs
bitfit 200 100
bitpaste

* BITFIT
* BITFIT
* BITFIT
* BITFIT
* LOGO:0
SETBITINDEX

SETBITINDEX index

This command sets the current bitmap cut buffer according to index. The index can range from 0 to 1023. Its purpose was to allow multiple images to be stored in memory ready for quick pasting in animation. The index of 0 is the default and also has the behavior of using the CLIPBOARD as the cut buffer. That is if you "cut" and image in PAINT you can paste it directly in MSWLogo. The reverse is also true, if "cut" an image in MSWLogo it is available to PAINT. Use `BITINDEX` to obtain the current setting.

index: (INTEGER) Specifies which bitmap you wish to select.

Example:

```
setbitindex 0
repeat 3 [fd 50 rt 120]
bitcut 100 100
cs
setbitindex 1
repeat 4 [fd 50 rt 90]
bitcut 100 100
cs
setbitmode 3
pu
ht
repeat 72 [fd 50 bitpaste bk 50 rt 5]
setbitindex 0
repeat 72 [fd 100 bitpaste bk 100 rt 5]
pd
```
BITINDEX

This command will output the current bitmap index set by \texttt{SETBITINDEX}.

\texttt{index \{INTEGER\}} the currently selected index.

Example:

\begin{verbatim}
setbitindex 99
show bitindex
99
\end{verbatim}
SETBITMODE

This command sets the current bitmap mode according to mode. The mode can range from 1 to 9. Its purpose is to allow images to be pasted using different methods. Sometimes you want the background erased and sometimes not. Sometimes you wish to invert the image before pasting it and sometimes not. Use BITMODE to obtain the current setting. These are the 9 methods:

mode 1: Take COPY of Memory then COPY to Screen.
mode 2: Take COPY of Memory OR with COPY of Screen then COPY to Screen.
mode 3: Take COPY of Memory AND with COPY of Screen then COPY to Screen
mode 4: Take COPY of Memory XOR with COPY of Screen then COPY to Screen
mode 5: Take COPY of Memory AND with INVERT of Screen then COPY to Screen
mode 6: Take INVERT of Memory then COPY to Screen
mode 7: Take COPY of Memory OR with COPY of Screen then INVERT to Screen
mode 8: Take INVERT of Memory OR with COPY of Screen then COPY to Screen
mode 9: Take INVERT of Screen then COPY to Screen

mode: (INTEGER) Specifies which mode you wish to select.

Example:

; Set screen to white
setscreencolor [255 255 255]
cs
pd
; Draw a circle and fill it with red
circle 50
setfloodcolor [255 0 0]
fill
pu
; Position to lower left of bounding rectangle
selxy -50 -50
bitcopy 101 101
; Set screen to yellow
setscreencolor [255 255 0]
cs
; Set paste mode to COPY
setbitmode 1
bitpaste
; Notice how the white background still exists
cs
; Set paste mode to AND
setbitmode 3
bitpaste
; Notice the white background disappeared
```
# $K • BITMODE

mode BITMODE

This command will output the current bitmap mode set by SETBITMODE.

mode:(INTEGER) the currently selected mode.

Example (save and restore bitmode):

    setbitmode 1
    make "savemode bitmode
    setbitmode 8
    show bitmode
    8
    setbitmode :savemode
    show bitmode
    1
```
BITBLOCK width height

This command will draw an opaque rectangle of the given dimensions. The color will be the color of SETFLOODCOLOR.

width:(INTEGER) Specifies the width of the block.
height:(INTEGER) Specifies the height of the block.

Example:

bitblock 200 100
bitblock 100 200
BITLOAD

BITLOAD bitmapname

This command will load the bitmap specified by its one input that must be a word that describes the bitmap file to load. This command is the same as the Bitmap Load Command from the menu. See also BITSAVE command.

bitmapname: (WORD) File name of bitmap to load.

Example:

show bitload "c:\windows\forest.bmp"
**BITSIZE**

`size BITSIZE bitmapname`

This command will output the size of the bitmap specified by its one input that must be a word that describes the bitmap file to get the size of.

`size:`(LIST) of 2 integers representing [Width Height] of bitmap loaded.

`bitmapname:`(WORD) File name of bitmap to get size of.

Example:

```
show bitsize "c:\windows\forest.bmp
[256 256]
```
# $ K * BITSAVE

**BITSAVE**

**BITSAVE bitmapname**

**(BITSAVE bitmapname maxcolordepth)**

This allows you to save a PICTURE (bitmap image) of your work on the computer's disk so that the computer will not forget it. It also allows you to add more work to an existing piece of work. REMEMBER if your image was generated with a LOGO program you really don’t need to save it as an image unless you want to use the image in another application such as Paint or as a Wallpaper. Its one input must be a word that describes the bitmap file to save. See also **BITLOAD** and **ACTIVEAREA** commands.

The format of the file you save things in, is known as, a Microsoft Windows Bitmap (.BMP). You can interchange these files with other applications such as Paint. Note that these files can be BIG and can take a while to read or write.

**bitmapname**: (WORD) File name of bitmap to save.

**maxcolordepth**: (INTEGER) Maximum Color Depth in Bits Per Pixel (1 = 2 colors, 4 = 16 colors, 8 = 256 colors, 24 = 16.7 million colors) default is 24.

Example:

```
repeat 72 [repeat 4 [fd 100 rt 90] rt 5]
bitsave "myfile.bmp
cs
bitload "myfile.bmp
```

* BITSAVE
* BITSAVE
* BITSAVE
* LOGO:0
GIFLOAD

GIFLOAD bitmapname

This command will load the GIF picture (compressed bitmap image) specified by its one input that must be a word that describes the bitmap file to load. This command is the same as the Bitmap Load Command from the menu. See also GIFSAVE command.

bitmapname:(WORD) File name of GIF to load.

