EAGLE

***E**ASILY **A**PPLICABLE **G**RAPHICAL **L**AYOUT **E**DITOR*

EAGLE Help

extracted from the EAGLE Help Function

of

Version 6.4.0

Table Of Contents -- EAGLE Help 6.4.0

General Help

While inside a board, schematic, or library editor window, pressing F1 or entering the command `HELP` will open the help page for the currently active command.

You can also display an editor command's help page by entering

```
HELP command
```

replacing "command" with, e.g., `MOVE`, which would display the help page for the MOVE command.

Anywhere else, pressing the F1 key will bring up a context sensitive help page for the menu, dialog or action that is currently active.

For detailed information on how to get started with EAGLE please read the following help pages:

- Quick Introduction
- Configuring EAGLE
- Command Line Options
- Control Panel

Configuring EAGLE

Global EAGLE parameters can be adjusted in the Control Panel.

The following editor commands can be used to customize the way EAGLE works. They can be given either directly from an editor window's command line, or in the eagle.scr file.

User Interface

Command menu	MENU command..;
Assign keys	ASSIGN function_key command..;

Snap function	SET SNAP_LENGTH number;
	SET CATCH_FACTOR value;
	SET SELECT_FACTOR value;

Content of menus	SET USED_LAYERS name | number;

	SET WIDTH_MENU value..;
	SET DIAMETER_MENU value..;
	SET DRILL_MENU value..;
	SET SMD_MENU value..;
	SET SIZE_MENU value..;
	SET ISOLATE_MENU value..;
	SET SPACING_MENU value..;
	SET MITER_MENU value..;
Wire bend	SET WIRE_BEND bend_nr;
Beep on/off	SET BEEP OFF | ON;

Screen Display

Color for grid lines	SET COLOR_GRID color;
Color for layer	SET COLOR_LAYER layer color;
Fill style for layer	SET FILL_LAYER layer fill;
Grid parameter	SET MIN_GRID_SIZE pixels;
Min. text size displayed	SET MIN_TEXT_SIZE size;
Display of net lines	SET NET_WIRE_WIDTH width;
Display of pads	SET DISPLAY_MODE REAL | NODRILL;
	SET PAD_NAMES OFF | ON;
Display of bus lines	SET BUS_WIRE_WIDTH width;
DRC fill style	SET DRC_FILL fill_name;
Polygon processing	SET POLYGON_RATSNEST OFF | ON;
Vector font	SET VECTOR_FONT OFF | ON;

Mode Parameters

Package check	SET CHECK_CONNECTS OFF | ON;
Grid parameters	GRID options;
Replace mode	SET REPLACE_SAME NAMES | COORDS;
UNDO Buffer	SET UNDO_LOG OFF | ON;
Wire Optimizing	SET OPTIMIZING OFF | ON;
Net wire termination	SET AUTO_END_NET OFF | ON;
Automatic junctions	SET AUTO_JUNCTION OFF | ON;

Presettings

Pad shape	CHANGE SHAPE shape;
Wire width	CHANGE WIDTH value;
Pad/via diameter	CHANGE DIAMETER diameter;
Pad/via/hole drill diam.	CHANGE DRILL value;
Smd size	CHANGE SMD width height;
Text height	CHANGE SIZE value;
Text thickness	CHANGE RATIO ratio;
Text font	CHANGE FONT font;
Text alignment	CHANGE ALIGN align;
Polygon parameter	CHANGE THERMALS OFF | ON;
Polygon parameter	CHANGE ORPHANS OFF | ON;
Polygon parameter	CHANGE ISOLATE distance;
Polygon parameter	CHANGE POUR SOLID | HATCH |

	CUTOUT;
Polygon parameter	CHANGE RANK value;
Polygon parameter	CHANGE SPACING distance;
Dimension type	CHANGE DTYPE value;

Command Line Options

You can call up EAGLE with command line parameters. Use the following format:

```
eagle [ options [ filename [ layer ] ] ]
```

Options

`-Axxx`	Assembly variant
`-Cxxx`	execute the given Command
`-Dxxx`	Draw tolerance (0.1 = 10%)
`-Exxx`	Drill tolerance (0.1 = 10%)
`-Fxxx`	Flash tolerance (0.1 = 10%)
`-N-`	no command line prompts
`-O+`	Optimize pen movement
`-Pxxx`	plotter Pen (layer=pen)
`-Rxxx`	drill Rack file
`-Sxxx`	Scriptfile
`-Uxxx`	User settings file
`-Wxxx`	aperture Wheel file
`-X-`	eXecute CAM Processor
`-c+`	positive Coordinates
`-dxxx`	Device (-d? for list)
`-e-`	Emulate apertures
`-f+`	Fill pads
`-hxxx`	page Height (inch)
`-m-`	Mirror output
`-oxxx`	Output filename
`-pxxx`	Pen diameter (mm)
`-q-`	Quick plot
`-r-`	Rotate output 90 degrees
`-sxxx`	Scale factor
`-u-`	output Upside down
`-vxxx`	pen Velocity
`-wxxx`	page Width (inch)
`-xxxx`	offset X (inch)
`-yxxx`	offset Y (inch)

where `xxx` means that further data, e.g. a file name or a decimal number needs to be appended to the option character (without space or separated by a space), as in

```
-Wmywheel.whl
-W mywheel.whl
-e      Aperture emulation on
-e+     dto.
-e-     Aperture emulation off
```

For flag options, a `'-'` means that the option is off by default, while `'+'` means it is on by default.

Flag options (e.g. `-e`) can be used without repeating the `'-'` character:

`-ecfm`	Aperture emulation on, positive oordinates on, fill pads on
`-ec-f+`	Aperture emulation on, positive oordinates **off**, fill pads **on**

User settings

User settings are stored in the `eaglerc` file, which, by default, is stored in `$HOME/.eaglerc`.

On **Windows** it is stored in the file `eaglerc.usr` under the directory that is defined by the registry key "HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders\AppData" if no environment variable named HOME is defined.

With the command line option `-U` another file can be specified for this. This makes particular sense when you want to use several EAGLE versions with different settings. Example:

```
            C:/Program Files/MyEAGLE5/bin/eagle.exe
                   -UC:/Settings/eaglerc5.usr
```

may start version 5 with a V5 specific,

```
            C:/Program Files/MyEAGLE6/bin/eagle.exe
                   -UC:/Settings/eaglerc6.usr
```

may start version 6 with a V6 specific `eaglerc` file, which may differ from eaglerc5.usr in the project directories for instance.

If a `'-'` sign is given as the file name, as in `-U-`, no `eaglerc` file will be read or written.

Defining Tolerance Values

Without `'+'` or `'-'` sign, a tolerance value applies to both directions:

`-D0.10`	adjusts the draw tolerance to ±10%
`-D+0.1 -D-0.05`	adjusts the draw toleranceto +10% and -5%

Executing commands

If a command is given with the `'-C'` option, as in

```
eagle -C "window (1 1) (2 2);" myboard.brd
```

EAGLE will load the given file and execute the command as if it had been typed into the editor window's command line.

The following conditions apply for the `'-C'` option:

- A file name (board, schematic or library) must be given, so that an editor window will be opened in which the command can be executed. That file doesn't necessarily need to exist. The command is executed after loading and confirming of related messages.
- The `eagle.scr` file will not be executed automatically.
- The option `'-s'` will be ignored.
- The user settings will not be written back to the `eaglerc` file.

- Any project that has been open when EAGLE was left the last time will not be opened.
- The command can be a single command, or a sequence of commands delimited by semicolons.

To run EAGLE without automatically executing the `eagle.scr` file or loading a project, the command string can be empty, as in

```
eagle -C ""
```

Note that in this special case there must be a blank between the option character and the quotes, so that the program will see the explicitly empty string. There also doesn't have to be a file name here, because no command will actually be executed.

Filename

If the given filename is `eagle.epf` (optionally preceded by a directory name), EAGLE will load that Project File. Otherwise, if no file extension is given, it defaults to `.brd`, to load a board file.

Quick Introduction

For a quick start you should know more about the following topics:

- Control Panel and Editor Windows
- Using Editor Commands
- Entering Parameters and Values
- Drawing a Schematic
- Checking the Schematic
- Generating a Board from a Schematic
- Checking the Layout
- Creating a Library Device
- Using the Autorouter
- Using the System Printer
- Using the CAM Processor

In case of problems please contact our free Technical Support.

Control Panel and Editor Windows

From the Control Panel you can open schematic, board, or library editor windows by using the File menu or double clicking an icon.

Entering Parameters and Values

Parameters and values can be entered in the EAGLE command line or, more conveniently, in the Parameter Toolbars which appear when a command is activated. As this is quite self-explanatory, the help text does not explicitly mention this option at other locations.

Wherever coordinates or sizes (like width, diameter etc.) can be entered, they may be given with units, as in 50mil or 0.8mm. If no unit is given, the current grid unit is used.

Drawing a Schematic

Create a Schematic File

Use File/New and Save as to create a schematic with a name of your choice.

Load a Drawing Frame

Load library FRAMES with USE and place a frame of your choice with ADD.

Place Symbols

Load appropriate libraries with USE and place symbols (see ADD, MOVE, DELETE, ROTATE, NAME, VALUE). Where a particular component is not available, define a new one with the library editor.

Draw Bus Connections

Using the BUS command, draw bus connections. You can NAME a bus in such a way that you can drag nets out of the bus which are named accordingly.

Draw Net Connections

Using the NET command, connect up the pins of the various elements on the drawing. Intersecting nets may be made into connections with the JUNCTION command.

Checking the Schematic

Carry out an electrical rule check (ERC) to look for open pins, etc., and use the messages generated to correct any errors. Use the SHOW command to follow complete nets across the screen. Use the EXPORT command to generate a netlist, pinlist, or partlist if necessary.

Generating a Board from a Schematic

By using the BOARD command or clicking the Switch-to-Board icon you can generate a board from the loaded schematic (if there is no board with the same name yet).

All the components, together with their connections drawn as airwires, appear beside a blank board ready for placing. Power pins are automatically connected to the appropriate supply (if not connected by a net on the schematic).

The board is linked to the schematic via Forward&Back Annotation. This mechanism makes sure that schematic and board are consistent. When editing a drawing, board and schematic must be loaded to keep Forward&Back Annotation active.

Set Board Outlines and Place Components

The board outlines can be adjusted with the MOVE and SPLIT commands as appropriate before moving each package on the board. Once all packages have been placed, the RATSNEST command is used to optimize airwires.

Define Restricted Areas

If required, restricted areas for the Autorouter can be defined as RECTangles, POLYGONs, or CIRCLEs on the tRestrict, bRestrict, or vRestrict layers. Note: areas enclosed by wires drawn on the Dimension layer are borders for the Autorouter, too.

Routing

Airwires are now converted into tracks with the aid of the ROUTE command. This function can also be performed automatically by the Autorouter, when available.

Checking the Layout

Check the layout (DRC) and correct the errors (ERRORS). Generate net, part, or pin list if necessary(EXPORT).

Creating a Library Device

Creating a new component part in a library has three steps. You must follow these steps as they build upon each other.

To start, open a library. Use the File menu Open or New command (not the USE command).

Create a Package

Packages are the part of the device that are added to a board.

Click the Edit Package icon and edit a new package by typing its name in the New field of the dialog box.

Set the proper distance GRID.

NAME and place PADs properly.

Add texts >NAME and >VALUE with the TEXT command (show actual name and value in the board) and draw package outlines (WIRE command) in the proper layers.

Create a Symbol

Symbols are the part of the device that are added to a schematic.

Click the Edit Symbol icon and edit a new symbol by typing its name in the New field of the dialog box.

Place and name pins with the commands PIN and NAME and provide pin parameters (CHANGE).

Add texts >NAME and >VALUE with the TEXT command (show actual name and value in the schematic) and draw symbol outlines (WIRE command) in the proper layers.

Create the Device

Devices are the "master" part of a component and use both a package and one or more symbols.

Click the Edit Device icon and edit a new device by typing its name in the New field of the dialog box.

Assign the package with the PACKAGE command.

Add the gate(s) with ADD, you can have as many gates as needed.

Use CONNECT to specify which of the packages pads are connected to the pins of each gate.

Save the library and you can USE it from the schematic or board editor.

Control Panel

The Control Panel is the top level window of EAGLE. It contains a tree view on the left side, and an information window on the right side.

Directories

The top level items of the tree view represent the various types of EAGLE files. Each of these can point to one or more directories that contain files of that type. The location of these directories can be defined with the directories dialog. If a top level item points to a single directory, the contents of that directory will appear if the item is opened (either by clicking on the little symbol to the left, or by double clicking the item). If such an item points to more directories, all of these directories will be listed when the item is opened.

Context menu

The context menu of the tree items can be accessed by clicking on them with the right mouse button. It contains options specific to the selected item.

Descriptions

The *Description* column of the tree view contains a short description of the item (if available). These descriptions are derived from the first non-blank line of the text from the following sources:

Directories	a file named DESCRIPTION in that directory
Libraries	the description of the library
Devices	the description of the device
Packages	the description of the package
Design Rules	the description of the design rules file
User Language Programs	the text defined with the `#usage` directive
Scripts	the comment at the beginning of the script

	file
CAM Jobs	the description of the CAM job

Drag&drop

You can use *Drag&Drop* to copy or move files and directories within the tree view. It is also possible to drag a device or package to a schematic, board or library window, respectively, and drop it there to add it to the drawing. User Language Programs and Scripts will be executed if dropped onto an editor window, and Design Rules will be applied to a board if dropped onto a board editor window. If a board, schematic or library file is dropped onto its respective editor window, it will be loaded into the editor. All of these functions can also be accessed through the *context menu* of the particular tree item.

Information window

The right hand side of the Control Panel displays information about the current item in the tree view. That information is derived from the places listed above under *Descriptions*. Devices and packages also show a preview of their contents.

Pulldown menu

The Control panel's *pulldown menu* contains the following options:

File

New	create a new file
Open	open an existing file
Open recent projects	open a recently used project
Save all	save all modified editor files
Close project	close the current project
Exit	exit from the program

View

Refresh	refresh the contents of the tree view
Search in tree	search in the contents of the tree view (see below)
Sort	change the sorting of the tree view

Options

Directories...	opens the directories dialog
Backup...	opens the backup dialog
User interface...	opens the user interface dialog
Window positions...	opens the window positions dialog

Window

Control Panel	switch to the Control Panel
1 Schematic - ...	switch to window number

2 Board - ...	switch to window number 1 switch to window number 2

Help

General	opens a general help page
Context	opens the help page for the current context
Control Panel	opens the help page you are currently looking at
EAGLE License	opens the license dialog
Check for Update	checks if a new version of EAGLE is available
About EAGLE	displays details on your EAGLE version and license

Search Bar

The *Search* pattern can be one or more words, separated by blanks. These words are searched case insensitively in the tree names and descriptions and must all match. The wildcard character '*' matches any number of non-whitespace characters, while '?' matches exactly one of these characters. To search for a wildcard character itself it has to be escaped like '*'. To restrict the search to a branch of the tree, the search can be started in its context menu. To find a NAND device from the 74xx series, e.g. enter: `74* nand`

Status line

The status line at the bottom of the Control Panel contains the full name of the currently selected item.

Context Menus

Clicking on an item in the Control Panel with the right mouse button opens a context menu which allows the following actions (not all of them may be present on a particular item):

New Folder

Creates a new folder below the selected folder and puts the newly created tree item into *Rename* mode.

Edit Description

Loads the DESCRIPTION file of a directory into the HTML editor.

Rename

Puts the tree item's text into edit mode, so that it can be renamed.

Copy

Opens a file dialog in which you can enter a name to which to copy this file or directory. You can also use *Drag&Drop* to do this.

Delete

Deletes the file or directory (you will be prompted to confirm that you really want this to happen).

Use

Marks this library to be *used* when searching for devices or packages. You can also click on the icon in the second column of the tree view to toggle this flag.

Use all

Marks all libraries in the Libraries path to be *used* when searching for devices or packages.

Use none

Removes the *use* marks from all libraries (including such libraries that are not in the Libraries path).

Update

Updates all parts used from this library in the loaded board and schematic.

Update in Library

Updates all packages used from this library in the loaded library.

Add to Schematic

Starts the ADD command in the schematic window with this device. You can also use *Drag&Drop* to do this.

Add to Board

Starts the ADD command in the board window with this package. You can also use *Drag&Drop* to do this.

Copy to Library

Copies the selected device set or package into the loaded library. You can also use *Drag&Drop* to do this.

New package variant in Library

Creates a new package variant with the selected package in the current device set of the loaded library. You can also use *Drag&Drop* to do this.

Open/Close Project

Opens or closes this project. You can also click on the icon in the second column of the tree view to do this.

New

Opens a window with a new file of the given type.

Open

Opens this file in the propper window. You can also use *Drag&Drop* to do this.

Print...

Prints the file to the system printer. See the chapter on printing to the system printer for more information on how to use the print dialogs.

Printing a file through this context menu option will always print the file as it is on disk, even if you have an open editor window in which you have modified the file! Use the PRINT command to print the drawing from an open editor window.
Please note that polygons in boards will not be automatically calculated when printing via the context menu! Only the outlines will be drawn. To print polygons in their calculated shape you have to load the drawing into an editor window, enter RATSNEST and then PRINT.

Run in ...

Runs this User Language Program in the current schematic, board or library. You can also use *Drag&Drop* to do this.

Execute in ...

Executes this script file in the current schematic, board or library. You can also use *Drag&Drop* to do this.

Load into Board

Loads this set of Design Rules into the current board. You can also use *Drag&Drop* to do this.

Directories

The *Directories* dialog is used to define the directory paths in which to search for files.

All entries may contain one or more directories, separated by a colon (`':'`), in which to look for the various types of files.

On **Windows** the individual directory names are separated by a semicolon (`';'`).

When entering an OPEN, USE, SCRIPT or RUN command, these paths will be searched left-to-right to locate the file. If the file dialog is used to access a file of one of these types, the directory into which the user has navigated through the file dialog will be implicitly added to the end of the respective search path.

The special variables `$HOME` and `$EAGLEDIR` can be used to reference the user's home directory and the EAGLE program directory, respectively.

On **Windows** the value of `$HOME` is either that of the environment variable HOME (if set), or the value of the registry key "HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders\Personal", which contains the actual name of the "My Documents" directory.

Backup

The *Backup* dialog allows you to customize the automatic backup function.

Maximum backup level

Defines how many backup copies of your EAGLE data files shall be kept when regularly saving a file to disk with the WRITE command (default is 9).

Auto backup interval (minutes)

Defines the maximum time after which EAGLE automatically creates a safety backup copy of any modified drawing (default is 5).

Automatically save project file

If this option is checked, your project settings will be automatically saved when you exit from the program. Note that if you uncheck this option while you have a project open, this project will not be saved when you close it, and thus this setting will not be stored in the project's eagle.epf file. This means that the next time you open the project, this option will be checked again. If you want this option to remain unchecked for the current project, you need to manually select "File/Save all" from the pulldown menu after unchecking this option.

User Interface

The *User interface* dialog allows you to customize the appearance of the layout, schematic and library editor windows.