Example:

show gifload "myfile.gif"
GIFSIZE

size GIFSIZE bitmapname

This command will output the size of the GIF picture (compressed bitmap image) specified by its one input that must be a word that describes the GIF file to get the size of.

size:(LIST) of 2 integers representing [Width Height] of bitmap loaded.

bitmapname:(WORD) File name of GIF to get size of.

Example:

show gifsizem "myfile.gif"
[256 256]
GIFSAVE

GIFSAVE bitmapname
(GIFSAVE bitmapname delay append loop maxcolor transcolorvector)

This command allows you to save a GIF picture (compressed bitmap image) of your work on the computer's disk so that it can be used on Web pages or Email. In its first form the one input must be a word that describes the bitmap file to save.

The second form allows multiple options:

You can save a GIF animation frame (series of compressed bitmap images). The delay specifies how long in milliseconds to display the frame. The append argument specifies whether your starting a new animation or appending to an existing animation. The loop argument specifies how many times the animation should loop (the default of -1 = no loop, 0 = loop forever, n = loop n times).

You can specify the maximum color depth with maxcolor, the choices are 1 (2 colors), 4 (16 colors), or 8 (256 colors) bits per pixel (8 being the default). If your image does not use many colors you can reduce the size of the GIF file produced significantly by using this option.

You can specify which color in the image you want to be transparent in the form [RED GREEN BLUE] (for this to work reliably you should be in 256 color Video mode).

See also GIFLOAD and ACTIVEAREA commands.

This commands saves in a format known as GIF (Graphics Interchange Format) and was created by CompuServe. The format is popular on the WWW (World Wide Web). It is popular for several reasons. One reason is that it is a compressed format and takes little time to transmit. A second reason is that it supports "Interfacing". Interfacing makes the image partially viewable as it's being transmitted (it appears to be blurry and comes into focus). Another reason is animated GIFs. Animated GIFs allow multiple images in the one file with specified delays between the images.

bitmapname:(WORD) File name of GIF to save.
delay:(INTEGER) Delay in millisecond for frame.
append:(BOOLEAN) True to append Frame False for new animation.
loop:(INTEGER) Looping control (-1 = no loop (default) , 0 = loop forever, n = loop n times).
maxcolor:(INTEGER) Maximum color depth in bits per pixel (1 (2 colors), 4 (16 colors), 8 (256 colors, default)).
transcolorvector:(LIST) List of three integers representing Red, Greed, Blue intensities each in the range 0-255 that you wish to set as the transparent color.

Notes:

GIFSAVE will work best when your computer is in 256 color mode.
You can call CLEARPALETTE between each frame because each frame has it's own "Color Palette".

# GIFSAVE
$ GIFSAVE
K GIFSAVE
* LOGO:0
Example (Single Image):

repeat 72 [repeat 4 [fd 100 rt 90] rt 5]
gifsave "myfile.gif"
cs
gifload "myfile.gif"

Example (GIF Animation):

cs
setactivearea [-150 -150 150 150]
; Flag to indicate we do not append the first frame
make "append "False
repeat 36 ~
| repeat 4 [fd 100 rt 90]
  rt 10
  ; Save a frame (no delay and loop forever)
  (gifsave "myfile.gif 0 :append 0)
  make "append "True
|

<Now Open the file myfile.gif with your Internet Browser (e.g. NetScape or Internet Explorer)>
# $ K • SETACTIVEAREA

SETACTIVEAREA area

This is the same the Bitmap Active Area Command from the menu. It sets the active area for printing and saving Bitmaps. See also ACTIVEAREA.

area: (LIST) of 4 integers representing [XLow YLow XHigh YHigh].

Example:

bitblock 100 100
setactivearea [0 0 50 50]
bitsave "myfile.bmp
cs
bitload "myfile.bmp
; See how only part of the image was restored

* SETACTIVEAREA
$ SETACTIVEAREA
K SETACTIVEAREA
* LOGO:0
**ACTIVEAREA**

**area** ACTIVEAREA

This command will output the current active area set by **SETACTIVEAREA**

**area**:(LIST) of 4 integers representing [XLow YLow XHigh YHigh].

Example:

```
setactivearea [-10 -10 10 10]
show activearea
[-10 -10 10 10]
```
Networking allows MSWLogo to communicate with the rest of the world. There are numerous forms of networking and MSWLogo uses just one form. This form is known as TCP/IP sockets (this is primarily what is used to build the Internet). MSWLogo can receive data from other computers or send data to other computers. The other computer does not need to be running MSWLogo although it will be likely. When you wish MSWLogo to initiate a connection use the NETCONNECTxxx commands. When you wish MSWLogo to accept a connect use NETACCEPTxxx commands. You can send and receive data on either type of connection. You can at most have 2 connections, one you initiated and one you accepted.

MSWLogo Networking assumes that the TCP/IP protocol is available and that it is available through what is known as WINSOCK.DLL (16bit MSWLogo) or WSOCK32.DLL (32bit MSWLogo). Microsoft offers TCP/IP protocol for Windows 3.11 for free. Microsoft includes TCP/IP protocol in Windows 95 and Windows NT. But it does have to be explicitly installed and setup on all platforms. There is also a public domain TCP/IP protocol known as Trumpet that can be used for Windows 3.1.

TCP/IP networks can run over a wide variety of devices even modems (PPP), LapLink cables etc. CompuServe, Microsoft Network, America Online and private Internet Service Providers (ISPs) all offer TCP/IP access to the rest of the world. What this means is that your friend can be on the west cost and you can be on the east coast and both run MSWLogo and draw on each others screen or build games that you can both participate in. Assuming you can find one another.