Controls

Pulldown menu	activates the pulldown menu at the top of the editor window
Action toolbar	activates the action toolbar containing buttons for "File", "Print" etc.
Parameter toolbar	activates the dynamic parameter toolbar, which contains all the parameters that are available for the currently active command
Command buttons	activates the command buttons
Command texts	activates the textual command menu
Sheet	aktivates the sheet thumbnail preview

thumbnails

Layout

Background	selects a black, white or colored background for the layout mode
Cursor	selects a small or large cursor for the layout mode

Schematic

Background	selects a black, white or colored background for the schematic mode
Cursor	selects a small or large cursor for the schematic mode

Vertical text

New drawings	selects the reading direction of vertical texts in newly created drawings
This drawing	selects the reading direction of vertical texts in the currently loaded drawing

Help

Bubble help	activates the "Bubble Help" function, which pops up a short hint about the meaning of several buttons when moving the cursor over them
User guidance	activates the "User Guidance" function, which displays a helping text telling the user what would be the next meaningful action when a command is active

Misc

Always vector font	always displays texts in drawings with the builtin vector font, regardless of which font is actually set for a particular text
Mouse wheel zoom	defines the zoom factor that will be used to zoom in and out of an editor window when the mouse wheel is turned ('0' disables this feature, the sign of this value defines the direction of the zoom operation)

Window positions

The *Window positions* dialog allows you to store the positions of all currently open windows, so that later, when a window of the same type is opened again, it will appear at the same position as before.

You can also delete all stored window positions, so that the window manager can decide again where to place newly opened windows.

Check for Update

The option "Help/Check for Update" in the Control Panel's pulldown menu opens a dialog that displays whether there is a new version of EAGLE available on the CadSoft server.

The **Configure** button opens a dialog in which you can specify if and how often a check for new versions should be done automatically upon program start (by default it checks once per day). If you need to use a proxy to access the Internet, this can also be specified in the configuration dialog. In the "Host" field enter the full name of the proxy host, without any `http://` prefix, and enter an optional port number in the "Port" field.

If you would like to be informed about beta versions of EAGLE, you can check the "Also check for beta versions" box.

Keyboard and Mouse

The *modifier keys* (`Alt`, `Ctrl` and `Shift`) are used to modify the behavior of certain mouse actions. Note that depending on which operating system or window manager you use, some of these keys (in combination with mouse events) may not be delivered to applications, which means that some of the functions described here may not be available.

Alt

Pressing the `Alt` key switches to an alternate GRID. This can typically be a finer grid than the normal one, which allows you to quickly do some fine positioning in a dense area, for instance, where the normal grid might be too coarse. The alternate grid remains active as long as the `Alt` key is held pressed down.

Ctrl

Pressing the `Ctrl` key while clicking on the right mouse button toggles between corresponding wire bend styles (only applies to commands that support wire bend styles, like, for instance, WIRE).

The `Ctrl` key together with the left mouse button controls special functionality of individual commands, like, for instance, selecting an object at its origin with the MOVE command.

If a command can select a group, the `Ctrl` key must be pressed together with the right mouse button when selecting the group (otherwise a context menu for the selected object would be opened).

On **Mac OS X** the `Cmd` key has to be used instead of the `Ctrl` key.

Shift

Pressing the `Shift` key while clicking on the right mouse button reverses the direction in which the wire bend styles are switched through (only applies to commands that support wire bend styles, like, for instance, WIRE).

The `Shift` key together with the left mouse button controls special functionality of individual commands, like, for instance, deleting a higher level object with the DELETE command.

Esc

Pressing the `Esc` key when a command is active will cancel the current activity of that command without canceling the entire command (if there is text in the command line, that text will be deleted first, and the next press of the `Esc` key will act on the command). For the MOVE command, for example, this means that an object that is currently attached to the cursor will be dropped and an other object can be selected.

Crsr-Up/Down

The keys `Crsr-Up` (cursor up) and `Crsr-Down` (cursor down) can be used in the command line of an editor window to scroll through the command history.

Function Keys

Function keys can be assigned any commands by using the ASSIGN command.

Left Mouse Button

The left mouse button is generally used to select, draw or place objects.

Center Mouse Button

The center mouse button changes the current layer or mirrors the object currently attached to the mouse cursor.

The following commands support the center mouse button:

ADD	mirror part
ARC	change active layer
CIRCLE	change active layer
COPY	mirror object
INVOKE	mirror gate
LABEL	change active layer
MOVE	mirror object or group
PASTE	mirror group
POLYGON	change active layer
RECT	change active layer
ROUTE	change active layer
SMD	change active layer
TEXT	change active layer
WIRE	change active layer

Click&Drag with the center mouse button will pan the drawing within the editor window.

Right Mouse Button

The right mouse button is mostly used to select a group, rotate objects attached to the mouse cursor, change wire bend styles and several other command specific functions.

When selecting an object with the right mouse button, a context specific popup menu is displayed from which commands that apply to this object can be selected. If there is currently a command active that can be applied to a group, the popup menu will contain an entry for this.

The following commands support the right mouse button:

ADD	rotate part
ARC	change direction of arc
BUS	change wire bend
CHANGE	apply change to group
DELETE	delete group
GROUP	close polygon
INVOKE	rotate gate

LABEL	rotate label
MIRROR	mirror group
MOVE	rotate object, select group
NET	change wire bend
PAD	rotate pad
PASTE	rotate group
PIN	rotate pin
POLYGON	change wire bend
RIPUP	ripup group
ROTATE	rotate group
ROUTE	change wire bend
SMD	rotate smd
SPLIT	change wire bend
TEXT	rotate text
WIRE	change wire bend

Mouse Wheel

Inside an editor window the mouse wheel can be used to zoom in and out.

Selecting objects in dense areas

When you try to select an object at a position where several objects are placed close together, a four way arrow and the question

Select highlighted object? (left=yes, right=next, ESC=cancel)

indicates that you can now choose one of these objects.

Press the right mouse button to switch to the next object.

Press the left mouse button to select the highlighted object.

Press Esc to cancel the selection procedure.

The command

```
SET Select_Factor select_radius;
```

defines the selection radius.

If the original selection was done with the right mouse button, a context specific popup menu will be displayed which applies to the first selected object, and which contains "Next" as the first entry. Clicking on this entry will cyclically switch through the objects within the selection radius.

Editor Windows

EAGLE knows different types of data files, each of which has its own type of editor window. By double clicking on one of the items in the Control Panel or by selecting a file from the **File/Open** menu, an editor window suitable for that file will be opened.

- Library Editor
- Schematic Editor
- Board Editor

- Text Editor

Library Editor

The *Library Editor* is used to edit a part library (`*.lbr`).

After opening a new library editor window, the edit area will be empty and you will have to use the EDIT command to select which package, symbol or device you want to edit or create.

Edit Library Object

In library edit mode you can edit packages, symbols, and devices.

Package: the package definition.

Symbol: the symbol as it appears in the circuit diagram.

Device: definition of the whole component. Contains one or more package variants and one or several symbols (e.g. gates). The symbols can be different from each other.

Click on the **Dev**, **Pac** or **Sym** button to select Device, Packages or Symbols, respectively.

If you want to create a new object, write the name of the new object into the **New** field. You can also edit an existing object by typing its name into this field. If you omit the extension, an object of the type indicated by the **Choose...** prompt will be loaded. Otherwise an object of the type indicated by the extension will be loaded.

If your license does not include the Schematic Module, the object type buttons (**Dev**...) will not appear in the menu.

Board Editor

The *Board Editor* is used to edit a board (`*.brd`).

When there is a schematic file (`*.sch`) with the same name as the board file (in the same directory), opening a board editor window will automatically open a Schematic Editor window containing that file and will put it on the desktop as an icon. This is necessary to have the schematic file loaded when editing the board causes modifications that have to be back-annotated to the schematic.

Schematic Editor

The *Schematic Editor* is used to edit a schematic (`*.sch`).

When there is a board file (`*.brd`) with the same name as the schematic file (in the same directory), opening a schematic editor window will automatically open a Board Editor window containing that file and will put it on the desktop as an icon. This is necessary to have the board file loaded when editing the schematic causes modifications that have to be forward-annotated to the board.

The combo box in the action toolbar of the schematic editor window allows you to switch between the various sheets of the schematic, or to add new sheets to the schematic (this can also be done using the EDIT command).

Text Editor

The *Text Editor* is used to edit any kind of text.

The text must be a pure ASCII file and must not contain any control codes. The main area of use for the text editor is writing User Language Programs and Script files.

Using an external text editor

If you prefer to use an external text editor instead of EAGLE's builtin text editor, you can specify the command necessary to start that editor in the "Options/User interface" dialog.

Within that command the following placeholders will be replaced with actual values:

`%C`	the column in which to place the cursor (currently always `1`)
`%F`	the name of the file to load
`%L`	the line in which to place the cursor

If the command consists only of a hyphen (`'-'`), EAGLE will never open a text editor window. This may be useful for people who always start their text editor by themselves.

The following restrictions apply when using an external text editor:

- The external text editor runs as a separate process, and EAGLE has no way of knowing whether the loaded file has been modified or not. It is up to you to save the file after you have made modifications.
- If the same file is loaded into the text editor several times, it depends on the configuration of the text editor in use whether it opens a new window each time, or whether it loads the file into the same window.
- The external text editor windows do not show up in EAGLE's window list, and are therefore not stored in the project file, and are not reopened when the project is opened again later.
- When leaving EAGLE, the external text editor processes will be terminated. It depends on the operating system and the particular text editor whether or not you are queried if a file has been modified and should be saved.
- The "File/Save all" function will not save files edited with an external text editor.
- The update report that may be given when loading a file from an older version of EAGLE is always displayed with the internal text editor.

Editor Commands

Change Mode/File Commands

CLOSE	Close drawing after editing
EDIT	Load/create a drawing
EXPORT	Generate ASCII list (e.g. netlist)
OPEN	Open library for editing
QUIT	Quit EAGLE
REMOVE	Delete files/library elements

SCRIPT	Execute command file
USE	Load library for placing elements
WRITE	Save drawing/library

Edit Drawings or Libraries

ADD	Add element to drawing/symbol to device
ARC	Draw arc
ATTRIBUTE	Define attributes
CIRCLE	Draw circle
CLASS	Define net classes
COPY	Copy objects/elements
CUT	Cut previously defined group
DELETE	Delete objects
DESCRIPTION	Change an object's description
GROUP	Define group for upcoming operation
HOLE	Define non-conducting hole
LAYER	Create/change layer
MIRROR	Mirror objects
MITER	Miter wire joints
MOVE	Move or rotate objects
NAME	Name object
PASTE	Paste previously cut group to a drawing
POLYGON	Draw polygon
RECT	Draw rectangle
ROTATE	Rotate objects
SMASH	Prepare NAME/VALUE text for moving
SPLIT	Bend wires/lines (tracks, nets, etc.)
TEXT	Add text to a drawing
VALUE	Enter/change value for component
WIRE	Draw line or routed track

Special Commands for Boards

DRC	Perform design rule check
ERRORS	Show DRC errors
LOCK	Lock component's position
RATSNEST	Show shortest air lines
REPLACE	Replace component
RIPUP	Ripup routed track
ROUTE	Route signal
SIGNAL	Define signal (air line)
VIA	Place via-hole

Special Commands for Schematics

BOARD	Create a board from a schematic
BUS	Draw bus line
ERC	Perform electrical rule check
GATESWAP	Swap equivalent 'gates'
INVOKE	Add certain 'gate' from a placed

	device
JUNCTION	Place connection point
LABEL	Provide label to bus or net
NET	Define net
PINSWAP	Swap equivalent pins

Special Commands for Libraries

CONNECT	Define pin/pad assignment
PACKAGE	Define package for device
PAD	Add pad to a package
PIN	Add pin to a symbol
PREFIX	Define default prefix for device
REMOVE	Delete library elements
RENAME	Rename symbol/package/device
SMD	Add smd pad to a package
TECHNOLOGY	Define technologies for a device
VALUE	Define if value text can be changed

Change Screen Display and User Interface

ASSIGN	Assign keys
CHANGE	Change parameters
DISPLAY	Display/hide layers
GRID	Define grid/unit
MENU	Configure command menu
SET	Set program parameters
WINDOW	Choose screen window

Miscellaneous Commands

AUTO	Start Autorouter
HELP	Show help page
INFO	Show information about object
MARK	Set/remove mark (for measuring)
OPTIMIZE	Optimize (join) wire segments
PRINT	Print to the system printer
REDO	Redo commands
RUN	Run User Language Program
SHOW	Highlight object
UNDO	Undo commands
UPDATE	Update library objects

Command Syntax

EAGLE commands can be entered in different ways:

- with the keyboard as text
- with the mouse by selecting menu items or clicking on icons
- with assigned keys (see ASSIGN command)

- with command files (see SCRIPT command)

All these methods can be mixed.

Commands and parameters in `CAPITAL LETTERS` are entered directly (or selected in the command menu with the mouse). For the input there is no difference between small and capital letters.

Parameters in `lowercase letters` are replaced by names, number values or key words. Example:

```
Syntax:  GRID grid_size grid_multiple;
Input:   GRID 1 10;
```

Shorten key words

For command names and other key words, only so many characters must be entered that they clearly differ from other key words.

Alternative Parameters

The sign | means that alternative parameters can be indicated. Example:

```
Syntax:  SET BEEP OFF | ON;
Input:   SET BEEP OFF;
         or
         SET BEEP ON;
```

Repetition Points

The signs .. mean that the function can be executed several times or that several parameters of the same type are allowed. Example:

```
Syntax:  DISPLAY option layer_name..
Input:   DISPLAY TOP PINS VIAS
```

Coordinates

The sign □ normally means that an object has to be selected with the left mouse button at this point in the command. Example:

```
Syntax:  MOVE □ □..
Input:   MOVE
         Mouse click on the first element to be moved
         Mouse click on the target position
         Mouse click on the second element to be moved
         etc.
```

This example also explains the meaning of the repetition points for commands with mouse clicks.

For the program each mouse click is the input of a coordinate. If coordinates are to be entered as text, the input via the keyboard must be as follows:

```
(x y)
```

x and y are numbers in the unit which has been selected with the GRID command. The input as text is mainly required for script files.

If a unit other than the one selected with the GRID command shall be used, it can be appended to the given coordinates, as in

```
(100mil 200mil)
```

Allowed units are `mm`, `mic`, `mil` and `in`. It is possible to use different units for x and y. The special coordinate

```
(@)
```

can be used to reference the current position of the mouse cursor within the draw window. For example, the input

```
MOVE R1 (@)
```

would move the part named R1 to the place currently pointed to with the mouse.

Any combination of the following modifiers may follow the opening brace in order to simulate a particular key that is held pressed with the "mouse click" or to change the type of coordinates:

`>`	right mouse button click
`A`	Alt key
`C`	Ctrl key
`P`	Polar coordinates (relative to the mark, x = radius, y = angle in degrees, counterclockwise)
`R`	Relative coordinates (relative to the mark)
`S`	Shift key

For example, the input

```
(CR> 1 2)
```

would result in a "right button mouse click" at (1 2) relative to the mark, with the Ctrl key held down (of course what happens with this kind of input will depend on the actual command). Note that if there is currently no mark defined, coordinates with `R` or `P` will be relative to the drawing's origin. Also, the modifier characters are not case sensitive, their sequence doesn't matter and there doesn't have to be a blank between them and the first coordinate digit. So the above example could also be written as `(r>cl 2)`. Values entered as "polar coordinates" will be stored internally as the corresponding pair of (x y) coordinates.

As an example for entering coordinates as text let's assume you wish to enter the exact dimensions for board outlines:

```
GRID 1 MM;
CHANGE LAYER DIMENSION;
WIRE 0 (0 0) (160 0) (160 100) (0 100) (0 0);
GRID LAST;
```

Decimal numbers

When entering decimal numbers in the command line of the editor window or in dialog input fields, you can use the comma as the decimal delimiter (as in `12,34`), if your locale settings allow this. However, when writing a script or a ULP that returns EAGLE commands through the `exit()` function, you should always use the 'dot' as the decimal delimiter (as in `12.34`), because otherwise your script or ULP might not work on other systems. In general, it is recommended to always use the 'dot' as the decimal delimiter.

Semicolon

The semicolon (';') terminates commands. A command needs to be terminated with a semicolon if there fewer than the maximum possible number of options. For example the command

```
WINDOW;
```

redraws the drawing window, whereas

```
WINDOW FIT
```

scales the drawing to fit entirely into the drawing window. There is no semicolon necessary here because it is already clear that the command is complete.

ADD

Function

Add elements into a drawing.
Add symbols into a device.

Syntax

```
ADD package_name[@library_name] [name] [orientation] ▯..
ADD device_name[@library_name] [name [gate]] [orientation] ▯..
ADD symbol_name [name] [options] ▯..
```

Mouse keys

Center mirrors the part.
Right rotates the part.
Shift+Right reverses the direction of rotating.

See also UPDATE, USE, INVOKE

The ADD command fetches a circuit symbol (gate) or a package from the active library and places it into the drawing.

During device definition the ADD command fetches a symbol into the device.

Usually you click the ADD command and select the package or symbol from the menu which opens. If necessary, parameters can now be entered via the keyboard.

If `device_name` contains wildcard characters (`'*'` or `'?'`) and more than one device matches the pattern, the ADD dialog will be opened and the specific device can be selected from the list. Note that the *Description* checkbox in the ADD dialog will be unchecked after any ADD command with a `device_name` has been given in the command line, no matter if it contains wildcards or not. This is because a `device_name` entered in the command line is only searched for in the device names, not in the descriptions.

The package or symbol is placed with the left button and rotated with the right button. After it has been placed another copy is immediately hanging from the cursor.

If there is already a device or package with the same name (from the same library) in the drawing, and the library has been modified after the original object was added, an automatic library update will be started and you will be asked whether objects in the drawing shall be replaced with their new versions. **Note: You should always run a Design Rule Check (DRC) and an Electrical Rule Check (ERC) after a library update has been performed!**

Fetching a Package or Symbol into a Drawing

Wildcards

The ADD command can be used with wildcards (`'*'` or `'?'`) to find a specific device. The ADD dialog offers a tree view of the matching devices, as well as a preview of the device and package variant.

To add directly from a specific library, the command syntax

```
ADD devicename@libraryname
```

can be used. `devicename` may contain wildcards and `libraryname` can be either a plain library name (like "ttl" or "ttl.lbr") or a full file name (like "/home/mydir/myproject/ttl.lbr" or "../lbr/ttl"). In case of blanks in the file name the whole expression has to be enclosed by apostrophs (like ADD 'DEV1A@/home/my dir/ttl.lbr').

Names

The package_name, device_name or symbol_name parameter is the name under which the package, device or symbol is stored in the library. It is usually selected from a menu. The name parameter is the name which the element is to receive in the drawing. If the name could be interpreted as an orientation or option, it must be enclosed in single quotes. If a name is not explicitly given it will receive an automatically generated name.

Example:

```
ADD DIL14 IC1 ▯
```

fetches the DIL14 package to the board and gives it the name IC1.