NETSTARTUP
NETSHUTDOWN
NETACCEPTON
NETACCEPTOFF
NETACCEPTRECEIVEVALUE
NETACCEPTSENДVALUE
NETCONNECTON
NETCONNECTOFF
NETCONNECTSENДVALUE
NETCONNECTRECEIVEVALUE

Tools of the trade (outside of MSWLogo):

DNS (Domain Name Service): DNS is an integral part of using TCP/IP protocols. TCP/IP in itself only understands machine addresses that look something like 21.100.200.5 (note each number field must be less than 255). But numbers have no meaning and are hard to remember for us humans, so machines are assigned names. DNS is basically machines that maintain databases of these mappings of names to numbers. DNS machines talk to each other to share information. MSWLogo uses DNS names when trying to make a connection. DNS is not required to setup a small TCP/IP network keep reading.

PING: is a simple (usually command line program) used as a simple test of network connectivity. Most TCP/IP implementations include a PING program. To use it just enter "PING remotenode" where remote node is the machine you’d like to test connectivity to. PING will tell if it can reach that machine and if it can how long it will take. Until PING works don’t bother trying any other TCP/IP based software (including MSWLogo Network commands). PING will accept both forms of the address (the number or the name). If the PING fails with the name then try the number (if you know it). If the number works but the name does not it, it usually indicates a DNS problem.

No DNS: If there is no DNS server, you can use what is known as a HOSTS file (usually kept in C:\Windows). Microsoft frequently installs a sample HOSTS file called HOSTS.SAM (copy HOSTS.SAM to HOSTS. and edit it). It’s simply a database for mapping name addresses to numbers. If using a HOSTS file then each machine on the network should use a copy of the same HOSTS file with the name to mapping of all the machines in the network. You can have both DNS and a HOSTS file, usually the HOSTS file is checked before DNS is attempted.

* Networking_Commands
& Networking_Commands
# Networking_Commands
% LOGO:0
Sockets and Services: Services is to Sockets as DNS is to Network Addresses. Services map "Service Names" like
(TELNET, FTP, POPMAIL) to Socket port numbers. You may find a SERVICES file in C:\Windows and you can
browse it if you like. MSWLogo does not support the Service name to Socket port number it only uses Socket port
numbers. But if you wish to connect to a service such as TELNET or act as a TELNET server then just look it up in the
services file. In general socket numbers under 5000 are reserved for things such as TELNET and FTP. It is better to
use socket numbers greater than 5000.

Networks: If it’s a private network under your total control you can do what ever you want. If they are all Windows PC’s
then you’ll probably have to do without a DNS server (and use HOSTS file). Generally DNS servers are UNIX
machines. If your part of a larger network you should contact your network administrators. If you’re already networked
using Netware it is commonly possible to run both TCP/IP and Netware at the same time (in fact it’s a piece of cake on
Windows95).

Service Providers: Some service providers do not give you a name and assign your TCP/IP number address "on-the-
fly", I’m not sure how to deal with these systems. Some service providers completely "wrap" your network access up
into one application that gives all the services you need. And they might not offer WinSock access. When asking a
service providers for help tell them that you want to run something like NetScape®. If NetScape works then
MSWLogo should also work (outgoing anyway). Allowing you to receive data is more complex, the other application
needs to be able to find you by name or at least by a number address if they use a HOSTS file. If you know you
machine name but don’t know it address you can use PING to get it, just PING yourmachinename and it will report the
address.
NETSTARTUP

(\texttt{NETSTARTUP \texttt{synchronous\_dns}})

Turns on Networking in MSWLogo. Optionally you may call it with the \texttt{SYNCHRONOUS\_DNS} flag as \texttt{0} (Asynchronous Default) or \texttt{1} (Synchronous). Asynchronous DNS is safer to use (less likely to cause MSWLogo to hang during a network name lookup). But some WinSock stacks have problems with it (e.g. Trumpet WinSock). If you get "Host not Found" errors and you're confident that DNS is working (e.g. ping host works) then try setting this flag to \texttt{1}.

\texttt{synchronous\_dns:(INTEGER) 1} for Synchronous \texttt{0} for Asynchronous (default).

Example:

\texttt{NETSTARTUP}
\texttt{NETSHUTDOWN}

\[ # \texttt{NETSTARTUP} \]
\[ \$ \texttt{NETSTARTUP} \]
\[ \texttt{K NETSTARTUP} \]
\[ * \texttt{LOGO:0} \]
NETSHUTDOWN

Turns off Networking in MSWLogo.

Example:

NETSTARUP
NETSHUTDOWN
status NETACCEPTON socket sendready receiveready

This command will enable you to accept a connection over network on the socket. To obtain the data received call NETACCEPTON in your receiveready code. To turn off accept then call NETACCEPTOFF command. When data is available the receiveready code will be executed. When you actually send data with NETACCEPTSENDVALUE it might return false (meaning it’s not ready to send). If this happens you must wait until the sendready code is executed before trying to send the data again. Note you cannot just send data from the sendready callback, it won’t be executed unless the previous send was blocked.

status (BOOLEAN): Status as to whether command was successful.
socket (INTEGER) is a number used to identify which socket to listen on.
sendready (LIST) is a (short) list of logo commands (or a procedure name) to execute when data is ready to send again.
receiveready (LIST) is a (short) list of logo commands (or a procedure name) to execute when data is available.

The SENDREADY callback will be called in the following order for the following reasons:
1: Successfully Issued Accept
2: Accepted Connection
3: Ready To Send
4: Ready To Send (after a send Error)
5: Ready To Send (after a send Error)

n: Ready To Send (after a send Error)

The RECEIVEREADY callback will be called in the following order for the following reasons:
1: Actual Data Received
2: Actual Data Received

n: Actual Data Received

Example:

netstartup ; Start up the network
; Wait for someone to connect to you on socket 5124
show netaccepton 5124 [print [ready to send]] [print [ready to receive]]
true
netacceptoff
netshutdown
# NETACCEPTOFF

NETACCEPTOFF

This command will disable accepting incoming connections.

Example:

netstartup ; Start up the network
; Wait for someone to connect to you on socket 5124
show netaccepton 5124 [print [ready to send]] [print [ready to receive]]
true
netacceptoff
netshutdown
NETACCEPSENDVALUE

sent NETACCEPSENDVALUE data

This command will send the data out the accept socket. When you send data with this command it might return False (meaning it’s not ready to send). If this happens you must wait until the sendready callback code of NETACCEPT is executed and then try sending the data again.

sent: (BOOLEAN) Status as to whether data was successfully queued to go out.

data: (LIST) Is the value of the data you wish to send over the network.

Example: (The Acceptor Sends Data To Who Connected)

netstartup ; Start up the network
; Wait for someone to connect to you on socket 5124
show netaccepton 5124 [print [Ok to Send Again]] []
true
; Connect to MyMachineName on socket 5124 (myself)
show netconnecton "MyMachineName 5124 [ ] [print netconnectreceivevalue] true
wait 100 ; Wait little for connection to establish
Ok to Send Again
Ok to Send Again
Ok to Send Again
show netacceptsendvalue [Who are you that called me] ; Send some data true
wait 100 ; Wait a little before we shut things down
Who are you that called me
netconnectoff
netacceptoff
netshutdown
NETACCEPTRECEIVEVALUE

```plaintext
data NETACCEPTRECEIVEVALUE

This command will output the value of the last network data received on the accept socket. It is best to call this procedure in your receive ready callback code of your NETACCEPTON command. It will be lost on the next network accept receive callback.

data (LIST) is the value of the last network data received on the accept socket.

Example: (The Initiator (connector) Sends Data To Who Accepted the connection)

netstartup ; Start up the network
; Wait for someone to connect to you on socket 5124
show netaccepton 5124 [] [print netacceptreceivevalue] true
; Connect to MyMachineName on socket 5124 (myself)
show netconnection "MyMachineName" 5124 [print [Ok to Send Again]] [] true
wait 100 ; Wait little for connection to establish
Ok to Send Again
Ok to Send Again
Ok to Send Again
show netconnectsendvalue [Hello thanks for taking my call] ; Send some data true
wait 100 ; Wait a little before we shut things down
Hello thanks for taking my call
netconnectoff
netacceptoff
netshutdown
```

# NETACCEPTRECEIVEVALUE
$ NETACCEPTRECEIVEVALUE
K NETACCEPTRECEIVEVALUE
* LOGO:0
status NETCONNECTON machinename socket sendready receiveready

This command will enable you to initiate a connection over the network to the socket at machinename. To turn off the connection then call NETCONNECTOFF command. To obtain the data received call NETCONNECTRECEIVEVALUE in your receiveready callback code. When you actually send data with NETCONNECTSENDVALUE it might return false (meaning it's not ready to send). If this happens you must wait until the sendready callback code is executed before you try to send the data again. Note you cannot just send data from the sendready callback code, it won't be executed unless the previous send was blocked.

status:(BOOLEAN): Status as to whether command was successful.
machinename:(LIST) is a TCP/IP host name from DNS or a Hosts file.
socket:(INTEGER) is a number used to identify which socket to send on.
sendready:(LIST) is a (short) list of logo commands (or a procedure name) to execute when data is ready to send again.
receiveready:(LIST) is a (short) list of logo commands (or a procedure name) to execute when data is available.

The SENDREADY callback will be called in the following order for the following reasons:
1: Successfully Issued Connection
2: Connection Made
3: Ready To Send
4: Ready To Send (after a send Error)
5: Ready To Send (after a send Error)
.
.
.
n: Ready To Send (after a send Error)

The RECEIVEREADY callback will be called in the following order for the following reasons:
1: Actual Data Received
2: Actual Data Received
.
.
.
n: Actual Data Received

Example:(this example assumes remotemachinename is listening on socket 5124)

netstartup ; Start up the network
show netconnection "remotemachinename 5124 [print [ready to send again]] [print [I have something]]
true
netconnectoff
netshutdown

# NETCONNECTON
# NETCONNECTON
^K NETCONNECTON
*L LOGO:0
NETCONNECTOFF

This command will disable allowing you to send data.

Example:

netstartup ; Start up the network
show netconnecton "remotemachinename 5124 [print [ready to send again]] [print [I have something]]
true
netconnectoff
netshutdown
sent NETCONNECTSENDCALL

This command will send the DATA out the open socket. When you send data with this command it might return False (meaning it's not ready to send). If this happens you must wait until the sendready callback procedure of NETCONNECT() is executed and then try sending the data again.

sent: (BOOLEAN): Status as to whether data was successfully queued to go out.

data: (LIST) Is the value of the data you wish to send over the network.

Example: (The Initiator (connector) Sends Data To Who Accepted the connection)

netstartup ; Start up the network
; Wait for someone to connect to you on socket 5124
show netaccepton 5124 [] [print netacceptreceivevalue]

true
; Connect to MyMachineName on socket 5124 (myself)
show netconnection "MyMachineName 5124 [print [Ok to Send Again]] []

true
wait 100 ; Wait little for connection to establish
Ok to Send Again
Ok to Send Again
Ok to Send Again

show netconnectsendvalue [Hello thanks for taking my call] ; Send some data
true
wait 100 ; Wait a little before we shut things down
Hello thanks for taking my call
netconnectoff
netacceptoff
netshutdown
\# \$ \& \K NETCONNECTRECEIVEVALUE

data NETCONNECTRECEIVEVALUE

This command will output the value of the last network data received on the connect socket. It is best to call this procedure in your receive ready callback code of your NETACCEPTON command. It will be lost on the next network receive callback.

data:(LIST) is the value of the last network received data.