If no name is given in the schematic, the gate will receive the prefix that was specified in the device definition with PREFIX, expanded with a sequential number (e.g. IC1).

Example:

```
ADD 7400 ▯ ▯ ▯ ▯ ▯
```

This will place a sequence of five gates from 7400 type components. Assuming that the prefix is defined as "IC" and that the individual gates within a 7400 have the names A..D, the gates in the schematic will be named IC1A, IC1B, IC1C, IC1D, IC2A. (If elements with the same prefix have already been placed the counting will proceed from the next sequential number.) See also INVOKE.

While an object is attached to the cursor, you can change the name under which it will be added to the drawing. This allows you to add several parts of the same type, but with different, explicitly defined names:

Example:

```
ADD CAP C1 ▯ C5 ▯ C7 ▯
```

Particular Gates

To fetch a particular gate of a newly added device the name of that gate can be given following the part name:

Example:

```
ADD 7400 IC1 A ▯
```

This is mainly useful if a schematic is to be generated through a script. Note that if a particular gate is added, no other gates with add level MUST or ALWAYS will be fetched automatically, and you will have to use the INVOKE command to invoke at least the MUST gates (otherwise the Electrical Rule Check will report them as missing).

Orientation

This parameter defines the orientation of the object in the drawing. Objects are normally rotated using the right mouse button. In Script files textual descriptions of this parameter are used:

`[S][M]Rnnn`

`S`	sets the **S**pin flag, which disable keeping texts readable from the bottom or right side of the drawing (only available in a board context)
`M`	sets the **M**irror flag, which mirrors the object about the y-axis
`Rnnn`	sets the **R**otation to the given value, which may be in the range `0.0`...`359.9` (at a resolution of 0.1 degrees) in a board context, or one of `0`, `90`, `180` or `270` in a schematic context (angles may be given as negative values, which will be converted to the corresponding positive value)

The key letters **`S`**, **`M`** and **`R`** may be given in upper- or lowercase, and there must be at least **`R`** followed by a number.

If the **M**irror flag is set in an element as well as in a text within the element's package, they cancel each other out. The same applies to the **S**pin flag.

Examples:

R0	no rotation
R90	rotated 90° counterclockwise
R-90	rotated 90° clockwise (will be converted to 270°)
MR0	mirrored about the y-axis
SR0	spin texts
SMR33.3	rotated 33.3° counterclockwise, mirrored and spin texts

Default: R0

```
ADD DIL16 R90 (0 0);
```

places a 16-pin DIL package, rotated 90 degrees counterclockwise, at coordinates (0 0).

Error messages

An error message appears if a gate is to be fetched from a device which is not fully defined (see BOARD command). This can be prevented with the "SET CHECK_CONNECTS OFF;" command. Take care: The BOARD command will perform this check in any case. Switching it off is only sensible if no pcb is to be made.

Fetch Symbol into Device

During device definition the ADD command fetches a previously defined symbol into the device. Two parameters (swaplevel and addlevel) are possible, and these can be entered in any sequence. Both can be preset and changed with the CHANGE command. The value entered with the ADD command is also retained as a default value.

Swaplevel

The swaplevel is an integer number, to which the following rules apply:

0:	The symbol (gate) can not be swapped with any other in the schematic.
>0	The symbol (gate) can be swapped with any other symbol of the same type in the schematic that has the same swaplevel (including swapping between different devices).

Default: 0

Addlevel

The following possibilities are available for this parameter:

`Next`	If a device has more than one gate, the symbols are fetched into the schematic with Addlevel Next.
`Must`	If any symbol from a device is fetched into the schematic, then a symbol defined with Addlevel Must must also appear. This happens automatically. It cannot be deleted until all the other symbols in the device have been deleted. If the only symbols remaining from a device are Must-symbols, the DELETE command will delete the entire device.
`Always`	Like Must, although a symbol with Addlevel Always can be deleted and brought back into the schematic with INVOKE.
`Can`	If a device contains Next-gates, then Can-gates are only fetched if explicitly called with INVOKE. A symbol with Addlevel Can is only then fetched into the schematic with ADD if the device only contains Can-gates and Request-gates.
`Request`	This property is usefully applied to devices' power-symbols. Request-gates can only be explicitly fetched into the schematic (INVOKE) and are not internally counted. The effect of this is that in devices with only one gate and one voltage supply symbol, the gate name is not added to the component name. In the case of a 7400 with four gates (plus power supply) the individual gates in the schematic are called, for example, IC1A, IC1B, IC1C and IC1D. A 68000 with only one *Gate*, the processor symbol, might on the other hand be called IC1, since its separate voltage supply symbol is not counted as a gate.

Example:

```
ADD PWR 0 REQUEST 
```

fetches the PWR symbol (e.g. a power pin symbol), and defines a Swaplevel of 0 (not swappable) and the Addlevel *Request* for it.

ARC

Function

Draw an arc of variable diameter, width, and length.

Syntax

```
ARC ['signal_name'] [CW | CCW] [ROUND | FLAT] [width] ▯ ▯ ▯
```

Mouse keys

Center selects the layer.
Right changes the orientation.

See also CHANGE, WIRE, CIRCLE

The ARC command, followed by three mouse clicks on a drawing, draws an arc of defined width. The first point defines a point on a circle, the second its diameter. Entering the second coordinate reduces the circle to a semi-circle, while the right button alters the direction from first to second point. Entry of a third coordinate truncates the semi-circle to an arc extending to a point defined by the intersection of the circumference and a line between the third point and the arc center.

The parameters CW and CCW enable you to define the direction of the arc (clockwise or counterclockwise). ROUND and FLAT define whether the arc endings are round or flat, respectively.

Signal name

The `signal_name` parameter is intended mainly to be used in script files that read in generated data. If a `signal_name` is given, the arc will be added to that signal and no automatic checks will be performed.
This feature should be used with great care because it could result in short circuits if an arc is placed in a way that it would connect different signals. Please run a Design Rule Check after using the ARC command with the `signal_name` parameter!

Line Width

The parameter "width" defines the thickness of the drawn line. It can be changed or predefined with the command:

```
CHANGE WIDTH width;
```

The adjusted width is identical to the line width for wires.

Arcs with angles of 0 or 360 degrees or a radius of 0 are not accepted.

Example for text input:

```
GRID inch 1;
ARC CW (0 1) (0 -1) (1 0);
```

generates a 90-degree arc with the center at the origin.

ASSIGN

Function

Modify key assignments.

Syntax

```
ASSIGN
ASSIGN function_key command..;
ASSIGN function_key;

function_key = modifier+key
```

`modifier =` any combination of `S` (Shift), `C` (Control), `A` (Alt) and `M` (Cmd, Mac OS X only)
`key = F1..F12, A-Z, 0-9, BS` (Backspace)

See also SCRIPT, Keyboard and Mouse

The ASSIGN command can be used to define the meaning of the function keys `F1` thru `F12`, the letter keys `A` thru `Z`, the (upper) digit keys `0` thru `9` and the `backspace` key (each also in combination with modifier keys).

The ASSIGN command without parameters displays the present key assignments in a dialog, which also allows you to modify these settings.

Keys can be assigned a single command or multiple commands. The command sequence to be assigned should be enclosed in apostrophes.

If `key` is one of `A-Z` or `0-9`, the `modifier` must contain at least `A`, `C` or `M`.

The `M` modifier is only available on **Mac OS X**.

Please note that any special operating system function assigned to a function key will be overwritten by the ASSIGN command (depending on the operating system, ASSIGN may not be able to overwrite certain function keys).
If you assign to a letter key together with the modifier `A`, (e.g. `A+F`), a corresponding hotkey from the pulldown menu is no longer available.

To remove an assignment from a key you can enter `ASSIGN` with only the function_key code, but no command.

Examples

```
ASSIGN F7 'change layer top; route';
ASS A+F7 'cha lay to; rou';
ASSIGN C+F10 menu add mov rou ''';''' edit;
ASSIGN CA+R 'route';
```

The first two examples have the same effect, since EAGLE allows abbreviations not only with commands but also with parameters (as long as they are unmistakable).

Please note that here, for instance, the change layer top command is terminated by a semicolon, but not the route command. The reason is that in the first case the command already contains all the necessary parameters, while in the second case coordinates still have to be added (usually with the mouse). Therefore the ROUTE command must not be deactivated by a semicolon.

Define Command Menu

If you want to assign the MENU command to a key, the separator character in the MENU command (semicolon) has to be enclosed in three pairs of apostrophes (see the third example). This semicolon will show up in the new menu.

Presetting of key assignments

`F1 HELP`	Help function
`Alt+F2 WINDOW FIT`	The whole drawing is displayed
`F2 WINDOW;`	Screen redraw
`F3 WINDOW 2`	Zoom in by a factor of 2
`F4 WINDOW 0.5`	Zoom out by a factor of 2
`F5 WINDOW (@);`	Cursor pos. is new center
`F6 GRID;`	Grid on/off

Further, many useful key assignments are contained in the initialisation script eagle.scr and can be adjusted to your individual needs.

ATTRIBUTE

Function

Definition of attributes for parts.

Syntax

```
ATTRIBUTE name [ 'value' ] [ options ]
ATTRIBUTE part_name attribute_name
ATTRIBUTE part_name attribute_name 'attribute_value'
[ [ orientation ] ▯ ]
ATTRIBUTE part_name attribute_name DELETE
ATTRIBUTE * [ name [ 'value' ] ]
ATTRIBUTE * name DELETE
ATTRIBUTE ▯..
```

See also TECHNOLOGY, NAME, VALUE, SMASH, TEXT

See the description of `orientation` at ADD.

An *attribute* is an arbitrary combination of a *name* and a *value*, that can be used to specify any kind of information for a given part.

Attributes can be defined in the library (for individual devices), in the schematic (for an actual part) or in the board (for an actual element). Attributes defined on the device level will be used for every part of that device type in the schematic. In a schematic, additional attributes can be defined for each part, and existing attributes from the devices can be overwritten with new values (if the attributes have been defined as *variable*). An element in the board has all the attributes of its corresponding part, and can have further attributes of its own.

Attributes in the Library

In a library the ATTRIBUTE command can be used to define the attributes of a given technology variant, using the syntax

```
ATTRIBUTE name [ 'value' ] [ options ]
```

The `name` may consist of any letters, digits, '_', '#' and '-' and may have any length; the first character must not be '-', though. Names are treated case insensitive, so PartNo is the same as PARTNO. The `value` may contain any characters and must be enclosed in single quotes.

The valid `options` are:

`delete`	Delete the attribute with the given name from all technology variants (in this case there must be no 'value').
`variable`	Mark this attribute as *variable*, so that it can be overwritten in the schematic (this is the default).
`constant`	Attributes marked as *constant* cannot be overwritten in the schematic (unless the user insists).

Options may be abbreviated and are case insensitive.

An already existing attribute can be switched between *variable* and *constant* without the need to repeat its value, as in

`ATTRIBUTE ABC '123'`	(variable by default)
`ATTRIBUTE ABC constant`	(ABC retains its value '123')

If the value of an attribute is changed, its *constant/variable* setting remains unchanged (unless explicitly given).

The attribute name **_EXTERNAL_** is reserved for marking of external devices (see PACKAGE).

Some special attribute names are not allowed, since they would interfere with the already existing text variables. If an attribute named VALUE is defined, its value will be used to initialize the actual value when placing a part in a schematic (in case the device set has 'Value On').

Attributes in the Schematic

In a schematic, the ATTRIBUTE command can be used to assign attributes to a part, in which case the value of such an attribute overwrites the value of the attribute with the same name in the library (if the device has such an attribute and allows overwriting). A part may also be given attributes that are not defined in the library at all.

Selecting the ATTRIBUTE command and clicking on a part shows a dialog in which all attributes of that part are listed and can be edited.

For a fully textual definition of an attribute, the following syntax can be used:

```
ATTRIBUTE part_name attribute_name 'attribute_value' orientation ▯
```

Note that in case of a multi-gate part, actually one of the gates (i.e. "instances") is selected. When selecting it via a mouse click it is already clear which gate is meant, while when selecting it via part_name, the full name consisting of the part and gate name should be given. While a specific part can only have one attribute with a given name, the attribute can be attached to any or all of its gates. If only the part name is given, the first visible gate will be implicitly selected.

If no coordinates are given (and the command is terminated with a `';'`), the behavior depends on whether the given attribute already exists for that part (either in the device or in the schematic). If the attribute already exists, only its value will be changed. If it doesn't exist yet, a new attribute with the given name and value will be placed at the origin of the selected gate of the part.

To delete an attribute from a part, the command

```
ATTRIBUTE part_name attribute_name DELETE
```

can be used.

When defining attributes via the command line or a script, use the CHANGE DISPLAY command to define which parts of the attribute (name, value, both or none of these) shall be visible.

Attributes in the Board

In a board, attributes can be assigned to elements with the ATTRIBUTE command, much the same as in schematics. By default elements have all the attributes that are defined for their part in the schematic (and their device in the library). Attributes with the same name for a given element/part pair will always have the same value (through Forward&Back Annotation). Elements can have additional attributes that are not present in the schematic or library.

Global attributes

Global attributes can be defined in boards and schematics by using `'*'` as the part name (which implies that this attribute applies to *all* parts). Alternatively global attributes can be defined through the menu option "Edit/Global attributes...". The global attributes of board and schematic are handled separately and are not connected via Forward&Back-Annotation.

Such an attribute could for instance be the author of a drawing, and can be used in the title block of a drawing's frame. It will be shown on every schematic sheet that has a drawing frame that contains a text variable with the same name.

Selecting the layer

Unlike other commands (like WIRE, for instance), the ATTRIBUTE command keeps track of its last used layer by itself. This has the advantage of making sure that attributes are always drawn into the right layer, no matter what layers other commands draw into. The downside of this is that the usual way of setting the layer in a script, as in

```
LAYER layer;
WIRE (1 2) (3 4);
```

doesn't work here. The layer needs to be selected while the ATTRIBUTE command is already active, which can be done like this

```
ATTRIBUTE parameters
LAYER layer
more parameters;
```

Note that the ATTRIBUTE line is **not** terminated with a `';'`, and that the LAYER command starts on a new line.
The commands

```
ATTRIBUTE
LAYER layer;
```

set the layer to use with subsequent ATTRIBUTE commands.

Examples

First the package and technology has to be selected (in case there is more than one) and then attributes for that technology can be defined:

```
PACKAGE N;
TECHNOLOGY LS;
ATTRIBUTE PartNo '12345-ABC';
ATTRIBUTE Temp '100K' constant;
ATTRIBUTE Remark 'mount manually';
```

AUTO

Function

Starts the Autorouter

Syntax

```
AUTO;
AUTO signal_name..;
AUTO ! signal_name..;
AUTO ▯..;
AUTO FOLLOWME
AUTO LOAD|SAVE filename;
```

See also SIGNAL, ROUTE, WIRE, RATSNEST, SET

The AUTO command activates the integrated Autorouter. If signal names are specified or signals are selected with the mouse, only these signals are routed. Without parameters the command will try to route all signals. If a "!" character is specified all signals are routed except the signals following the "!" character. The "!" character must be the first parameter and must show up only once.

The `LOAD` and `SAVE` options can be used to load the Autorouter parameters from or save them to the given file. If *filename* doesn't have the extension `".ctl"` it will be appended automatically.

Without any parameters (or if no terminating `';'` is given), the AUTO command opens a dialog in which the parameters that control the routing algorithm can be configured. The special option `FOLLOWME` opens this dialog in a mode where only the parameters controlling the Follow-me router can be modified.

Example

```
AUTO ! GND VCC;
```

In every case the semicolon is necessary as a terminator. A menu for adjusting the Autorouter control parameters opens if you select AUTO from the command menu or type in AUTO from the keyboard (followed by Return key).

Wildcards

If a `signal_name` parameter is given, the characters `'*'`, `'?'` and `'[]'` are *wildcards* and have the following meaning:

`*`	matches any number of any characters
`?`	matches exactly one character
`[...]`	matches any of the characters between the brackets

If any of these characters shall be matched exactly as such, it has to be enclosed in brackets. For example, `abc[*]ghi` would match `abc*ghi` and not `abcdefghi`.

A range of characters can be given as `[a-z]`, which results in any character in the range `'a'`...`'z'`.

Polygons

When the Autorouter is started all Polygons are calculated.

Protocol File

A protocol file (name.pro) is generated automatically.

Board Size

The Autorouter puts a rectangle around all objects in the board and takes the size of this rectangle as the routing area. Wires in the Dimension layer are border lines for the Autorouter. This means you can delimit the route area with closed lines drawn into this layer with the WIRE command.

In practice you draw the board outlines into the Dimension layer with the WIRE command and place the components within this area.

Signals

Signals defined with EAGLE's SIGNAL command, polygons, and wires drawn onto the Top, Bottom, and ROUTE2...15 layers are recognized by the Autorouter.

Restricted Areas

Objects in the layers tRestrict, bRestrict, and vRestrict are treated as restricted areas for the Top and Bottom side and for vias respectively.

If you want the Autorouter not to use a layer, select "N/A" in the preferred direction field.

Canceling

If you cancel the Autorouter by clicking on the STOP button, any airwires that have not yet been routed, are not automatically recalculated. Use the RATSNEST command to do this.

BOARD

Function

Converts a schematic into a board.

Syntax

```
BOARD [ grid ]
```

See also EDIT

The command BOARD is used to convert a schematic drawing into a board.

If the board already exists, it will be loaded into a board window.

If the board does not exist, you will be asked whether to create that new board. If a `grid` is given, the parts on the board will be placed in the given raster, as in

```
BOARD 5mm
```

which would place the parts in a 5 millimeter raster (default is 50mil). The number must be given with a unit, and the maximum allowed value is 10mm.

The BOARD command will never overwrite an existing board file. To create a new board file if there is already a file with that name, you have to remove that file first.

Creating a board from a schematic

The first time you edit a board the program checks if there is a schematic with the same name in the same directory and gives you the choice to create the board from that schematic. If you have opened a schematic window and want to create a board, just type

```
edit .brd
```

in the editor window's command line.

All relevant data from the schematic file (name.sch) will be converted to a board file (name.brd). The new board is loaded automatically as an empty card with a size of 160x100mm (Light edition: 100x80mm). All packages and connections are shown on the left side of the board. Supply pins are already connected (see PIN command).

If you need board outlines different to the ones that are generated by default, simply delete the respective lines and use the WIRE command to draw your own outlines into the *Dimension* layer. The recommended width for these lines is 0.

A board file cannot be generated:

- if there are gates in the schematic from a device for which no package has been defined (error message: "device name has no package). Exception: if there are only pins with Direction "Sup" (supply symbols)
- if there are gates in the schematic from a device for which not all pins have been assigned to related pads of a package (error message: "device name has unconnected pins"). Exception: device without pins (e.g. frames)

BUS

Function

Draws buses in a schematic.

Syntax

```
BUS [bus_name] ▯ [curve | @radius] ▯..
```

Mouse keys

Right changes the wire bend style (see SET Wire_Bend).
Shift+Right reverses the direction of switching bend styles.
Ctrl+Right toggles between corresponding bend styles.