Example: (The Acceptor Sends Data To Who Connected )

netstartup ; Start up the network
; Wait for someone to connect to you on socket 5124
show netaccepton 5124 [print [Ok to Send Again]]) []
true
; Connect to MyMachineName on socket 5124 (myself)
show netconnecton *"MyMachineName 5124 [] [print netconnectreceivevalue]
true
wait 100 ; Wait little for connection to establish
Ok to Send Again
Ok to Send Again
Ok to Send Again
show netaccepstsendvalue [Who are you that called me] ; Send some data
ture
wait 100 ; Wait a little before we shut things down
Who are you that called me
netconnectoff
netacceptoff
netshutdown

\# NETCONNECTRECEIVEVALUE
\$ NETCONNECTRECEIVEVALUE
\& NETCONNECTRECEIVEVALUE
\K NETCONNECTRECEIVEVALUE
\* LOGO:0
# $ K * MULTI MEDIA/MIDI/SOUND

Multi Media in Logo means that you, as a logo programmer, can manipulate Multi Media devices such as cdplayers, sound boards, and more.

**MIDI COMMANDS**

**SOUND WAVE COMMANDS**

**SOUND COMMANDS**

**MULTI MEDIA COMMANDS**
MIDI COMMANDS

MIDIOPEN
MIDICLEOSE
MIDIMESSAGE
MIDI TABLE
MIDI CONTROL
MIDI GLOSSARY
MIDI INSTRUMENT
MIDIOPEN

device MIDIOPEN
device (MIDIOPEN id)

This command opens the MIDI device and accesses it through a MIDI device driver. The device driver chosen depends on several things. In form 1 (no arguments) the MIDI-MAPPER is attempted to be used. In form 2 (id argument specified) lets you choose any available MIDI driver available on your system. The "id" starts at 0 up to 1 less than number of MIDI drivers available. To determine which "id" maps to which driver try several MIDIOPEN commands with increasing "id". MIDIOPEN will output the name of the driver being used.

device:(LIST) When successfully opened outputs the name of the driver being used.

id:(INTEGER) Is an index that specifies which MIDI device driver you wish to open. When no id given it selects the MIDI MAPPER device driver.

Basically MIDI allows you to generate sound on your sound card. You will need to install the appropriate drivers under Windows for your sound card to work. Most sound cards come with a MIDI player which basically reads MIDI messages from the file and passes them to the MIDI driver. If you have your MIDI player working under Windows then MSWLogo should work too.

MIDI commands in MSWLogo does not use MIDI files (.MID or .MDI). Instead you directly build up sequences of MIDI messages and send them directly to the MIDI device. Note, you can manipulate (play) MIDI files using the MCI command. MSWLogo is not a MIDI sequencer. You can think of MIDI commands in MSWLogo as a programmable keyboard. It just so happens that the link between your programmable keyboard (MIDI commands) and your Speaker is MIDI.

This is only available to Windows 3.1 or compatible systems. See also the MIDIMESSAGE and MIDICLOSE commands.

Example:

PRINT MIDIOPEN
[MIDI Mapper]
MIDICLOSE

* MIDIOPEN
* MIDIOPEN
* MIDIOPEN
* LOGO:0
MIDICLOSE

This command closes the MIDI device. There are no inputs or outputs. See also the MIDIOPEN command.

Example:

PRINT MIDIOPEN
[MIDI Mapper]
MIDICLOSE

# $ K • MIDICLOSE

* MIDICLOSE
$ MIDICLOSE
K MIDICLOSE
* LOGO:0
MIDIMESSAGE message

The input must be a list. You must already of issued a MIDOPEN command to use this command.

message:(LIST) A MIDI message in one of 3 forms explained below.

There are 3 forms of the MIDI message, the Short form, the Long form and the System Exclusive form.

Short form message:

[status data1 data2]

The Short form is the common form and always has a 3 integer list. The first integer is known as the STATUS BYTE (it can also be thought of as a COMMAND BYTE). It must be followed by 2 data bytes even if the message requires only 1 (just use 0).

status:(INTEGER) In MIDI terminology this is the "status byte" but I call it the "command code" in the table below.
data1:(INTEGER) Data byte 1 of the MIDI "command code".
data2:(INTEGER) Data byte 2 of the MIDI "command code".

Long form message:

[status data1 data2 status data1 data2 ...]

The Long form is similar to the Short form but integer list contains many short messages (triples).

System Exclusive form message:

[240 data1 data2 data3 data4 ...]

The SYSTEM EXCLUSIVE form must be led by the system exclusive status byte 240 (F0 hex). It can then be followed by any amount of data bytes.

data1:(INTEGER) First data byte specific to your midi device.
data2:(INTEGER) First data byte specific to your midi device.
and so on....

See the MIDI TABLE which basically is a Specification of the MIDI Message.

This documentation is not attempt to teach you MIDI. But there is enough information here to hopefully get you started. For more information you may be interested in purchasing a book on MIDI such as:

MIDI BASICS by Akira Otsuka and Akihiko Nakajima.

Example:

PRINT MIDOPEN
[MIDI Mapper]
MIDIMESSAGE (LIST 192+13 56 0 192+13 56 0)
<listen to tone>
MIDIMESSAGE (LIST 144+13 100 100)
MIDICLOSE
## MIDI TABLE

<table>
<thead>
<tr>
<th>COMMAND NAME</th>
<th>COMMAND CODE</th>
<th>DATA BYTE 1</th>
<th>DATA BYTE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note Off</td>
<td>128 + Channel</td>
<td>0-127 Pitch</td>
<td>0-127 Velocity</td>
</tr>
<tr>
<td>Note On</td>
<td>144 + Channel</td>
<td>0-127 Pitch</td>
<td>0-127 Velocity</td>
</tr>
<tr>
<td>Poly Pressure</td>
<td>160 + Channel</td>
<td>0-127 Pitch</td>
<td>0-127 Pressure</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>0-127 MIDI Control</td>
<td>0-127 MSB</td>
</tr>
<tr>
<td>Program Change</td>
<td>192 + Channel</td>
<td>0-127 Program</td>
<td>Not used</td>
</tr>
<tr>
<td>Channel Pressure</td>
<td>208 + Channel</td>
<td>0-127 Pressure</td>
<td>Not used</td>
</tr>
<tr>
<td>Pitch Wheel</td>
<td>224 + Channel</td>
<td>0-127 LSB</td>
<td>0-127 MSB</td>
</tr>
<tr>
<td>System Exclusive</td>
<td>240</td>
<td>0-127 Id Code</td>
<td>Any number of bytes</td>
</tr>
<tr>
<td>Undefined</td>
<td>241</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>Song Position</td>
<td>242</td>
<td>0-127 LSB</td>
<td>0-127 MSB</td>
</tr>
<tr>
<td>Song Select</td>
<td>243</td>
<td>0-127 Song</td>
<td>Not used</td>
</tr>
<tr>
<td>Undefined</td>
<td>244</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>Undefined</td>
<td>245</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>Tune Request</td>
<td>246</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>End of Exclusive</td>
<td>247</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>Timing Clock</td>
<td>248</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>Undefined</td>
<td>249</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>Start</td>
<td>250</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>Continue</td>
<td>251</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>Stop</td>
<td>252</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>Undefined</td>
<td>253</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>Active Sensing</td>
<td>254</td>
<td>Not used</td>
<td>Not used</td>
</tr>
<tr>
<td>System Reset</td>
<td>255</td>
<td>Not used</td>
<td>Not used</td>
</tr>
</tbody>
</table>

See also the [MIDI GLOSSARY](#)
# MIDI CONTROL

<table>
<thead>
<tr>
<th>COMMAND NAME</th>
<th>COMMAND CODE</th>
<th>DATA BYTE 1</th>
<th>DATA BYTE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>0 Undefined</td>
<td>0-127 MSB</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>1 Modulation Wheel</td>
<td>0-127 MSB</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>2 Breath Controller</td>
<td>0-127 MSB</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>3 After Touch</td>
<td>0-127 MSB</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>4 Foot Controller</td>
<td>0-127 MSB</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>5 Portamento Time</td>
<td>0-127 MSB</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>6 Data Entry</td>
<td>0-127 MSB</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>7 Main Volume</td>
<td>0-127 MSB</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>8-31 Undefined</td>
<td>0-127 MSB</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>32-63 LSB of 0-31</td>
<td>0-127 LSB</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>64 Damper Pedal</td>
<td>0:Off 127:On</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>65 Portamento</td>
<td>0:Off 127:On</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>66 Sostenuto</td>
<td>0:Off 127:On</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>67 Soft Pedal</td>
<td>0:Off 127:On</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>68-92 Undefined</td>
<td>0:Off 127:On</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>93 Chorus</td>
<td>0:Off 127:On</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>94 Celeste</td>
<td>0:Off 127:On</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>95 Phaser</td>
<td>0:Off 127:On</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>96 Data Entry + 1</td>
<td>0:Off 127:On</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>97 Data Entry - 1</td>
<td>0:Off 127:On</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>98-121 Undefined</td>
<td>0:Off 127:On</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>122 Local Control</td>
<td>0-127</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>123 All Notes Off</td>
<td>0</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>124 Omni Mode off</td>
<td>0-15</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>125 Omni Mode on</td>
<td>0</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>126 Mono on/Poly off</td>
<td>0</td>
</tr>
<tr>
<td>Control Change</td>
<td>176 + Channel</td>
<td>127 Poly on/Mono off</td>
<td>0</td>
</tr>
</tbody>
</table>

See also the [MIDI GLOSSARY](#)
**MIDI GLOSSARY**

Channel: A channel is a number from 0-15 which corresponds to channels 1-16.

Pitch: A pitch is a number from 0-127 and corresponds to a note on the instrument.

Velocity: A velocity is a number from 0-127 and corresponds to how fast the key (or string) is pressed or released (most terminology is in reference to keyboards). 0 means is released.

Pressure: A pressure is a number from 0-127 and corresponds to the characteristics of how the key is hit.

Program: A program is a number from 0-127 and corresponds to the instrument to use. See the [MIDI INSTRUMENT](#) table.

MSB: Most Significant Bits.

LSB: Least Significant Bits.

Id Code: Manufactures Id Code. Used to enter System Exclusive Mode which is specific to the Manufacturer of the device.

Song: A song is a rhythm machine.
# § K • MIDI INSTRUMENT

Piano

0 - Acoustic Grand Piano
1 - Bright Acoustic Piano
2 - Electric Grand Piano
3 - Honky-tonk Piano
4 - Rhodes Piano
5 - Chorused Piano
6 - Harpsichord
7 - Clavinet

Chromatic Percussion

8 - Celesta
9 - Glockenspiel
10 - Music box
11 - Vibraphone
12 - Marimba
13 - Xylophone
14 - Tubular Bells
15 - Dulcimer

Organ

16 - Hammond Organ
17 - Percussive Organ
18 - Rock Organ
19 - Church Organ
20 - Reed Organ
21 - Accordion
22 - Harmonica
23 - Tango Accordion

Guitar

24 - Acoustic Guitar (nylon)
25 - Acoustic Guitar (steel)
26 - Electric Guitar (jazz)
27 - Electric Guitar (clean)
28 - Electric Guitar (muted)
29 - Overdriven Guitar
30 - Distortion Guitar
31 - Guitar Harmonics