See also NET, NAME, SET

The command BUS is used to draw bus connections onto the Bus layer of a schematic diagram. Bus_name has the following form:

```
SYNONYM:partbus,partbus,..
```

where SYNONYM can be any name. Partbus is either a simple net name or a bus name range of the following form:

```
Name[LowestIndex..HighestIndex]
```

where the following condition must be met:

0 <= LowestIndex <= HighestIndex <= 511

If a name is used with a range, that name must not end with digits, because it would become unclear which digits belong to the Name and which belong to the range.

If a bus wire is placed at a point where there is already another bus wire, the current bus wire will be ended at that point. This function can be disabled with "`SET AUTO_END_NET OFF;`", or by unchecking "Options/Set/Misc/Auto end net and bus".

If the *curve* or *@radius* parameter is given, an arc can be drawn as part of the bus (see the detailed description in the WIRE command).

Bus name examples

```
A[0..15]
RESET
DB[0..7],A[3..4]
ATBUS:A[0..31],B[0..31],RESET,CLOCK,IOSEL[0..1]
```

If no bus name is used, a name of the form B$1 is automatically allocated. This name can be changed with the NAME command at any time.

The line width used by the bus can be defined for example with

```
SET Bus_Wire_Width 40;
```

to be 40 mil. (Default: 30 mil).

Inverted signals

The name of an inverted signal ("active low") can be displayed overlined if it is preceded with an exclamation mark (`'!'`), as in

```
  ATBUS:A[0..31],B[0..31],!RESET,CLOCK,IOSEL[0..1]
```

which would result in

$\texttt{ATBUS:A[0..31],B[0..31],}\overline{\texttt{RESET}}\texttt{,CLOCK,IOSEL[0..1]}$

You can find further details about this in the description of the TEXT command.

CHANGE

Function

Changes parameters.

Syntax

```
CHANGE option • •..
```

Mouse keys

Ctrl+Right changes parameter of the group.

The CHANGE command is used to change or preset properties of objects. The objects are clicked on with the mouse after the desired parameters have been selected from the CHANGE command menu or have been typed in from the keyboard.

Parameters adjusted with the CHANGE command remain as preset properties for objects added later.

All values in the CHANGE command are used according to the actual grid unit.

Change Groups

When using the CHANGE command with a group, the group is first identified with the GROUP command before entering the CHANGE command with appropriate parameters. The right button of the mouse is then used to execute the changes.

What can be changed?

Layer	`CHANGE LAYER name	number`
Text	`CHANGE TEXT [ text ]`
Text height	`CHANGE SIZE value`
Text thickness	`CHANGE RATIO ratio`
Text line distance	`CHANGE LINEDISTANCE value`
Text font	`CHANGE FONT VECTOR	PROPORTIONAL	FIXED`
Text alignment	`CHANGE ALIGN BOTTOM	LEFT	CENTER	TOP	RIGHT`
Wire width	`CHANGE WIDTH value`
Wire style	`CHANGE STYLE value`
Arc cap	`CHANGE CAP ROUND	FLAT`
Pad shape	`CHANGE SHAPE SQUARE	ROUND	OCTAGON	LONG	OFFSET`
Pad/via/smd flags	`CHANGE STOP	CREAM	THERMALS	FIRST OFF	ON`
Pad/via diameter	`CHANGE DIAMETER diameter`
Pad/via/hole drill	`CHANGE DRILL value`
Via layers	`CHANGE VIA from-to`
Smd dimensions	`CHANGE SMD width height`
Smd roundness	`CHANGE ROUNDNESS value`
Pin	`CHANGE DIRECTION NC	IN	OUT	IO	OC	HIZ	SUP |`

parameters	`PAS | PWR`
	`CHANGE FUNCTION NONE | DOT | CLK | DOTCLK`
	`CHANGE LENGTH POINT | SHORT | MIDDLE | LONG`
	`CHANGE VISIBLE BOTH | PAD | PIN | OFF`
	`CHANGE SWAPLEVEL number`
Polygon parameters	`CHANGE THERMALS OFF | ON`
	`CHANGE ORPHANS OFF | ON`
	`CHANGE ISOLATE distance`
	`CHANGE POUR SOLID | HATCH | CUTOUT`
	`CHANGE RANK value`
	`CHANGE SPACING distance`
Gate parameters	`CHANGE SWAPLEVEL number`
	`CHANGE ADDLEVEL NEXT | MUST | ALWAYS | CAN | REQUEST`
Net class	`CHANGE CLASS number | name`
Package	`CHANGE PACKAGE part_name [device_name] | 'device_name' [part_name]`
Technology	`CHANGE TECHNOLOGY part_name [device_name] | 'device_name' [part_name]`
Attribute display	`CHANGE DISPLAY OFF | VALUE | NAME | BOTH`
Frame parameters	`CHANGE COLUMS value`
	`CHANGE ROWS value`
	`CHANGE BORDER NONE | BOTTOM | RIGHT | TOP | LEFT | ALL`
Label	`CHANGE XREF OFF | ON`
Dimension type	`CHANGE DTYPE value`
Dimension unit	`CHANGE DUNIT [MIC | MM | MIL | INCH] [OFF | ON] [precision]`
Dimension line	`CHANGE DLINE width [ extension_width [ extension_length [ extension_offset ]]] (extension values can be set to AUTO; unchanged preceding values can be skipped with '-')`

CIRCLE

Function

Adds circles to a drawing.

Syntax

```
CIRCLE ▯ ▯.. [center, circumference]
CIRCLE width ▯ ▯..
```

Mouse keys

Center selects the layer.

See also CHANGE, WIRE

The CIRCLE command is used to create circles. Circles in the layers tRestrict, bRestrict, and vRestrict define restricted areas. They should be defined with a width of 0.

The width parameter defines the width of the circle's circumference and is the same parameter as used in the WIRE command. The width can be changed with the command:

```
CHANGE WIDTH width;
```

where *width* is the desired value in the current unit.

A circle defined with a width of 0 will be filled.

Example

```
GRID inch 1;
CIRCLE (0 0) (1 0);
```

generates a circle with a radius of 1 inch and the center at the origin.

CLASS

Function

Define and use net classes.

Syntax

```
CLASS
CLASS number|name
CLASS number [ name [ width [ clearance [ drill ] ] ] ]
[ number:clearance .. ]
```

See also Design Rules, NET, SIGNAL, CHANGE

The CLASS command is used to define or use net classes.

Without parameters, it offers a dialog in which the net classes can be defined.

If only a `number` or `name` is given, the net class with the given number or name is selected and will be used for subsequent NET and SIGNAL commands.

If both a `number` and a `name` are given, the net class with the given number will be assigned all the following values and will also be used for subsequent NET and SIGNAL commands. If any of the parameters following `name` are omitted, the net class will keep its respective value.

If `number` is negative, the net class with the absolute value of `number` will be cleared. The default net class `0` can't be cleared.

Net class names are handled case insensitive, so SUPPLY would be the same as Supply or SuPpLy.

Using several net classes in a drawing increases the time the Autorouter needs to do its job. Therefore it makes sense to use only as few net classes as necessary (only the number of net classes actually used by nets or signals count here, not the number of defined net classes).

In order to avoid conflicts when CUT/PASTEing between drawings it makes sense to define the same net classes under the same numbers in all drawings.

The Autorouter processes signals sorted by their total width requirements (Width plus Clearance), starting with those that require the most space. The bus router only routes signals with net class `0`.

The net class of an existing net/signal can be changed with the CHANGE command.

Width

The *width* parameter defines a minimum width that all objects in this net class must have.

Clearance

The *clearance* parameter defines the minimum clearance between objects of different signals in this net class and objects in other net classes.

Drill

The *drill* parameter defines a minimum drill size that all objects in this net class must have (only applies to objects that actually have a drill parameter, like pads and vias).

Clearance between net classes

If a clearance is given in the form `number:clearance`, it defines the minimum clearance between signals in this net class and signals in the net class with the given `number`. The command

```
CLASS 3 1:0.6mm 2:0.8mm
```

defines a minimum clearance of 0.6mm between signals in net classes 1 and 3, and one of 0.8mm between signals in net classes 2 and 3. Note that the numbers in `number:clearance` must be less than or equal to the number of the net class itself (`'3'` in the above example), so

```
CLASS 3 1:0.6mm 2:0.8mm 3:0.2mm
```

would also be valid, whereas

```
CLASS 3 1:0.6mm 2:0.8mm 3:0.2mm 4:0.5mm
```

would not be allowed.

CLOSE

Function

Closes an editor window.

Syntax

`CLOSE`

See also OPEN, EDIT, WRITE, SCRIPT

The CLOSE command is used to close an editor window. If the drawing you are editing has been modified you will be prompted whether you wish to save it.

This command is mainly used in script files.

CONNECT

Function

Assigns pads to pins.

Syntax

```
    CONNECT
    CONNECT [ ALL | ANY ] gate_name.pin_name pad_name..
    CONNECT [ ALL | ANY ] pin_name pad_name..
```

See also PREFIX, OPEN, CLOSE, SCRIPT

This command is used in the device editing mode in order to define the relationship between the pins of a gate and the pads of the corresponding package in the library. First of all, it is necessary to define which package is to be used by means of the PACKAGE command.

If the CONNECT command is invoked without parameters, a dialog is presented which allows you to interactively assign the connections.

Device with one Gate

If only one gate is included in a device, the parameter gate_name can be dropped, e.g.:

```
CONNECT gnd 1 rdy 2 phi1 3 !irq 4 nc1 5...
```

(Note: "!" is used to indicate inverted data signals.)

Device with several Gates

If several gates are present in a device, parameters must be entered with gate_name, pin_name and pad_name each time. For example:

```
CONNECT A.I1     1  A.I2  2   A.O  3;
CONNECT B.I1     4  B.I2  5   B.O  6;
CONNECT C.I1    13  C.I2  12  C.O 11;
CONNECT D.I1    10  D.I2  9   D.O  8;
CONNECT PWR.gnd  7;
CONNECT PWR.VCC 14;
```

In this case, the connections for four NAND gates of a good old 7400 are allocated. The device includes five gates - A, B, C, D, and PWR. The gate inputs are named I1 and I2 while the output is named O.

The CONNECT command can be repeated as often as required. It may be used with all pin/pad connections or with only certain pins. Each new CONNECT command overwrites the previous conditions for the relevant pins.

Note that if you have a large number of connections in a single device, the CONNECT command works a lot faster if all connections are given in one single call, like shown in the example below.

Several Pads connected to the same Pin

Some parts, like power amplifiers or BGA chips, may have several pads that are connected internally. This may be done for better heat dissipation or to allow for higher currents. The CONNECT command can handle these cases by simply listing all related pad names, separated by blanks (and therefore enclosed in single quotes), as in

```
CONNECT ALL I1 '1 3 5';
CONNECT ANY O1 '2 4 6';
```

In the first example the pin `I1` is connected to the three pads `1`, `3` and `5`. If the pin `I1` is connected to a net in the schematic, all three pads must be explicitly connected to the corresponding signal in the board.
In the second example, the keyword `ANY` indicates that any one (or even all) of the pads `2`, `4` or `6` can be connected to the signal. It is even allowed to use this internal connection as a "bridge" by connecting one segment of the signal to, say, pad `2`, while connecting the rest of the signal to pad `6`, without any explicit external connection between these two pads. Of course, when designing a library part and using `ANY` in a CONNECT command, you need to be sure that the part will be able to handle cross currents running through its pads. If in doubt, use `ALL` (which is the default and may be omitted).

If a pin name would collide with one of the keywords `ALL` or `ANY`, the pin name needs to be enclosed with single quotes. As soon as one of these keywords appears in a CONNECT command, it applies to all pin/pad connections that follow it, until a different keyword is seen, as in

```
CONNECT 'A' '1' 'B' '2' ANY 'C' '3 4 5 6' 'D' '7 8' ALL 'E' '9 10 11';
```

The RATSNEST and AUTO command will handle the `ALL` and `ANY` cases appropriately.

In the CONNECT dialog the "Connect" button creates a new connection between the selected pin and the selected pads. There can be more than one pad selected, in which case all of these pads will be connected to the selected pin. Use the Ctrl and Shift keys in the usual way to mark more than one pad as selected.
The "Append" button adds the selected pads to the current connection.
The "Disconnect" button removes the selected connection and puts the pin and pads back in their separate lists, keeping them selected so that it is easy to make modifications. A Disconnect immediately followed by a Connect results in the same configuration as before the Disconnect (and vice versa).
If a connection contains more than one pad, an icon indicates whether any or all of these pads need to be externally connected to a signal. Click on this icon to toggle the mode. When such a connection item is expanded, all the pads are listed separately, and clicking on Disconnect with one of the pads selected will only disconnect that one pad.

Gate or Pin names that contain periods

If a gate or pin name contains a period, simply enter them without any special consideration (no quoting or escape characters are necessary).

Example

```
ed 6502.dev;
prefix 'IC';
package dil40;
connect gnd 1 rdy 2 phi1 3 !irq 4 nc1 5 !nmi 6 \
        sync 7 vcc 8  a0 9 a1 10 a2 11 a3 12 a4 \
        13 a5 14 a6 15 a7 16 a8 17 a9 18 a10 19 \
        a11 20 p$0 21 a12 22 a13 23 a14 24 a15 \
        25 d7 26 d6 27 d5 28 d4 29 d3 30 d2 31 \
        d1 32 d0 33 r/w 34 nc2 35 nc3 36 phi0 37 \
        so 38 phi2 39 !res 40;
```

If a command is continued at the next line, it is advisable to insert the character "\" at the end of the line to ensure the following text cannot be confused with an EAGLE command.

Confusing parameters with commands can also be avoided by enclosing the parameters in apostrophes.

COPY

Function

Copy objects.

Syntax

```
COPY ▯ ▯..
COPY deviceset@library [name]
COPY package@library [name]
```

Mouse keys

Ctrl+Left selects an object at its origin.
Ctrl+Right selects the group.
Center mirrors the selected object or the group.
Right rotates the selected object or the group.
Shift+Right reverses the direction of rotating.

See also GROUP, CUT, PASTE, ADD, INVOKE, POLYGON

The COPY command is used to copy objects within the same drawing, or between libraries. EAGLE will generate a new name for the copy but will retain the old value. When copying signals (wires), buses, and nets the names are retained, but in all other cases a new name is assigned.

Copy to the system's clipboard

The COPY command in EAGLE traditionally only copied objects by clicking on them with the mouse and placing them within the same drawing. It also copied library objects between libraries. However, before version 6 it did not copy the current group selection to the system's clipboard, like other Windows programs do. In EAGLE, the CUT command was used for this. Unfortunately, this has irritated Windows users time and again, so beginning with version 6, the COPY command also copies the selected group of objects to the system's clipboard, while still retaining the full functionality of previous versions. If you don't like this, you can use the SET command

```
SET Cmd.Copy.ClassicEagleMode 1
```

to get back the original behavior of the COPY command (as well as the CUT command).

Copy Wires

If you copy wires or polygons, belonging to a signal, the copy will belong to the same signal. Please note, for this reason, if two wires overlap after the use of the COPY command, the DRC will not register an error. If a net or bus wire is copied in a schematic, it belongs to the same segment as the original wire, even if there is no visible connection. This can lead to unexpected effects, for instance when renaming them later. Therefore COPY should not be used with net or bus wires, respectively.

Copy Parts

When copying a part in a schematic, there will always be a new instance of the complete part added, even if only a single gate of a multi-gate part is selected. In addition to the selected gate, any other gates of that device which have Add-Level MUST or ALWAYS will automatically be invoked.

If you just want to use another gate of a multi-gate part, you should use the INVOKE command instead.

Copy library objects

By writing `COPY deviceset@library` or `COPY package@library` you can copy a device set or a package from a given library into the currently loaded library. `library` can be either a plain library name or a file path (see ADD command). In case of ambiguity you can add the suffix `[.dev]` for device sets or `[.pac]` for packages. If an additional `name` is given, the copied object will be given that name. This can also be done through the library objects' context menu or via *Drag&Drop* from the Control Panel's tree view.

Note that any existing library objects (device sets, symbols or packages) used by the copied library object will be automatically updated.

Copy a group

Copying a group by selecting it with the right mouse button is actually done by doing an implicit CUT operation, immediately followed by a PASTE.

Copy objects to an other sheet

To copy objects to an other sheet of the same schematic, you need to GROUP the objects, do a COPY (or CUT), switch to the target sheet and then do PASTE.

CUT

Function

Copies a group into the clipboard.

Syntax

```
CUT ▯
CUT;
```

See also PASTE, COPY, GROUP

Parts of a drawing (or even a whole board) can be copied onto other drawings by means of the commands CUT and PASTE.

To do this you first define a group (GROUP command). Then use the CUT command, followed by a reference point (mouse click or coordinates (x y)) to put the selected objects into the buffer. `CUT;` automatically puts the reference point at the center of the selected objects (snapped to the grid). Now you can change to an other drawing (EDIT) and copy the contents of the buffer onto the new drawing by executing the PASTE command.

Reference Point

If you click the mouse after selecting the CUT command, the position of the mouse cursor defines a reference point for the group, i.e. when using the PASTE command, the mouse cursor will be at the exact position of the group.

Note

Unlike other (Windows-) programs EAGLE's CUT command does not physically remove the marked group from the drawing; it only copies the group into the clipboard. Unfortunately, this has irritated Windows users time any again, so beginning with version 6, the CUT command no longer appears in the main pulldown menu and the command button toolbar (it is still available from the command line and within scripts). Windows users will simply use the COPY command to copy the selected group of objects into the system's clipboard. This, however, will not allow them to define an explicit reference point for the selected group. It will always be selected at the center of the group's bounding rectangle. Using a reference point is only possible with the CUT command. If you don't like this, you can use the SET command

```
SET Cmd.Copy.ClassicEagleMode 1
```

to get back the original behavior of the CUT command (as well as the COPY command).

DELETE

Function

Deletes objects.

Syntax

```
DELETE ▯..
DELETE name ..
DELETE SIGNALS
```

Mouse keys

Shift+Left deletes higher level object.
Ctrl+Left deletes a wire joint.
Ctrl+Right deletes the group.

See also RIPUP, DRC, GROUP

The DELETE command is used to delete the selected object.

Parts, pads, smds, pins and gates can also be selected by their name, which is especially useful if the object is outside the currently shown window area. Note that when selecting a multi-gate part in a schematic by name, you will need to enter the full instance name, consisting of part and gate name.

Attributes of parts can be selected by entering the concatenation of part name and attribute name, as in `R5>VALUE`.

Clicking the right mouse button deletes a previously defined GROUP.

After deleting a group it is possible that airwires which have been newly created due to the removal of a component may be "left over", because they have not been part of the original group. In such a case you should re-calculate the airwires with the RATSNEST command.

With active Forward&Back Annotation, no wires or vias can be deleted from a signal that is connected to components in a board. Also, no components can be deleted that have signals connected to them. Modifications like these have to be done in the schematic.

Use the RIPUP command to convert an already routed connection back into an airwire.

The DELETE command has no effect on layers that are not visible (refer to DISPLAY).

The DRC might generate error polygons which can only be deleted with DRC CLEAR.

Deleting Wire Joints

If the DELETE command, with the `Ctrl` key pressed, is applied to the joining point of two wires, these wires are combined to form one straight wire. For this to work the two wires must be in the same layer and have the same width and line style, and must both have round endings (in case of arcs).

Deleting Polygon Corners

The DELETE command deletes one corner at a time from a polygon. The whole polygon is deleted if there are only three corners left.

Deleting Components

Components can be deleted only if the tOrigins layer (or bOrigins with mirrored components) is visible and if (with active Forward&Back Annotation) no signals are connected to the component (see also REPLACE). Please note that an element may appear to be not connected (no airwires or wires leading to any of it's pads), while in fact it **is** connected to a supply voltage through an implicit power pin. In such a case you can only delete the corresponding part in the schematic.

Deleting Junctions, Nets, and Buses

The following rules apply:

- If a bus is split into two parts, both keep the initial name.
- If a net is split into two parts, the larger one keeps the initial name while the smaller one gets a new (generated) name.
- After the DELETE command, labels belong to the segment next to them.
- If a junction point is deleted, the net is separated at this location. Please check the names of the segments with the SHOW command.

Deleting Supply Symbols

If the last supply symbol of a given type is deleted from a net segment that has the same name as the deleted supply pin, that segment is given a newly generated name (if there are no other supply symbols still attached to that segment) or the name of one of the remaining supply symbols.

Deleting Signals

If you select wires (tracks) or vias belonging to a signal with the DELETE command three cases have to be considered:

- The signal is split into two parts. EAGLE will generate a new name for the smaller part of the signal and keep the previous name for the larger one.
- The signal is deleted from one end. The remaining part of the signal will keep the previous name.
- The signal had only one airwire. It will be deleted completely and its name won't exist any longer.

After wires or vias have been deleted from a signal which contains polygons, all polygons belong to the signal keeping the original name (usually the bigger part).

Deleting all Signals

DELETE SIGNALS can be used to delete all signals on a board. This is useful if you want to read in a new or changed netlist (see EXPORT). Only those signals are deleted which are connected to pads.

If you want to delete a part that has the name SIGNALS, you need to write the name in single quotes.

Deleting higher level objects

If the `Shift` key is pressed when clicking on an object, the object that is hierarchically above the selected one will be deleted. This applies to the following objects:

`Gate`	Deletes the entire part containing this gate (even if the gates are spread over several sheets). If f/b annotation is active, the wires connected to the element in the board will not be ripped up (as opposed to deleting a single gate), except for those cases where a pin of the deleted part is only connected directly to one single other pin and no net wire
`Polygon Wire`	Deletes the entire polygon
`Net/Bus Wire`	Deletes the entire net or bus segment

Don't forget: Deleting can be reversed by the UNDO command!

DESCRIPTION

Function

Defines the description of a drawing or a library object.

Syntax

```
DESCRIPTION [ * ]
DESCRIPTION [ * ] description_string;
```

See also CONNECT, PACKAGE, VALUE

This command is used to define or edit the description of a drawing or a library object.

The `description_string` may contain HTML tags.

The first non-blank line of `description_string` will be used as a short descriptive text (*headline*) in the Control Panel.

The DESCRIPTION command without a parameter opens a dialog in which the text can be edited. The upper pane of this dialog shows the formatted text, in case it contains HTML tags, while the lower pane is used to edit the raw text. At the very top of the dialog the *headline* is displayed as it would result from the first non-blank line of the description. The headline is stripped of any HTML tags.

By default the DESCRIPTION command works on the description of the object that is currently edited, like a device set, package, symbol, board or sheet. If, in a library, there is no currently edited object (as can be the case after it has been newly loaded) the description of the library will be changed.

To explicitly access the description of a library, even if a device, package or symbol is already being edited, enter the asterisk character (`'*'`) as the first parameter to the DESCRIPTION command. This is also the way to access the description of a schematic, as opposed to the description of an individual sheet.

Example

```
DESCRIPTION '<b>Quad NAND</b><p>\nFour NAND gates with 2 inputs each.';
```

This would result in

Quad NAND

Four NAND gates with 2 inputs each.

DIMENSION

Function

Adds dimensioning to a drawing.

Syntax

```
DIMENSION [dtype] ▯ ▯..
```

Mouse keys

Center selects the layer.
Right changes the dtype.
Shift+Right reverses the direction of changing the dtype.
Ctrl+Left when starting a dimension does not select an object.

See also WIRE, CHANGE, CIRCLE, HOLE

The DIMENSION command adds dimensioning to a drawing. It can either be applied to an object, or it can draw arbitrary dimensions.

If the first point selects an object, a suitable dimension object is generated as follows:

straight wire	linear dimension displaying the distance between the end points of the wire
curved wire	radius dimension displaying the radius of the arc
circle	diameter dimension displaying the diameter of the circle
hole	diameter dimension displaying the diameter of the hole

If no object is selected, or a wire is selected at one of its end points, a dimension object is generated according to the current dimension type. If this dimension type is not the one that is needed, the right mouse button can be clicked to loop through the various types.

To draw an arbitrary dimension even at close proximity to an object that would trigger a specific kind of dimension, press the `Ctrl` key with the first click. This may also be useful when using the DIMENSION command in a script (by adding the 'C' modifier to the first coordinate), to make sure the dimension appears exactly as intended.

The way in which a dimension object is drawn (line, unit, precision) can be configured with "CHANGE DLINE/DUNIT" or with its properties dialog. Note that the "Unit" parameter in this dialog refers to the unit in which the actual numbers of the dimension object will be displayed.

Dimension Type

Every dimension object has three coordinates that define its reference points and an alignment point. How these coordinates are actually interpreted to display a dimension object depends on the dtype property.

Parallel

A *parallel* dimension displays the distance between its first and second reference point. The dimension line is parallel to the line going through its reference points, and it runs through the given alignment point. The actual position of the alignment point doesn't matter, only its distance from the the line through its reference points is taken into account. When a parallel dimension object is newly created or modified, the alignment point is normalized, so that it lies in the middle of the dimension line.

Horizontal

Same as *parallel*, but the dimension line extends only in X direction, and it displays only the X distance between the reference points.

Vertical

Like *horizontal*, but for Y.

Radius

A *radius* dimension displays the distance between its first and second reference point. The first reference point is at the center of the arc this dimension is drawn for, while the second point is somewhere on the arc itself. If the alignment point is between the two reference points, the dimension line is drawn between the reference points, which is "inside" the arc. Otherwise the dimension line is drawn "outside" of the arc. If the measurement text is too long to fit on an inside radius dimension, the dimension line is drawn on the outside. A radius dimension automatically displays a cross at its first reference point (which is the center of the arc). When a radius dimension object is newly created or modified, the alignment point is normalized, so that it lies in the middle of the dimension line for an "inside" dimension, or just beyond the arrow for an "outside" dimension.

Diameter

A *diameter* dimension displays the distance between its first and second reference point. The two reference points are on opposite sides of the circle's circumference, so their distance measures the circle's diameter. If the alignment point is between the two reference points, the dimension line is drawn between the reference points, which is "inside" the circle. Otherwise the dimension line is drawn "outside" of the circle, much like a *parallel* dimension. If the measurement text is too long to fit on an inside diameter dimension, the dimension line is drawn on the outside. A diameter dimension automatically displays a cross in the middle between its two reference points (which is the center of the circle). When a diameter dimension object is newly created or modified, the alignment point is normalized, so that it lies at the same coordinates as its second reference point for an "inside" dimension, or in the middle of the dimension line for an "outside" dimension.

Angle

An *angle* dimension displays the angle between the second and third reference point, measured counterclockwise around the first reference point (which is the center of the arc). When an angle dimension object is newly created or modified, the second reference point is normalized, so that it has the same distance from the first point as the third one does.

Leader

A *leader* dimension can be used to point at something in a drawing. There is an arrow at the first point, and the second and third point define a (bent) line. The leader doesn't display any measurement. You can use the TEXT command to place any text you need.

Selection

A dimension object can be selected at any of its three points.

DISPLAY

Function

Selects the visible layers.

Syntax

```
DISPLAY
DISPLAY [option] layer_number..
DISPLAY [option] layer_name..
```

See also LAYER, PRINT

Valid options are: ALL, NONE, LAST, ? and ??

The DISPLAY command is used to choose the visible layers. As parameters, the layer number and the layer name are allowed (even mixed). If the parameter ALL is chosen, all layers become visible. If the parameter NONE is used, all layers are switched off. For example:

```
DISPLAY NONE BOTTOM;
```

Following this command only the Bottom layer is displayed.

If the parameter LAST is given, the previously visible layers will be displayed.

Please note that only those signal layers (1 through 16) are available that have been entered into the layer setup in the Design Rules.

If the layer name or the layer number includes a negative sign, it will be filtered out. For example:

```
DISPLAY TOP -BOTTOM -3;
```

In this case the Top layer is displayed while the Bottom layer and the layer with the number 3 are not shown on the screen.

Avoid layer names ALL and NONE as well as names starting with a "-".

Some commands (PAD, SMD, SIGNAL, ROUTE) automatically activate certain layers.

If the DISPLAY command is invoked without parameters, a dialog is presented which allows you to adjust all layer settings.

Undefined Layers

The options '?' and '??' can be used to control what happens if an undefined layer is given in a DISPLAY command. Any undefined layers following a '?' will cause a warning and the user can either accept it or cancel the entire DISPLAY command. Undefined layers following a '??' will be silently ignored. This is most useful for writing script files that shall be able to handle any drawing, even if a particular drawing doesn't contain some of the listed layers.

```
DISPLAY TOP BOTTOM ? MYLAYER1 MYLAYER2 ?? OTHER WHATEVER
```

In the above example the two layers TOP and BOTTOM are required and will cause an error if either of them is missing. MYLAYER1 and MYLAYER2 will just be reported if missing, allowing the user to cancel the operation, and OTHER and WHATEVER will be displayed if they are there, otherwise they will be ignored.

The '?' and '??' options may appear any number of times and in any sequence.

Pads and Vias

If pads or vias have different shapes on different layers, the shapes of the currently visible (activated with DISPLAY) signal layers are displayed on top of each other.

If the color selected for layer 17 (Pads) or 18 (Vias) is 0 (which represents the current background color), the pads and vias are displayed in the color and fill style of the respective signal layers. If no signal layer is visible, pads and vias are not displayed.

If the color selected for layer 17 (Pads) or 18 (Vias) is not the background color and no signal layers are visible, pads and vias are displayed in the shape of the uppermost and undermost layer.

This also applies to printouts made with PRINT.

Selecting Objects

If you want to select certain objects or elements (e.g. with MOVE or DELETE) the corresponding layer must be visible. Elements can only be selected if the tOrigins (or bOrigins with mirrored elements) layer is visible!

Parameter Aliases

Parameter aliases can be used to define certain parameter settings to the DISPLAY command, which can later be referenced by a given name. The aliases can also be accessed by clicking on the DISPLAY button and holding the mouse button pressed until the list pops up. A right click on the button also pops up the list.

The syntax to handle these aliases is:

`DISPLAY = name parameters`
: Defines the alias with the given *name* to expand to the given *parameters*. The *name* may consist of any number of letters, digits and underlines, and is treated case insensitive. It must begin with a letter or underline and may not be one of the option keywords.

`DISPLAY = name @`
: Defines the alias with the given *name* to expand to the current parameter settings of the command.

`DISPLAY = ?`
: Asks the user to enter a name for defining an alias for the current parameter settings of the command.

`DISPLAY = name`
: Opens the DISPLAY dialog and allows the user to select a set of layers that will be defined as an alias under the given *name*.

`DISPLAY = name;`
: Deletes the alias with the given *name*.

`DISPLAY name`
: Expands the alias with the given *name* and executes the DISPLAY command with the resulting set of parameters. The *name* may be abbreviated and there may be other parameters before and after the alias (even other aliases). Note that in case *name* is an abbreviation, aliases have precedence over other parameter names of the command.

Example:

```
DISPLAY = MyLayers None Top Bottom Pads Vias Unrouted
```

Defines the alias "MyLayers" which, when used as in

```
DISPLAY myl
```

will display just the layers Top, Bottom, Pads, Vias and Unrouted (without the "None" parameter the given layers would be displayed in addition to the currently visible layers). Note the abbreviated use of the alias and the case insensitivity.

DRC

Function

Checks design rules.

Syntax

```
DRC
DRC ▯ ▯ ;
DRC LOAD|SAVE filename;
DRC *
```

See also Design Rules, CLASS, SET, ERC, ERRORS

The command DRC checks a board against the current set of Design Rules.

Please note that electrically irrelevant objects (wires in packages, rectangles, circles and texts) are not checked against each other for clearance errors.

The errors found are displayed as error polygons in the respective layers, and can be browsed through with the ERRORS command.

Without parameters the DRC command opens a Design Rules dialog in which the board's Design Rules can be defined, and from which the actual check can be started.

If two coordinates are given in the DRC command (or if the Select button is clicked in the Design Rules dialog) all checks will be performed solely in the defined rectangle. Only errors that occur (at least partly) in this area will be reported.

If you get DRC errors that don't go away, even if you modify the Design Rules, make sure you check the Net class of the reported object to see whether the error is caused by a specific parameter of that class.

To delete all error polygons use the command

```
ERRORS CLEAR
```

The `LOAD` and `SAVE` options can be used to load the Design Rules from or save them to the given file. If *filename* doesn't have the extension `".dru"` it will be appended automatically.

If the DRC command is given an asterisk character (`'*'`) as the first parameter, the Design Rules dialog will be opened and allow editing the Design Rules, without triggering an actual check when the dialog is confirmed.

Related SET commands

The SET command can be used to change the behavior of the DRC command:

```
SET DRC_FILL  fill_name;
```

Defines the fill style used for the DRC error polygons. Default is LtSlash.

EDIT

Function

Loads an existing drawing to be edited or creates a new drawing.

Syntax

```
EDIT name
EDIT name.ext
EDIT .ext
EDIT .sX [ .sY ]
```

See also OPEN, CLOSE, BOARD

The EDIT command is used to load a drawing or if a library has been opened with the OPEN command, to load a package, symbol, or device for editing.

`EDIT name.brd`	loads a board
`EDIT name.sch`	loads a schematic
`EDIT name.pac`	loads a package
`EDIT name.sym`	loads a symbol
`EDIT name.dev`	loads a device
`EDIT .s3`	loads sheet 3 of a schematic
`EDIT .s5 .s2`	moves sheet 5 before sheet 2 and loads it (if sheet 5 doesn't exist, a new sheet

	is inserted before sheet 2)
`EDIT .s2 .s5`	moves sheet 2 before sheet 5 and loads it (if sheet 5 doesn't exist, sheet 2 becomes the last sheet)

Wildcards in the name are allowed (e.g. *.brd).

The EDIT command without parameters will cause a file dialog (in board or schematic mode) or a popup menu (in library mode) to appear from which you can select the file or object.

To change from schematic to a board with the same name the command

```
EDIT .brd
```

can be used. In the same way to change from board to schematic use the command

```
EDIT .sch
```

To edit another sheet of a schematic the command

```
EDIT .sX
```

(X is the sheet number) or the combo box in the action toolbar of the editor window can be used. If the given sheet number doesn't exist, a new sheet is created.

You can also switch between sheets by clicking on an icon of the sheet thumbnail preview. Drag&drop in the thumbnail preview allows you to reorder sheets.

Symbols, devices or packages may only be edited if a library is first opened with the OPEN command.

Which Directory?

EDIT loads files from the project directory.

ERC

Function
   Electrical Rule Check.
Syntax
   `ERC`

See also DRC, ERRORS, Consistency Check

This command is used to test schematics for electrical errors. The result of the check is presented in the ERRORS dialog.

Consistency Check

The ERC command also performs a Consistency Check between a schematic and its corresponding board, provided the board file has been loaded before starting the ERC. As a result of this check the automatic Forward&Back Annotation will be turned on or off, depending on whether the files have been found to be consistent or not.

Please note that the ERC detects inconsistencies between the implicit power and supply pins in the schematic and the actual signal connections in the board. Such inconsistencies can

occur if the supply pin configuration is modified after the board has been created with the BOARD command. Since the power pins are only connected "implicitly", these changes can't always be forward annotated.
If such errors are detected, Forward&Back Annotation will still be performed, but the supply pin configuration should be checked!

ERRORS

Function

Shows the errors found by the ERC or DRC command.

Syntax

```
ERRORS
ERRORS CLEAR
```

See also ERC, DRC

The command ERRORS is used to show the errors found by the Electrical Rule Check (ERC) or the Design Rule Check (DRC). If selected, a window is opened in which all errors are listed. If no ERC or DRC has been run for the loaded drawing, yet, the respective check will be started first.

The list view in the ERRORS dialog has up to four sections that contain *Consistency errors*, *Errors*, *Warnings* and *Approved* messages, respectively.

Selecting an entry with the mouse causes the error to be marked in the editor window with a rectangle and a line from the upper left corner of the screen.

Double clicking an entry centers the drawing to the area where the error is located. Checking the "Centered" checkbox causes this to happen automatically.

Marking a message as processed

The *Processed* button marks a message as processed. It is still contained in the list, but there is no error indicator in the editor window any more (except if the list entry is selected). This can be used to mark messages as "done" after fixing the related problem, without having to run the check again. After the next ERC/DRC the message will be either gone, or marked as unprocessed again if the problem still persists.

Approving a message

If an error or warning can't be fixed, but apparently doesn't matter (which the user has to decide), it can be moved to the *Approved* section by pressing the *Approve* button. Messages in that section will not draw error indicators in the editor window (except if the list entry is selected) and are implicitly marked as "processed". If any of these messages no longer apply after the next ERC/DRC, they will be deleted. All approved messages are stored in the drawing file, so that it is documented which ones have been explicitly approved by the user. Note that consistency errors can not be approved - they always have to be fixed in order to activate Forward&Back Annotation.

Clearing the list

The *Clear all* button deletes all entries form the list, except for the approved messages. This can be used to get rid of the error indicators in the editor window. The next ERC/DRC will regenerate the messages again, if they still apply.

The list can also be cleared by entering the command

```
ERRORS CLEAR
```

EXPORT

Function

Generation of data files.

Syntax

```
EXPORT SCRIPT filename;
EXPORT NETLIST filename;
EXPORT NETSCRIPT filename;
EXPORT PARTLIST filename;
EXPORT PINLIST filename;
EXPORT DIRECTORY filename;
EXPORT IMAGE filename|CLIPBOARD [MONOCHROME|WINDOW]
resolution;
```

See also SCRIPT, RUN

The EXPORT command is used to provide you with ASCII text files which can be used e.g. to transfer data from EAGLE to other programs, or to generate an image file from the current drawing.

By default the output file is written into the **Project** directory.

The command generates the following output files:

SCRIPT

A library previously opened with the OPEN command will be output as a script file. When a library has been exported and is to be imported again with the SCRIPT command, a new library should be opened in order to avoid duplication - e.g. the same symbol is defined more than once. Reading script files can be accelerated if the command

```
Set Undo_Log Off;
```

is given before.