Bass

32 - Acoustic Bass
33 - Electric Bass (finger)

* MIDI_INSTRUMENT
* MIDI_INSTRUMENT
* § MIDI_INSTRUMENT
* LOGO:0
34 - Electric Bass (pick)  
35 - Fretless Bass  
36 - Slap Bass 1  
37 - Slap Bass 2  
38 - Synth Bass 1  
39 - Synth Bass 2  

Strings  
40 - Violin  
41 - Viola  
42 - Cello  
43 - Contrabass  
44 - Tremolo Strings  
45 - Pizzicato Strings  
46 - Orchestral Harp  
47 - Timpani  

Ensemble  
48 - String Ensemble 1  
49 - String Ensemble 2  
50 - Synth Strings 1  
51 - Synth Strings 2  
52 - Choir Aahs  
53 - Voice Oohs  
54 - Synth Voice  
55 - Orchestra Hit  

Brass  
56 - Trumpet  
57 - Trombone  
58 - Tuba  
59 - Muted Trumpet  
60 - French Horn  
61 - Brass Section  
62 - Synth Brass 1  
63 - Synth Brass 2  

Reed  
64 - Soprano Sax  
65 - Alto Sax  
66 - Tenor Sax  
67 - Baritone Sax  
68 - Oboe  
69 - English Horn  
70 - Bassoon  
71 - Clarinet  

Pipe  
72 - Piccolo  
73 - Flute  
74 - Recorder
75 - Pan Flute
76 - Bottle Blow
77 - Shakuhachi
78 - Whistle
79 - Ocarina

Synth Lead
80 - Lead 1 (square)
81 - Lead 2 (sawtooth)
82 - Lead 3 (căliope lead)
83 - Lead 4 (chiff lead)
84 - Lead 5 (charang)
85 - Lead 6 (voice)
86 - Lead 7 (fifths)
87 - Lead 8 (brass + lead)

Synth Pad
88 - Pad 1 (new age)
89 - Pad 2 (warm)
90 - Pad 3 (polysynth)
91 - Pad 4 (choir)
92 - Pad 5 (bowed)
93 - Pad 6 (metallic)
94 - Pad 7 (halo)
95 - Pad 8 (sweep)

Synth Effects
96 - FX 1 (rain)
97 - FX 2 (soundtrack)
98 - FX 3 (crystal)
99 - FX 4 (atmosphere)
100 - FX 5 (brightness)
101 - FX 6 (goblins)
102 - FX 7 (echoes)
103 - FX 8 (sci-fi)

Ethnic
104 - Sitar
105 - Banjo
106 - Shamisen
107 - Koto
108 - Kalimba
109 - Bagpipe
110 - Fiddle
111 - Shanai

Percussive
112 - Tinkle Bell
113 - Agogo
114 - Steel Drums
115 - Woodblock
<table>
<thead>
<tr>
<th></th>
<th>Sound Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>116</td>
<td>Taiko Drum</td>
</tr>
<tr>
<td>117</td>
<td>Melodic Tom</td>
</tr>
<tr>
<td>118</td>
<td>Synth Drum</td>
</tr>
<tr>
<td>119</td>
<td>Reverse Cymbal</td>
</tr>
<tr>
<td>120</td>
<td>Guitar Fret Noise</td>
</tr>
<tr>
<td>121</td>
<td>Breath Noise</td>
</tr>
<tr>
<td>122</td>
<td>Seashore</td>
</tr>
<tr>
<td>123</td>
<td>Bird Tweet</td>
</tr>
<tr>
<td>124</td>
<td>Telephone Ring</td>
</tr>
<tr>
<td>125</td>
<td>Helicopter</td>
</tr>
<tr>
<td>126</td>
<td>Applause</td>
</tr>
<tr>
<td>127</td>
<td>Gunshot</td>
</tr>
</tbody>
</table>
SOUND WAVE COMMANDS

PLAYWAVE
# $ K * PLAYWAVE

PLAYWAVE wavefile flags

The wavefile must be a filename of a .WAV file. The flags describe how the sound should be played. This capability also exists through the MCI command but with finer control. However this command does not require a true sound card to enjoy. There is a publicly available driver from Microsoft called SPEAKER(.EXE, .DRV, .ZIP ?) which will emulate the wave file capabilities over the PC speaker.

wavefile(LIST): The name of .WAV file.

flags(INTEGER): A flag to indicate how you want the sound played.
0 = Synchronous does not return until completed.
1 = Asynchronous returns immediately while sound is still playing.
2 = Don't use the default sound if the specified one cannot be found.
4 = In memory sound (not supported).
8 = Continue to loop the sound until another sound command is issued.
16 = Don't stop an already playing sound.

Note: these flags can be combined by adding them together.

Example:

playwave "c:\windows\tada.wav 1+8
playwave [] 0
# $K$ - SOUND COMMANDS

SOUNDON
SOUNDOFF
SOUND
**SOUNDON**

**SOUNDON frequency**  (library procedure)

The input is the frequency of the sound you want to hear. The larger the frequency the high the pitch. The sound will only come out the PC speaker and will work on all Windows systems. The sound will remain on until you issue a **SOUNDOFF** command.

Warning: This command will not work on NT because it uses OUTPORT.

Example:

```plaintext
soundon 1000
soundoff
```
SOUNDOFF

This command will shutoff a sound turned on by a SOUNDON command.

Warning: This command will not work on NT because it uses OUTPORT.

Example:

soundon 1000
soundoff

# SOUNDOFF
$ SOUNDOFF
$ SOUNDOFF
* SOUNDOFF
* LOGO:0
SOUND soundstream

The input must be a list of pairs. Each pair specifies a frequency (in hertz) and duration (clock ticks). MSWLogo cannot yield to other applications while a sound vector is being played. The larger the frequency the high the pitch. The large the duration the longer the sound. The sound will only come out the PC speaker.

soundstream:(LIST) A list of frequency duration pairs (integers) [frequency duration frequency duration ...].