NETLIST

Generates a netlist for the loaded schematic or board. Only nets which are connected to elements are listed.

NETSCRIPT

Generates a netlist for the loaded schematic in the form of a script file. This file can be used to read a new or changed netlist into a board where elements have already been placed or previously routed tracks have been deleted with `DELETE SIGNALS`. Note that while reading such a script into a board no schematic that is consistent with this board may be loaded.

PARTLIST

Generates a component list for schematics or boards. Only elements with pins/pads are included.

PINLIST

Generates a list with pads and pins, containing the pin directions and the names of the nets connected to the pins.

DIRECTORY

Lists the directory of the currently opened library.

IMAGE

Exporting an *IMAGE* generates an image file with a format corresponding to the given filename extension. The following image formats are available:

`.bmp`	Windows Bitmap Files
`.png`	Portable Network Graphics Files
`.pbm`	Portable Bitmap Files
`.pgm`	Portable Grayscale Bitmap Files
`.ppm`	Portable Pixelmap Files
`.tif`	TIFF Files
`.xbm`	X Bitmap Files
`.xpm`	X Pixmap Files

The *resolution* parameter defines the image resolution (in 'dpi').

If *filename* is the special name CLIPBOARD (upper or lowercase doesn't matter) the image will be copied into the system's clipboard.

The optional keyword *MONOCHROME* creates a black&white image.

The optional keyword *WINDOW* creates an image of the currently visible area in the editor window. Without this keyword, the image will contain the entire drawing.

Further formats

A lot of further formats like DXF or Hyperlynx can be exported by ULPs. They can be started from command line using the RUN command. Under 'File/Export' a number these format exports are also available.

FRAME

Function

Adds a frame to a drawing.

Syntax

`FRAME [ columns [ rows ] ] [ borders ] ▯ ▯`

Mouse keys

Center selects the layer.

See also LABEL

The FRAME command draws a frame with numbered columns and rows. The two points define two opposite corners of the frame. Pressing the center mouse button changes the layer to which the frame is to be added.

The `columns` parameter defines the number of columns in the frame. There can be up to 127 columns. By default the columns are numbered from left to right. If the `columns` value is negative, they are numbered from right to left.

The `rows` parameter defines the number of rows in the frame. There can be up to 26 rows. Rows are marked from top to bottom with letters, beginning with 'A'. If the `rows` value is negative, they are marked from bottom to top. If `rows` is given, it must be preceeded by `columns`.

The `borders` parameter, if given, defines which sides of the frame will have a border with numbers or letters displayed. Valid options for this parameter are `Left`, `Top`, `Right` and `Bottom`. By default all four sides of the frame will have a border. If any of these options is given, only the requested sides will have a border. The special options `None` and `All` can be used to have no borders at all, or all sides marked.

Even though you can draw several frames in the same drawing, only the first one will be used for calculating the positions of parts and nets. These positions can be used, for instance, in a User Language Program to generate a list of parts with their locations in their respective frame. They are also used internally to automatically generate cross references for labels.

Due to the special nature of the frame object, it doesn't have a rotation of its own, and it doesn't get rotated with the ROTATE command.

A frame can be drawn directly into a board or schematic, but more typically you will want to create a special symbol or package drawing that perhaps also contains a title block, which you can then use in all your drawings. The "frames" library that comes with EAGLE contains several drawing frames.

Example

```
FRAME 10 5 TOP LEFT ▯ ▯
```

draws a frame with 10 columns (numbered from left to right) and 5 rows (marked 'A' to 'E' from top to bottom) that has the column and row indicators drawn only at the top and left border.

GATESWAP

Function

Swaps equivalent gates on a schematic.

Syntax
```
    GATESWAP □ □..;
    GATESWAP gate_name gate_name..;
```

See also ADD

Using this command two gates may be swapped within a schematic. Both gates must be identical with the same number of pins and must be allocated the same Swaplevel in the device definition. They do not, however, need to be in the same device.

The name used in the GATESWAP command is the displayed name on the schematic (e.g. U1A for gate A in device U1).

If a device is not used anymore after the GATESWAP command, it is deleted automatically from the drawing.

GRID

Function

Defines grid.

Syntax
```
    GRID option..;
    GRID;
```

Keyboard

`F6: GRID;` turns the grid on or off.

See also SCRIPT

The GRID command is used to specify the grid and the current unit. Given without an option, this command switches between GRID ON and GRID OFF.

The following options exist:

`GRID ON;`	Displays the grid on the screen
`GRID OFF;`	Turns off displayed grid
`GRID DOTS;`	Displays the grid as dots
`GRID LINES;`	Displays the grid as solid lines
`GRID MIC;`	Sets the grid units to micron
`GRID MM;`	Sets the grid units to mm
`GRID MIL;`	Sets the grid units to mil
`GRID INCH;`	Sets the grid units to inch
`GRID FINEST;`	Sets the grid to the finest possible value
`GRID grid_size;`	Defines the distance between the grid points in the actual unit
`GRID LAST;`	Sets grid to the most recently used values
`GRID DEFAULT;`	Sets grid to the standard values

`GRID grid_size grid_multiple;`	grid_size = grid distance
	grid_multiple = grid factor

`GRID ALT ...;`	Defines the alternate grid

Examples

```
Grid mm;
Set Diameter_Menu 1.0 1.27 2.54 5.08;
Grid Last;
```

In this case you can change back to the last grid definition although you don't know what the definition looked like.

```
GRID mm 1 10;
```

for instance specifies that the distance between the grid points is 1 mm and that every 10th grid line will be displayed.

Note: The first number in the GRID command always represents the grid distance, the second - if existing - represents the grid multiple.

The GRID command may contain multiple parameters:

```
GRID inch 0.05 mm;
```

In this case the grid distance is first defined as 0.05 inch. Then the coordinates of the cursor are chosen to be displayed in mm.

```
GRID DEFAULT;
```

Sets grid to the standard value for the current drawing type.

```
GRID mil 50 2 lines on alt mm 1 mil;
```

Defines a 50 mil grid displayed as lines (with only every other line visible), and sets the alternate grid size to 1 mm, but displays it in mil.

Pressing the `Alt` key switches to the alternate Grid. This can typically be a finer grid than the normal one, which allows you to quickly do some fine positioning in a dense area, for instance, where the normal grid might be too coarse. The alternate grid remains active as long as the `Alt` key is held pressed down.

Parameter Aliases

Parameter aliases can be used to define certain parameter settings to the GRID command, which can later be referenced by a given name. The aliases can also be accessed by clicking on the GRID button and holding the mouse button pressed until the list pops up. A right click on the button also pops up the list.

The syntax to handle these aliases is:

`GRID = name parameters`
: Defines the alias with the given *name* to expand to the given *parameters*. The *name* may consist of any number of letters, digits and underlines, and is treated case insensitive. It must begin with a letter or underline and may not be one of the option keywords.

`GRID = name @`
: Defines the alias with the given *name* to expand to the current parameter settings of the command.

`GRID = ?`
: Asks the user to enter a name for defining an alias for the current parameter settings of the command.

`GRID = `*`name`*

Opens the GRID dialog and allows the user to adjust the grid parameters and define an alias for them under the given *name*.

`GRID = `*`name`*`;`

Deletes the alias with the given *name*.

`GRID `*`name`*

Expands the alias with the given *name* and executes the GRID command with the resulting set of parameters. The *name* may be abbreviated and there may be other parameters before and after the alias (even other aliases). Note that in case *name* is an abbreviation, aliases have precedence over other parameter names of the command.

Example:

```
GRID = MyGrid inch 0.1 lines on
```

Defines the alias "MyGrid" which, when used as in

```
GRID myg
```

will change the current grid to the given settings. Note the abbreviated use of the alias and the case insensitivity.

GROUP

Function

Defines a group.

Syntax

```
GROUP ▯..
GROUP ALL
GROUP;
```

Mouse keys

Left&Drag defines a rectangular group.
Shift+Left adds the new group to an existing one.
Ctrl+Left toggles the group membership of the selected object.
Ctrl+Shift+Left toggles the group membership of the higher level object.
Right closes the group polygon.

See also CHANGE, CUT, PASTE, MIRROR, DELETE

The GROUP command is used to define a group of objects for a successive command. Also a whole drawing or an element can be defined as a group. Objects are selected - after activating the GROUP command - by click&dragging a rectangle or by drawing a polygon with the mouse. The easiest way to close the polygon is to use the right mouse button. Only objects from displayed layers can become part of the group.

The keyword `ALL` can be used to define a group that includes the entire drawing area.

The group includes:

- all objects whose origin is inside the polygon
- all wires with at least one end point inside the polygon
- all circles whose center is inside the polygon
- all rectangles with any corner inside the polygon

Move Group

In order to move a group it is necessary to select the MOVE command with the right mouse button. When moving wires (tracks) with the GROUP command that have only one end point in the polygon, this point is moved while the other one remains at its previous position.

For instance: In order to change several pad shapes, select CHANGE and SHAPE with the left mouse button and select the group with the right mouse button.

The group definition remains until a new drawing is loaded or the command

```
GROUP;
```

is executed.

Extending the group

If you press the `Shift` key together with any mouse click when defining the group, the newly defined group will be added to the existing group (if any).

Individual objects

You can toggle the group membership of an individual object by clicking on it with the `Ctrl` key pressed. If you also press the `Shift` key when doing so, the group membership of the next higher level object is toggled. For instance, when clicking on a net wire in a schematic with the GROUP command and `Ctrl+Shift` pressed, the group membership of the entire segment will be toggled.

HELP

Function

Help for the current command.

Syntax

```
HELP
HELP command
```

Keyboard

`F1: HELP` activates the context sensitive help.

This command opens a context sensitive help window.

A `command` name within the HELP command shows the help page of that command.

Example

```
HELP GRID;
```

displays the help page for the GRID command.

HOLE

Function

Add drill hole to a board or package.

Syntax

```
HOLE drill ▯..
```

See also VIA, PAD, CHANGE

This command is used to define e.g. mounting holes (has no electrical connection between the different layers) in a board or in a package. The parameter drill defines the diameter of the hole in the actual unit. It may be up to 0.51602 inch (13.1 mm).

Example

```
HOLE 0.20 ▯
```

If the actual unit is "inch", the hole will have a diameter of 0.20 inch.

The entered value for the diameter (also used for via-holes and pads) remains as a presetting for successive operations. It may be changed with the command:

```
CHANGE DRILL value ▯
```

A hole can only be selected if the Holes layer is displayed.

A hole generates a symbol in the Holes layer as well as a circle with the diameter of the hole in the Dimension layer. The relation between certain diameters and symbols is defined in the "Options/Set/Drill" dialog. The circle in the Dimension layer is used by the Autorouter. As it will keep a (user-defined) minimum distance between via-holes/wires and dimension lines, it will automatically keep this distance to the hole.

In the layers tStop and bStop, holes generate the solder stop mask, whose diameter is determined by the Design Rules.

INFO

Function

Display and modify object properties.

Syntax

```
INFO ▯..
INFO name ..
```

See also CHANGE, SHOW

The INFO command displays further details about an object's properties on screen, e.g. wire width, layer number, text size etc. It is also possible to modify properties in this dialog.

Parts, pads, smds, pins and gates can also be selected by their name, which is especially useful if the object is outside the currently shown window area. Note that when selecting a multi-gate part in a schematic by name, you will need to enter the full instance name, consisting of part and gate name.

Attributes of parts can be selected by entering the concatenation of part name and attribute name, as in `R5>VALUE`.

INVOKE

Function

Call a specific symbol from a device.

Syntax

```
INVOKE □ orientation □
INVOKE part_name gate_name orientation □
```

Mouse keys

Center mirrors the gate.
Right rotates the gate.
Shift+Right reverses the direction of rotating.

See also COPY, ADD

See the ADD command for an explanation of Addlevel und Orientation.

The INVOKE command is used to select a particular gate from a device which is already in use and place it in the schematic (e.g. a power symbol with Addlevel = Request).

Gates are activated in the following way:

- Enter the part name (e.g. IC5) and select the gate from the popup dialog that appears.
- Define device and gate name from the keyboard (e.g. INVOKE IC5 POWER).
- Select an existing gate from the device with the mouse and then select the desired gate from the popup menu which appears.

The final mouse click positions the new gate.

If an already invoked gate is selected in the dialog, the default button changes to "Show", and a click on it zooms the editor window in on the selected gate, switching to a different sheet if necessary.

Gates on Different Sheets

If a gate from a device on a different sheet is to be added to the current sheet, the name of the part has to be specified in the INVOKE command. In this case the right column of the popup menu shows the sheet numbers where the already used gates are placed.

JUNCTION

Function

Places a dot at intersecting nets.

Syntax

```
JUNCTION □..
```

See also NET

This command is used to draw a connection dot at the intersection of nets which are to be connected to each other. Junction points may be placed only on a net. If placed on the intersection of different nets, the user is given the option to connect the nets.

If a net wire is placed at a point where there are at least two other net wires and/or pins, a junction will automatically be placed. This function can be disabled with "`SET AUTO_JUNCTION OFF;`", or by unchecking "Options/Set/Misc/Auto set junction".

On the screen junction points are displayed at least with a diameter of five pixels.

LABEL

Function

Attaches text labels to buses and nets.

Syntax

```
LABEL [XREF] [orientation] ▯ ▯..
```

Mouse keys

Center selects the layer.
Right rotates the label.
Shift+Right reverses the direction of rotating.

See also NAME, BUS, FRAME

Bus or net names may be placed on a schematic in any location by using the label command. When the bus or net is clicked on with the mouse, the relevant label attaches to the mouse cursor and may be rotated, changed to another layer, or moved to a different location. The second mouse click defines the location of the label.

The orientation of the label may be defined textually using the usual definitions as listed in the ADD command (R0, R90 etc.).

Buses and nets may have any number of labels.

Labels cannot be changed with "CHANGE TEXT".

Labels are handled by the program as text, but their value corresponds to the name of the appropriate bus or net. If a bus or net is renamed with the NAME command, all associated labels are renamed automatically.

If a bus, net, or label is selected with the SHOW command, all connected buses, nets and labels are highlighted.

Cross-reference labels

If the optional keyword `XREF` is given, the label will be a "cross-reference" label. Cross-reference labels can be used in multi-sheet schematics to indicate the next sheet a particular net appears on (note that this only works for nets, not for busses!). The `XREF` keyword is mainly for use in scripts. Normally the setting is taken from what has previously been set with CHANGE XREF, or by clicking on the Xref button in the parameter toolbar.

The format in which a cross-reference label is displayed can be controlled through the "Xref label format" string, which is defined in the "Options/Set/Misc" dialog, or with the SET command. The following placeholders are defined, and can be used in any order:

`%F`	enables drawing a flag border around the label
`%N`	the name of the net
`%S`	the next sheet number
`%C`	the column on the next sheet
`%R`	the row on the next sheet

The default format string is `"%F%N/%S.%C%R"`. Apart from the defined placeholders you can also use any other ASCII characters.

The column and row values only work if there is a frame on the next sheet on which the net appears. If `%C` or `%R` is used and there is no frame on that sheet, they will display a question mark (`'?'`).

When determining the column and row of a net on a sheet, first the column and then the row within that column is taken into account. Here XREF labels take precedence over normal labels, which again take precedence over net wires. For a higher sheet number, the frame coordinates of the left- and topmost field are taken, while for a lower sheet number those of the right- and bottommost field are used.

The orientation of a cross-reference label defines whether it will point to a "higher" or a "lower" sheet number. Labels with an orientation of R0 or R270 point to the right or bottom border of the drawing, and will therefore refer to a higher sheet number. Accordingly, labels with an orientation of R90 or R180 will refer to a lower sheet number. If a label has an orientation of R0 or R270, but the net it is attached to is not present on any higher sheet, a reference to the next lower sheet is displayed instead (the same applies accordingly to R90 and R180). If the net appears only on the current sheet, no cross-reference is shown at all, and only the net name is displayed (surrounded by the flag border, if the format string contains the `%F` placeholder).

A cross-reference label that is placed on the end of a net wire will connect to the wire so that the wire is moved with the label, and vice versa.

The cross-reference label format string is stored within the schematic drawing file.

A cross-reference label can be changed to a normal label either through the CHANGE command or the label's *Properties* dialog.

Selecting the layer

Unlike other commands (like WIRE, for instance), the LABEL command keeps track of its last used layer by itself. This has the advantage of making sure that labels are always drawn into the right layer, no matter what layers other commands draw into. The downside of this is that the usual way of setting the layer in a script, as in

```
LAYER layer;
WIRE (1 2) (3 4);
```

doesn't work here. The layer needs to be selected while the LABEL command is already active, which can be done like this

```
LABEL parameters
LAYER layer
more parameters;
```

Note that the LABEL line is **not** terminated with a `';'`, and that the LAYER command starts on a new line.
The commands

```
LABEL
LAYER layer;
```

set the layer to use with subsequent LABEL commands.

LAYER

Function

Changes and defines layers.

Syntax

```
LAYER layer_number
LAYER layer_name
LAYER layer_number layer_name
LAYER [??] -layer_number
```

See also DISPLAY

Choose Drawing Layer

The LAYER command with one parameter is used to change the current layer, i.e. the layer onto which wires, circles etc. will be drawn. If LAYER is selected from the menu, a popup menu will appear in which you may change to the desired layer. If entered from the command line, 'layer_number' may be the number of any valid layer, and 'layer_name' may be the name of a layer as displayed in the popup menu.

Certain layers are not available in all modes.

Please note that only those signal layers (1 through 16) are available that have been entered into the layer setup in the Design Rules.

Define Layers

The LAYER command with two parameters is used to define a new layer or to rename an existing one. If you type in at the command prompt e.g.

```
LAYER 101 SAMPLE;
```

you define a new layer with layer number 101 and layer name SAMPLE.

If a package contains layers not yet specified in the board, these layers are added to the board as soon as you place the package into the board (ADD or REPLACE).

The predefined layers have a special function. You can change their names, but their functions (related with their number) remain the same.

If you define your own layers, you should use only numbers greater than 100. Numbers below may be assigned for special purposes in later EAGLE versions.

Delete Layers

The LAYER command with the minus sign and a layer_number deletes the layer with the specified number, e.g.

```
LAYER -103;
```

deletes the layer number 103. Layers to be deleted must be empty. If this is not the case, the program generates the error message

"layer is not empty: #"

where "#" represents the layer number. If you want to avoid any error messages in a layer delete operation you can use the '??' option. This may be useful in scripts that try to delete certain layers, but don't consider it an error if any of these layers is not empty or not present at all.