Example:

sound [1000 200]

Example:

make "zing []
repeat 50 [make "zing lput repcount*10 :zing make "zing lput 100 :zing]
sound :zing

* SOUND
* SOUND
* SOUND
* LOGO:0
# MULTI MEDIA COMMANDS

**MCI**

---

* MULTI_MEDIA_COMMANDS
* MULTI MEDIA COMMANDS
* MULTI MEDIA COMMANDS
* LOGO:0
status MCI mci-command-list

status (MCI mci-command-list callback)

The mci-command-list must be a list. The MCI command may or may not output a list depending on the MCI command you issue. The MCI interface is very powerful. It opens the door to letting Logo control any Windows Multi Media device. These include Sound cards, CD-ROM players, VIDEO movies and more. In the second form if the mci-command-list uses the "notify" option then your callback code will be executed when the command is completed (as opposed to having to "wait"). The callback mechanism allows you, for example, to play a wavefile and be notified when it's completed so that you can start another immediately without sitting and "waiting" for it.

The MCI command is designed to let you, as a Logo programmer, write procedures to manipulate Multi Media devices. You can now link sounds to the steps of drawing a picture. You can narrate your own slide show. You can even ask your user questions in your own voice.

status:(LIST) sometimes an output from MCI and sometimes nothing depending on the command.
mci-command-list:(LIST) This is described in a separate help file Help MCI Command.
callback:(LIST) is a (short) list of logo commands (or a procedure name) to execute when the MCI command completes if you use the "notify" option instead of "wait".

The MCI interface allows you to start a device and optionally "wait" for it to finish or "notify" you when it has finished the request.

Example:

to soundit
print mci [open c:\\windows\\tada.wav type waveaudio alias wal]
print mci [open c:\\windows\\ding.wav type waveaudio alias wa2]
mci [seek wal to start]
mci [play wa1 wait]
repeat 2-
  [~
mci [seek wa2 to start]-
mci [play wa2 wait]-
]mci [close wa1]
mci [close wa2]end

Note: That the Microsoft speaker sound card emulator does NOT work with MCI.
**ABBREVIATIONS**

This section is primarily here for On-line HELP.

**ABBREVIATION LIST**

* ABBREVIATIONS
* ABBREVIATIONS
* ABBREVIATIONS
* LOGO:0
$\$K\$ ABBREVIATION LIST

BK
BS
BE
BL
CS
CT
CO
DOWN
ED
ER
ERF
FD
FS
HT
IFF
IFT
LR
LT
OP
PD
PE
PPT
PX
PU
PR
RC
RCS
RL
RW
RR
RT
SE
SetFC
SETH
SETPC
SETSC
ST
SS
TS
UP

$\$ ABBREVIATION LIST
$\$ ABBREVIATION LIST
$\$ ABBREVIATION LIST
$\$ LOGO:0
This is an abbreviation for \textsc{sum}. 

*LOGO:0
This is an abbreviation for BACK.
This is an abbreviation for \textsc{butfirsts}. 
This is an abbreviation for BUTFIRST.
This is an abbreviation for BUTLAST.

# § K * BL

BL

LOGO:0
This is an abbreviation for CLEARSCREEN.
CT

This is an abbreviation for CLEARTEXT.
This is an abbreviation for \textit{CONTINUE}
This is an abbreviation for **DOWNPITCH**.
This is an abbreviation for EDIT.
This is an abbreviation for ERASE.
This is an abbreviation for ERASEFILE.
This is an abbreviation for FORWRD.
This is an abbreviation for **FULLSCREEN**
This is an abbreviation for HIDE\textsc{TURTLE}.
This is an abbreviation for IFFALSE.
This is an abbreviation for \texttt{IFTRUE}.
LR

This is an abbreviation for LEFTROLL.
This is an abbreviation for **LEFT**.
This is an abbreviation for OUTPUT.
This is an abbreviation for **PENDOWN**
This is an abbreviation for PENERASE.
This is an abbreviation for **PENPAINT**.

* PPT
$ PPT
^K PPT
* LOGO:0
PX

This is an abbreviation for PENREVERSE
This is an abbreviation for PENUP
This is an abbreviation for PRINT.
This is an abbreviation for READCHAR.
RCS

This is an abbreviation for READCHARS.

^ RCS
^ RCS
^ RCS
* LOGO:0
This is an abbreviation for READLIST.
This is an abbreviation for **RIGHTROLL**.
This is an abbreviation for **RIGHT**.
This is an abbreviation for READWORD.
This is an abbreviation for SENTENCE.
SETFC

This is an abbreviation for SETFLOODCOLOR.
This is an abbreviation for SETHEADING.
SETPC

This is an abbreviation for SETPENCOLOR.
SETSC

This is an abbreviation for SETSCREENCOLOR
This is an abbreviation for SHOWTURTLE
This is an abbreviation for SPLITSCREEN.
This is an abbreviation for **TEXTSCREEN**
This is an abbreviation for **UPPITCH**.
This is an abbreviation for \textbf{DIFFERENCE}. 

\texttt{# $K$ \_} 

\texttt{\_8} 

\texttt{\_} 

\texttt{\_} 

\texttt{* LOGO:0}
This is an abbreviation for QUOTIENT.
This is an abbreviation for \textit{EQUALP}. 

\begin{verbatim}
# $K$ =

This is an abbreviation for \textit{EQUALP}.

\end{verbatim}
This is an abbreviation for **GREATERP**

```
# _5
$ >
^ >
* LOGO:0
```
This is an abbreviation for LESSP.
END OF DOCUMENT