Predefined EAGLE Layers

Layout

1 Top	Tracks, top side
2 Route2	Inner layer
3 Route3	Inner layer
4 Route4	Inner layer
5 Route5	Inner layer
6 Route6	Inner layer
7 Route7	Inner layer
8 Route8	Inner layer
9 Route9	Inner layer
10 Route10	Inner layer
11 Route11	Inner layer
12 Route12	Inner layer
13 Route13	Inner layer
14 Route14	Inner layer
15 Route15	Inner layer
16 Bottom	Tracks, bottom side
17 Pads	Pads (through-hole)
18 Vias	Vias (through-hole)
19 Unrouted	Airwires (rubberbands)
20 Dimension	Board outlines (circles for holes)
21 tPlace	Silk screen, top side
22 bPlace	Silk screen, bottom side
23 tOrigins	Origins, top side
24 bOrigins	Origins, bottom side
25 tNames	Service print, top side
26 bNames	Service print, bottom side
27 tValues	Component VALUE, top side
28 bValues	Component VALUE, bottom side
29 tStop	Solder stop mask, top side
30 bStop	Solder stop mask, bottom side
31 tCream	Solder cream, top side
32 bCream	Solder cream, bottom side
33 tFinish	Finish, top side
34 bFinish	Finish, bottom side
35 tGlue	Glue mask, top side
36 bGlue	Glue mask, bottom side
37 tTest	Test and adjustment inf., top side
38 bTest	Test and adjustment inf. bottom side
39 tKeepout	Nogo areas for components, top side
40 bKeepout	Nogo areas for components, bottom side
41 tRestrict	Nogo areas for tracks, top side

42 bRestrict	Nogo areas for tracks, bottom side
43 vRestrict	Nogo areas for via-holes
44 Drills	Conducting through-holes
45 Holes	Non-conducting holes
46 Milling	Milling
47 Measures	Measures
48 Document	General documentation
49 Reference	Reference marks
51 tDocu	Part documentation, top side
52 bDocu	Part documentation, bottom side

Schematic

91 Nets	Nets
92 Busses	Buses
93 Pins	Connection points for component symbols with additional information
94 Symbols	Shapes of component symbols
95 Names	Names of component symbols
96 Values	Values/component types
97 Info	General information
98 Guide	Guide lines

LOCK

Function

Locks the position and orientation of a part in the board.

Syntax

```
LOCK ▯..
LOCK name ..
```

Mouse keys

Ctrl+Right applies the command to the group.
Shift+Left reverses the lock operation ("unlocks" the part).
Ctrl+Shift+Right "unlocks" all parts in the group.

See also MIRROR, MOVE, ROTATE SMASH

The LOCK command can be applied to parts in a board, and prevents them from being moved, rotated, or mirrored. This is useful for things like connectors, which need to be mounted at a particular location and must not be inadvertently moved.

The origin of a locked part is displayed as an 'x' to have a visual indication that the part is locked.

If a group is moved and it contains locked parts, these parts (together with any wires ending at their pads) will not move with the group.

Detached texts of a locked part can still be moved individually, but they won't move with a group.

Parts can also be selected by their name, which is especially useful if the object is outside the currently shown window area.

A "locked" part can be made "unlocked" by clicking on it with the `Shift` key pressed (and of course the LOCK command activated).

MARK

Function

Defines a mark on the drawing area.

Syntax

```
MARK ▯
MARK;
```

See also GRID

The MARK command allows you to define a point on the drawing area and display the coordinates of the mouse cursor relative to that point at the upper left corner of the screen (with a leading 'R' character). This command is useful especially when board dimensions or cutouts are to be defined. Entering MARK; turns the mark on or off.

Please choose a grid fine enough before using the MARK command.

MEANDER

Function

Balance lengths of differential pairs and increase the length of a signal segment.

Syntax

```
MEANDER [length] ▯ ..
```

Mouse keys

Ctrl+Left measures the length of the selected signal segment.
Ctrl+Shift+Left measures the maximum length of the selected signal segments.
Right toggles between symmetrical and asymmetrical meanders.

See also ROUTE

The MEANDER command can be used to balance the lengths of signals forming a differential pair. To do this, just click on a wire of a differential pair and move the mouse cursor away from the selection point. If there is a difference in the length of the two signals, and the current mouse position is far enough away from the selection point, a "meander" shaped sequence of wires will be drawn that increases the length of the shorter signal segment. An indicator attached to the mouse cursor shows the target length (which is the length of the longer signal segment), as well as the deviation (in percent) of the two signals from the target length.

The meander starts at the first click point and extends to the second point the mouse is moved to. The maximum (perpendicular) size of the meander is defined by the distance of the mouse to the meandered wire.

If a single meander isn't enough to balance the lengths, you can add further meanders at different locations.

Measuring signal lengths

If you click on a signal wire with the `Ctrl` key pressed, the length of that signal segment will be measured and displayed on the screen in a little indicator near the mouse position.

You can use this to measure the length of a given signal segment and use that as the target length for meandering an other segment.

If you do the measuring with `Ctrl+Shift` pressed, the maximum length of this or any previously selected segments will be taken. This can be used to easily determine the maximum length of several bus signals and then meandering each of them to that length.

At any time you can enter a value in the command line to set the target length.

When meandering a differential pair with a given target length, the meander first tries to balance the length of the two signal segments that form the differential pair, and then increases the total length of both segments.

To reset the target length you can either restart the MEANDER command or enter a value of `0` in the command line.

Symmetrical and asymmetrical meanders

By default a meander is generated symmetrical, which means it extends to both sides along the selected wire. If this is not what you need (either because there is only space on one side, or because the longer one of the wires of a differential pair shall not be elongated) you can switch to asymmetric mode by clicking the right mouse button. The actual mouse position will decide which side of the wire the meander extends to. Move the mouse around to find the proper position.

Length tolerance

The value defined in the Design Rules under "Misc/Max. length difference in differential pairs" is used to select the color when displaying the length deviations while drawing a meander. If the percentage is shown in green, the respective segment lies within the given tolerance. Otherwise the percentage is displayed in red. The default for this parameter is 10mm.

MENU

Function

Customizes the textual command menu.

Syntax

```
MENU option ..;
MENU;
```

See also ASSIGN, SCRIPT

The MENU command can be used to create a user specific command menu.

The complete syntax specification for the `option` parameters is

```
option    := command | submenu | delimiter
command   := [ icon ] text1 [ ':' text2 ]
submenu   := [ icon ] text '{' option [ '|' option ] '}'
icon      := '[' filename ']'
delimiter := '---'
```

A menu option can be a simple command, as in

```
MENU Display Grid;
```

which would set the menu to the commands `Display` and `Grid`. `Display` and `Grid` are interpreted both as menu text and as commands.
It can be an aliased command, as in

```
MENU 'MyDisp : Display None Top Bottom Pads Vias;' 'MyGrid : Grid mil 100 lines
on;';
```

which would set the menu to show the command aliases `MyDisp` and `MyGrid` and actually execute the command sequence behind the `':'` of each option (`text2`, see above) when the respective button is clicked.
It can also be a submenu button as in

```
MENU 'Grid { Fine : Grid inch 0.001; | Coarse : Grid inch 0.1; }';
```

which would define a button labeled `Grid` that, when clicked opens a submenu with the two options `Fine` and `Coarse`.
Character `'|'` is only necessary as a separator in submenu entries (`submenu`, see above).

The special option `'---'` can be used to insert a delimiter, which may be useful for grouping buttons.

A command button can display an icon by preceding the button text with the file name of an icon, enclosed in square brackets, as in

```
MENU '[/path/to/myicon.png] Set a fine grid : Grid inch 0.001;';
```

Here the button will display only the given icon, and "Set a fine grid" will be used as a "tool tip" that gets displayed when the mouse cursor is moved over the button. The filename needs not be enquoted additionally (like for masking spaces).
If an icon is used in a popup menu, like

```
MENU 'Grid { [/path/to/myicon.png] Set a fine grid : Grid inch 0.001; }';
```

then both the icon and the text will be displayed, as with any other popup menu.
If the icon's file name doesn't include a directory path, it is searched for in the current working directory and in the EAGLE 'bin' directory.

Note that any *option* that consists of more than a single word, or that might be interpreted as a command, must be enclosed in single quotes. If you want to use the MENU command in a script to define a complex menu, and would like to spread the menu definitions over several lines to make them more readable, you need to end the lines with a backslash character (`'\'`) as in

```
MENU 'Grid {\
             Fine : Grid inch 0.001; |\
             Coarse : Grid inch 0.1;\
           }';
```

Examples

```
MENU Move Delete Rotate Route ';' Edit;
```

would create a command menu that contains the commands Move...Route, the semicolon, and the Edit command.

The command

```
MENU;
```

switches back to the default menu. Note that the ';' entry should always be added to the menu. It is used to terminate many commands.

Complex example:

```
MENU '[draw.png] Draw {\
                        Wire {\
                               Continous : CHANGE STYLE Continuous; WIRE |\
                               DashDot :  CHANGE STYLE DashDot; WIRE |\
                               Help : HELP WIRE;\
                             }|\
                        Rectangle {\
                                    RECT |\
                                    Help : HELP RECT; \
                                  }\
                       }\
      [export.png] Export {\
                            Script : EXPORT SCRIPT |\
                            Image : EXPORT IMAGE\
                          }\
      MyScript : SCRIPT MyScript.scr;';
```

This menu consists of the 3 entries Draw, Export and MyScript, whereat Draw and Export have submenus and are supplied with icons. Draw consists of the submenus Wire and Rectangle, whereat Wire consists of the entries Continous, DashDot and Help and Rectangle consists of the entries RECT (text and command RECT) and Help.
The submenu of Export has the entries Script and Image.

MIRROR

Function

Mirrors objects and groups.

Syntax

```
MIRROR ▯..
MIRROR name..
```

Mouse keys

Ctrl+Right mirrors the group.

See also ROTATE, LOCK, TEXT

Using the MIRROR command, objects may be mirrored about the y axis. One application for this command is to mirror components to be placed on the reverse side of the board.

Parts, pads, smds and pins can also be selected by their name, which is especially useful if the object is outside the currently shown window area.

Attributes of parts can be selected by entering the concatenation of part name and attribute name, as in `R5>VALUE`.

Components can be mirrored only if the appropriate tOrigins/bOrigins layer is visible.

When packages are selected for use with the MIRROR command, connected wires on the outer layers are mirrored, too (beware of short circuits!).

Note that any objects on inner layers (2...15) don't change their layer when they are mirrored. The same applies to vias.

Parts cannot be mirrored if they are locked, or if any of their connected pads would extend outside the allowed area (in case you are using a limited edition of EAGLE).

Mirror a Group

In order to mirror a group of elements, the group is first defined with the GROUP command and polygon in the usual manner. The MIRROR command is then selected and the right mouse button is used to execute the change. The group will be mirrored about the vertical axis through the next grid point.

Wires, circles, pads and polygons may not be individually mirrored unless included in a group.

Mirror Texts

Text on the solder side of a pc board (Bottom and bPlace layers) is mirrored automatically so that it is readable when you look at the solder side of the board.

Mirrored text in a schematic will be printed on the other side of its origin point, but it will still remain normally readable.

MITER

Function

Miters wire joints.

Syntax

```
MITER [radius] ▯..
```

Mouse keys

Left&Drag dynamically modifies the miter.
Right toggles between round and straight mitering.

See also SPLIT, WIRE, ROUTE, POLYGON

The MITER command can be used to take the edge off a point where two wires join. The two existing wires need to be on the the same layer and must have the same width and wire style.

Mitering a point

If you select a point where exactly two straight wires join, an additional wire will be inserted between these two wires, according to the given *radius*. If you click&drag on such a point with the left mouse button, you can define the mitering wire dynamically.

Mitering a wire

If you select a wire (which may also be an arc) somewhere in the middle between its end points, and that wire is connected to exactly two other straight wires (one at each end), the selected wire will be "re-mitered" according to the given *radius*. If you click&drag on such a wire with the left mouse button, you can define the mitering wire dynamically.

Straight versus round mitering

If *radius* is positive, the inserted wire will be an arc with the given radius; if it is negative, a straight wire will be inserted (imagine the `'-'` sign as indicating "straight"). You can toggle between round and straight mitering by pressing the right mouse button.

Miter radius and wire bend style

The *radius* you give in the MITER command will be used in all other commands that draw wires in case the wire bend style is one of the 90 or 45 degree styles. If you have set round mitering, it will apply to both the 90 and 45 degree bend styles; in case of straight mitering only the 90 degree bend styles are affected.

MOVE

Function

Moves objects.

Syntax

```
MOVE □ □..
MOVE name □..
```

Mouse keys

Ctrl+Left selects an object at its origin or modifies it (see note).
Ctrl+Right selects the group.
Left&Drag immediately moves the object.
Ctrl+Right&Drag immediately moves the group.
Center mirrors the selected object or the group.
Right rotates the selected object or the group.
Shift+Right reverses the direction of rotating.

Keyboard

`F7: MOVE` activates the MOVE command.

See also GROUP, LOCK, RATSNEST

The MOVE command is used to move objects.

Parts, pads, smds, pins and gates can also be selected by their name, which is especially useful if the object is outside the currently shown window area. Note that when selecting a multi-gate part in a schematic by name, you will need to enter the full instance name, consisting of part and gate name.

Attributes of parts can be selected by entering the concatenation of part name and attribute name, as in `R5>VALUE`.

Elements can be moved only if the appropriate tOrigins/bOrigins layer is visible.

The MOVE command has no effect on layers that are not visible (refer to DISPLAY).

The ends of wires (tracks) that are connected to an element cannot be moved at this point.

When moving elements, connected wires (tracks) that belong to a signal are moved too (beware of short circuits!).

If an object is selected with the left mouse button and the button is not released, the object can be moved immediately ("click&drag"). The same applies to groups when using the right mouse button. In this mode, however, it is not possible to rotate or mirror the object while moving it.

Parts cannot be moved if they are locked, or if any of their connected pads would extend outside the allowed area (in case you are using a limited edition of EAGLE).

Move Wires

If, following a MOVE command, two wires from different signals are shorted together, they are maintained as separate signals and the error will be flagged by the DRC command.

Move Groups

In order to move a group, the selected objects are defined in the normal way (GROUP command and polygon) before selecting the MOVE command and clicking the group with the right mouse button. The entire group can now be moved and rotated with the right mouse button.

Hints for Schematics

If a supply pin (Direction Sup) is placed on a net, the pin name is allocated to this net.

Pins placed on each other are connected together.

If unconnected pins of an element are placed on nets or pins then they are connected with them.

If nets are moved over pins they are not connected with them.

Selecting objects at their origin

Normally a selected object remains within the grid it has been originally placed on. If you press `Ctrl` while selecting an object, the point where you have selected the object is pulled towards the cursor and snapped into the current grid.

If you select a *wire* somewhere in the middle (not at one of its end points) with `Ctrl` pressed, the end points stay fixed and you can bend the wire, which changes it into an arc. The same way the curvature of an arc (which is basically a wire) can be modified.

If you select a *rectangle* at one of its corners with `Ctrl` pressed, you can resize both the rectangle's width and height. Selecting an edge of the rectangle with `Ctrl` pressed lets you resize the rectangle's width or height, respectively. Selecting the rectangle at its center with `Ctrl` pressed pulls it towards the cursor and snaps it into the current grid.

If you select a *circle* at its circumference with `Ctrl` pressed, the center stays fixed and you can resize the circle's diameter. Selecting the center point this way pulls it towards the cursor and snaps it into the current grid.

Move part of a sheet to an other sheet

You can move part of a sheet to an other sheet of the same schematic without affecting the board (in case Forward&Back Annotation is active) by defining a GROUP that contains the objects you want to move, selecting that group with the MOVE command and then switching to the desired sheet, with the MOVE command still active and having the group attached to the cursor. In the new sheet the MOVE command will be active again and will have the previously defined group attached to the cursor. Now place the group as usual, and all the affected objects will be transferred from the original sheet to the current sheet. If the current sheet is the same as the original sheet, nothing happens.

Note that only wires that have both ends in the group will be transferred, and any part that is transferred takes all its electrical connections with it, even if a net wire attached to one of its pins is not transferred because its other end is not in the group. In case a pin in the new sheet has an electrical connection, but no other pin, wire or junction attached to it to make this visible, a junction will be automatically generated at this point.

This process can even be scripted. For instance

```
edit .s1
group (1 1) (1 2) (2 2) (2 1) (1 1)
move (> 0 0)
edit .s2
(0 0)
```

would switch to the first sheet, define a group, select that group with MOVE, switch to the second sheet and place the group. Note the final `(0 0)`, which are coordinates to the implicitly invoked MOVE command.

See the EDIT command if you want to just reorder the sheets.

NAME

Function

Displays and changes names.

Syntax

```
NAME ▯..
NAME new_name ▯
NAME old_name new_name
```

See also SHOW, SMASH, VALUE

The NAME command is used to display or edit the name of the selected object.

Parts, elements, pads, smds, pins and gates can also be selected by their name, which is especially useful if the object is outside the currently shown window area. Other object types (e.g. nets, busses, signals) have to be clicked first.

Library

When in library edit mode, the NAME command is used to display or edit the name of the selected pad, smd, pin or gate.

Automatic Naming

EAGLE generates names automatically: E$.. for elements, S$.. for signals, P$.. for pads, pins and smds. In general, it is convenient to substitute commonly used names (e.g. 1...14

for a 14-pin dual inline package) in place of these automatically generated names. Automatic naming of parts can be controlled with PREFIX.

Schematic

If nets or buses are to be renamed, the program has to distinguish between three cases because they can consist of several segments placed on different sheets. Thus a menu will ask the user:

This segment
Every segment on this sheet
All segments on all sheets

These questions appear in a popup menu if necessary and can be answered either by selecting the appropriate item with the mouse or by pressing the appropriate hot key (T, E, A).

Polygon

When renaming a signal polygon in a board, you can choose whether to rename only this polygon (and thus move it from one signal into another), or to give the entire signal a different name.

NET

Function

Draws nets on a schematic.

Syntax

```
NET [net_name] ▯ [curve | @radius] ▯..
```

Mouse keys

Right changes the wire bend style (see SET Wire_Bend).
Shift+Right reverses the direction of switching bend styles.
Ctrl+Right toggles between corresponding bend styles.

See also BUS, NAME, CLASS, SET

The net command is used to draw individual connections (nets) onto the Net layer of a schematic drawing. The first mouse click marks the starting point for the net, the second marks the end point of a segment. Two mouse clicks on the same point end the net.

If a net wire is placed at a point where there is already another net or bus wire or a pin, the current net wire will be ended at that point. This function can be disabled with "`SET AUTO_END_NET OFF;`", or by unchecking "Options/Set/Misc/Auto end net and bus".

If a net wire is placed at a point where there are at least two other net wires and/or pins, a junction will automatically be placed. This function can be disabled with "`SET AUTO_JUNCTION OFF;`", or by unchecking "Options/Set/Misc/Auto set junction".

If the *curve* or *@radius* parameter is given, an arc can be drawn as part of the net (see the detailed description in the WIRE command).

Select Bus Signal

If a net is started on a bus, a popup menu opens from which one of the bus signals can be selected. The net then is named correspondingly and becomes part of the same signal. If the bus includes several part buses, a further popup menu opens from which the relevant part bus can be selected.

Net Names

If the NET command is used with a net name then the net is named accordingly.

If no net name is included in the command line and the net is not started on a bus, then a name in the form of N$1 is automatically allocated to the net.

Nets or net segments that run over different sheets of a schematic and use the same net name are connected.

Net names should not contain a comma (`','`), because this is the delimiting character in busses.

Line Width

The width of the line drawn by the net command may be changed with the command:

```
SET NET_WIRE_WIDTH width;
```

(Default: 6 mil).

Inverted signals

The name of an inverted signal ("active low") can be displayed overlined if it is preceded with an exclamation mark (`'!'`), as in

```
  !RESET
```

which would result in

$\overline{\text{RESET}}$

You can find further details about this in the description of the TEXT command.

OPEN

Function

Opens a library for editing.

Syntax

```
OPEN library_name
```

See also CLOSE, USE, EDIT, SCRIPT

The OPEN command is used to open an existing library or create a new library. Once the library has been opened or created, an existing or new symbol, device, or package may be edited.

This command is mainly used in script files.

OPTIMIZE

Function

Joins wire segments.

Syntax

```
OPTIMIZE;
OPTIMIZE signal_name ..
OPTIMIZE ▯..
```

Mouse keys

Ctrl+Right optimizes the group.

See also SET, SPLIT, MOVE, ROUTE

The OPTIMIZE command (which is only applicable in a board drawing) joins wire segments which lie in one straight line. The individual segments must be on the same layer and have the same width. This command is useful to reduce the number of objects in a drawing and to facilitate moving a complete track instead of individual segments.

If signal names are given, or a signal is selected, the command affects only the respective signals.

Note that when selecting an object with the mouse (or selecting a group) only actual signal wires are optimized. To optimize all wires in a board drawing, do

```
OPTIMZE;
```

Automatic Optimization

This wire optimization takes place automatically after MOVE, SPLIT, or ROUTE commands unless it is disabled with the command:

```
SET OPTIMIZING OFF;
```

or you have clicked the same spot twice with the SPLIT command.

The OPTIMIZE command works in any case, no matter if Optimizing is enabled or disabled.

PACKAGE

Function

Defines a package variant for a device.

Syntax

```
PACKAGE
PACKAGE pname vname
PACKAGE pname@lname vname
PACKAGE name
PACKAGE -old_name new_name
PACKAGE -name
```

See also CONNECT, TECHNOLOGY, PREFIX

This command is used in the device edit mode to define, delete or rename a package variant. In the schematic or board editor the PACKAGE command behaves exactly like "CHANGE PACKAGE".

Without parameters a dialog is opened that allows you to select a package and define this variant's name.

The parameters `pname vname` assign the package `pname` to the new variant `vname`.

The notation `pname@lname vname` fetches the package `pname` from library `lname` and creates a new package variant. This can also be done through the library objects' context menu or via *Drag&Drop* from the Control Panel's tree view.

The single parameter `name` switches to the given existing package variant. If no package variants have been defined yet, and a package of the given name exists, a new package variant named '' (an "empty" name) with the given package will be created (this is for compatibility with version 3.5).

If `-old_name new_name` is given, the package variant `old_name` is renamed to `new_name`.

The single parameter `-name` deletes the given package variant.

The name of a package variant will be appended to the device set name to form the full device name. If the device set name contains the character `'?'`, that character will be replaced by the package variant name. Note that the package variant is processed after the technology, so if the device set name contains neither a `'*'` nor a `'?'` character, the resulting device name will consist of *device_set_name*+*technology*+*package_variant*.

Following the PACKAGE command, the CONNECT command is used to define the correspondence of pins in the schematic device to pads on the package.

When the BOARD command is used in schematic editing mode to create a new board, each device is represented on a board layout with the appropriate package as already defined with the PACKAGE command.

Devices without packages

Devices can also be created without assigning a package, for example for frames, supply devices, external or other devices that only make sense in a schematic. This can be done by creating a device set with adequate gates, technologies and attributes (if necessary) without using the PACKAGE command. If saved, a packageless variant is created (with empty string as variant name). As soon as a package is assigned, the packageless variant gets overwritten by this and no further packageless variants can be created.
As soon as gates contain pins, packageless devices only make limited sense (see below).

Supply devices

In order to use supply symbols in schematics, packageless supply devices are common. The device usually consists of exactly one symbol with a *Sup* pin (see PIN command).

External devices

These are for documenting assemblies in schematic that are not relevant for the board

because they are externally added e.g for simulation or test purposes.
Such devices must be marked with the attribute **_EXTERNAL_** (see ATTRIBUTE command). The value is not relevant. In this case any gates with pins can be defined without a package. The atttribute must have been assigned in the library, not in schematic or board.

Note that supply or external devices are no longer treated as such, as soon as packages are assigned. The pins have to be connected with pads then.

PAD

Function
Adds pads to a package.

Syntax
`PAD [diameter] [shape] [orientation] [flags] ['name'] ▯..`

Mouse keys
Right rotates the pad.
Shift+Right reverses the direction of rotating.

See also SMD, CHANGE, DISPLAY, SET, NAME, VIA, Design Rules

The PAD command is used to add pads to a package. When the PAD command is active, a pad symbol is attached to the cursor and can be moved around the screen. Pressing the left mouse button places a pad at the current position. Entering a number changes the diameter of the pad (in the actual unit). Pad diameters can be up to 0.51602 inch (13.1 mm).

The `orientation` (see description in ADD) may be any angle in the range `R0`...`R359.9`. The `S` and `M` flags can't be used here.

Example

```
PAD 0.06 ▯
```

The pad will have a diameter of 0.06 inch, provided the actual unit is "inch". This diameter remains as a presetting for successive operations.

Pad Shapes

A pad can have one of the following shapes:

Square	
Round	
Octagon	octagonal
Long	elongated
Offset	elongated with offset

These shapes only apply to the outer layers (Top and Bottom). In inner layers the shape is always "round".

With elongated pads, the given diameter defines the smaller side of the pad. The ratio between the two sides of elongated pads is given by the parameter Shapes/Elongation in the Design Rules of the board (default is 100%, which results in a ratio of 2:1).

The pad shape or diameter can be selected while the PAD command is active, or it can be changed with the CHANGE command, e.g.:

```
CHANGE SHAPE OCTAGON 
```

The drill size may also be changed using the CHANGE command. The existing values then remain in use for successive pads.

Because displaying different pad shapes and drill holes in their real size slows down the screen refresh, EAGLE lets you change between real and fast display mode by the use of the SET commands:

```
SET DISPLAY_MODE REAL | NODRILL;
```

Note that the actual shape and diameter of a pad will be determined by the Design Rules of the board the part is used in.

Arbitrary Pad Shapes

If the standard pad shapes are not sufficient for a particular package, you can create arbitrary pad shapes by drawing a polygon around a pad, or by drawing wires that have one end connected to the pad.

The following conditions apply:

- A polygon in a signal layer (1-16) is considered connected to a pad if the center of the pad lies within the area defined by the center lines of the polygon wires.
- A wire in a signal layer is considered connected to a pad if one of its end points coincides with the center of the pad. Any wire connected to the other end of such a wire is also electrically connected to the pad.
- Only **one** polygon per pad is taken into account on any given signal layer. If more than one polygon is connected to the same pad in the same layer, they will cause DRC errors.
- Polygons connected to a pad will be ignored by the Autorouter when routing that signal. They will be considered obstacles when routing other signals.
- Wires connected to a pad will be handled like any other signal wires by the Autorouter, with the exception that they cannot be split.
- Solder stop masks are only generated for the pad itself. If any additional solder stop mask is required, it has to be drawn explicitly into the respective layer(s).
- When generating thermals, the additional polygon shape is taken into account.
- If a polygon or wire is connected to more than one pad within a package, only one of the pads will be considered electrically connected to the polygon or wire. The other pads will cause DRC errors, unless they are all connected to the same pin in a device.
- If a polygon contains more than one pad, only one of them (the first one found in the data structure) will generate thermals. If all of these pads shall generate thermals, you need to draw separate polygons (one per pad) that overlap accordingly.
- If several pads are connected to the same pin in a device, and these pads have overlapping wires or polygons in the package, DRC errors will occur unless the pin is actually connected to a net (i.e. the pads are connected to a signal).

Pad Names

Pad names are generated by the program automatically and can be changed with the NAME command. The name can also be defined in the PAD command. Pad name display can be turned on or off by means of the commands:

```
SET PAD_NAMES OFF | ON;
```

This change will be visible after the next screen refresh.

Flags

The following *flags* can be used to control the appearance of a pad:

`NOSTOP`	don't generate solder stop mask
`NOTHERMALS`	don't generate thermals
`FIRST`	this is the "first" pad (which may be drawn with a special shape)

By default a pad automatically generates solder stop mask and thermals as necessary. However, in special cases it may be desirable to have particular pads not do this. The above `NO...` flags can be used to suppress these features.
If the Design Rules of a given board specify that the "first pad" of a package shall be drawn with a particular shape, the pad marked with the `FIRST` flag will be displayed that way.
A newly started PAD command resets all flags to their defaults. Once a flag is given in the command line, it applies to all following pads placed within this PAD command (except for `FIRST`, which applies only to the pad immediately following this option).

Single Pads

Single pads in boards can be used only by defining a package with one pad. Via-holes can be placed in board but they don't have an element name and therefore don't show up in the netlist.

Alter Package

It is not possible to add or delete pads in packages which are already used by a device, because this would change the pin/pad allocation defined with the CONNECT command.

PASTE

Function

Copies the contents of the clipboard or a drawing file to a drawing.

Syntax

```
PASTE [ orientation ] •
PASTE [ orientation ] [ offset ] filename •
```

Mouse keys

Center mirrors the contents of the clipboard.
Right rotates the contents of the clipboard.
Shift+Right reverses the direction of rotating.

See also CUT, COPY, GROUP

See the ADD command for an explanation of `orientation`.

Using the commands GROUP, CUT, and PASTE, parts of a drawing/library can be copied to the same or different drawings/libraries. When using the PASTE command, the following points should be observed:

- CUT/PASTE cannot be used in device editing mode.
- Elements and signals on a board can only be copied to a board.
- Parts, buses and nets on a schematic can only be copied to a schematic.
- Pads and smds can only be copied from package to package.
- Pins can only be copied from symbol to symbol.
- When copying elements, signals, pads, smds and pins, a new name is allocated if the previous name is already used in the new drawing.
- Buses retain the same names.
- Nets retain the same name as long as one of the net segments has a label, or is connected to a supply pin. Otherwise a new name is generated if the previous name is already in use.

If there are modified versions of devices or packages in the clipboard, an automatic library update will be started to replace the objects in the schematic or board with the ones from the clipboard. **Note: You should always run a Design Rule Check (DRC) and an Electrical Rule Check (ERC) after a library update has been performed!**

Pasting from a file

If a file name is given in the command line, the complete content of that file is pasted into the current drawing. If the given file is one of a consistent board/schematic pair, both files will be pasted into the corresponding drawings of the currently edited project.

Assume you have a consistent board/schematic pair that contains the design of an amplifier, where the schematic may consist of several sheets. Now if you want to place this amplifier several times into your project, you can simply do

```
PASTE 100 amplifier.sch ▯
PASTE 200 amplifier.sch ▯
```

This example also shows the use of an `offset`, which adds the given value to all part and net names in the pasted files (unless they retain their name, see below). So the first amplifier channel will have all parts and nets named starting at 100, while the second one will have them start at 200. If no offset is given, new names are generated as necessary.

Just like in a normal PASTE operation, when pasting from a file, nets that have a label or are connected to a supply pin, retain their name while all others will get newly generated names. It is enough for a net to retain its name if it is labeled or connected to a supply pin on one sheet, even if it appears on several sheets.

Unless the PASTE operation is done in a script file, you will be offered a dialog that shows all the net names. By clicking on the names in the "New name" column you can edit individual net names. Icons indicate whether a net in the pasted drawing has a label or a supply pin, and whether the net will be connected to an existing net with the same name in the edited drawing.

If you paste a schematic into a schematic drawing, all sheets of the pasted schematic will be added as separate new sheets to the edited drawing (this is not possible in the Light edition, which can handle only a single schematic sheet). The corresponding board (if any) will be placed below the existing content of the edited board drawing. If you want to have explicit control over where the board is placed, you can perform the PASTE operation in the board, in which case the schematic sheets will be added just the same, but the board will be attached to the mouse cursor and you will be able to place it exactly where you want it.

You can also paste from a file using *Drag&Drop*, by pressing the `Ctrl` key when dropping the file.

If the file name could be mistaken as an orientation or an offset value, you can enclose it in single quotes.

PIN

Function

Defines connection points for symbols.

Syntax

```
PIN 'name' options •..
```

Mouse keys

Right rotates the pin.
Shift+Right reverses the direction of rotating.

See also NAME, SHOW, CHANGE

Options

There are six possible options:

Direction
Function
Length
Orientation
Visible
Swaplevel

Direction

The logical direction of signal flow. It is essential for the Electrical Rule Check (ERC) and for the automatic wiring of the power supply pins. The following possibilities may be used:

NC	not connected
In	input
Out	output (totem-pole)
IO	in/output (bidirectional)
OC	open collector or open drain
Hiz	high impedance output (e.g. 3-state)
Pas	passive (for resistors, capacitors etc.)
Pwr	power input pin (Vcc, Gnd, Vss, Vdd, etc.)
Sup	general supply pin (e.g. for ground symbol)

Default: IO

If Pwr pins are used on a symbol and a corresponding Sup pin exists on the schematic, nets are connected automatically. The Sup pin is not used for components.

Function

The graphic representation of the pin:

None	no special function
Dot	inverter symbol
Clk	clock symbol
DotClk	inverted clock symbol

Default: None

Length

Length of the pin symbol:

Point	pin with no connection or name
Short	0.1 inch long connection
Middle	0.2 inch long connection
Long	0.3 inch long connection

Default: Long

Orientation

The orientation of the pin. When placing pins manually the right mouse button rotates the pin. The parameter "orientation" is mainly used in script files:

R0	connection point on the right
R90	connection point above
R180	connection point on the left
R270	connection point below

Default: R0

Visible

This parameter defines if pin and/or pad name are visible in the schematic:

Off	pin and pad name not drawn
Pad	pad name drawn, pin name not drawn
Pin	pin name drawn, pad name not drawn
Both	pin and pad name drawn

Default: Both

Swaplevel

An integer number. Swaplevel = 0 indicates that a pin can not be swapped with another. The allocation of a number greater than 0 indicates that a pin may be swapped with any other in the same symbol with the same swaplevel number. For example: The inputs of a NAND gate could be allocated the same swaplevel number as they are all identical.

Default: 0

Using the PIN Command

The PIN command is used to define connection points on a symbol for nets. Pins are drawn onto the Symbols layer while additional information appears on the Pins layer. Individual pins may be assigned various options in the command line. The options can be listed in any order or omitted. In this case the default options are valid.

If a name is used in the PIN command, it must be enclosed in apostrophes. Pin names can be changed in the symbol edit mode using the NAME command.

Automatic Naming

Pins may be automatically numbered in the following way. In order to place the pins D0...D7 on a symbol, the first pin is placed with the following command:

```
PIN 'D0' *
```

and the location for the other pins defined with a mouse click for each.

Predefine options with CHANGE

All options may be predefined with CHANGE commands. The options remain in use until edited by a new PIN or CHANGE command.

The SHOW command may be used to show pin options such as Direction and Swaplevel.

Pins with the same Name

If it is required to define several pins in a component with the same name, the following procedure can be used:

For example, suppose that three pins are required for GND. The pins are allocated the names GND@1, GND@2 and GND@3 during the symbol definition. Then only the characters before the "@" sign appear in the schematic.

It is not possible to add or delete pins in symbols which are already used by a device because this would change the pin/pad allocation defined with the CONNECT command.

Pin Lettering

The position of pin and pad names on a symbol relative to the pin connection point can not be changed, nor can the text size. When defining new symbols please ensure their size is consistent with existing symbols.

Inverted pins

The name of an inverted pin ("active low") can be displayed overlined if it is preceded with an exclamation mark (' ! '), as in

```
  !RESET
```

which would result in

$\overline{\text{RESET}}$

You can find further details about this in the description of the TEXT command.

PINSWAP

Function

Swap pins or pads.

Syntax

```
PINSWAP □ □..
```

See also PIN

The PINSWAP command is used to swap pins within the same symbol which have been allocated the same swaplevel (> 0). Swaplevel, see PIN command. If a board is tied to a schematic via Back Annotation two pads can only be swapped if the related pins are swappable.

On a board without a schematic this command permits two pads in the same package to be swapped. The Swaplevel is not checked in this case.

Wires attached to the swapped pins are moved with the pins so that short circuits may appear. Please perform the DRC and correct possible errors.

POLYGON

Function

Draws polygon areas.

Syntax

```
POLYGON [signal_name] [width] □ [curve | @radius] □ □..
```

Mouse keys

Center selects the layer.
Right changes the wire bend style (see SET Wire_Bend).
Shift+Right reverses the direction of switching bend styles.
Ctrl+Right toggles between corresponding bend styles.
Ctrl+Left when placing a wire end point defines arc radius.
Left twice at the same point closes the polygon.

See also CHANGE, DELETE, RATSNEST, RIPUP, WIRE, MITER

The POLYGON command is used to draw polygon areas. Polygons in the layers Top, Bottom, and Route2..15 are treated as signals. Polygons in the layers t/b/vRestrict are protected areas for the Autorouter.

If the *curve* or *@radius* parameter is given, an arc can be drawn as part of the polygon definition (see the detailed description in the WIRE command).

Note

You should avoid using very small values for the *width* of a polygon, because this can cause

extremely large amounts of data when processing a drawing with the CAM Processor.
The polygon *width* should always be larger than the hardware resolution of the output device. For example when using a Gerber photoplotter with a typical resolution of 1 mil, the polygon *width* should not be smaller than, say, 6 mil. Typically you should keep the polygon *width* in the same range as your other wires.

If you want to give the polygon a name that starts with a digit (as in `0V`), you must enclose the name in single quotes to distinguish it from a *width* value.

The parameters `Isolate` and `Rank` only have a meaning for polygons in layers Top...Bottom.

Outlines or Real Mode

Polygons belonging to a signal can be displayed in two different modes:

1. Outlines	only the outlines as defined by the user are displayed.
2. Real mode	all of the areas are visible as calculated by the program.

In "outlines" mode a polygon is drawn with dotted wires, so that it can be distinguished from other wires. The board file contains only the "outlines".

The default display mode is "outlines" as the calculation is a time consuming operation.

When a drawing is generated with the CAM Processor all polygons are calculated.

The RATSNEST command starts the calculation of the polygons (this can be turned off with `SET POLYGON_RATSNEST OFF;`). Clicking the STOP button terminates the calculation of the polygons. Already calculated polygons are shown in "real mode", all others are shown in "outline mode".

The RIPUP command changes the display mode of a polygon to "outline".

CHANGE operations re-calculate a polygon if it was shown in "real mode" before.

Other commands and Polygons

Polygons are selected at their edges (like wires).

SPLIT: Inserts a new polygon edge.

DELETE: Deletes a polygon corner (if only three corners are left the whole polygon is deleted).

CHANGE LAYER: Changes the layer of the whole polygon.

CHANGE WIDTH: Changes the parameter width of the whole polygon.

MOVE: Moves a polygon edge or corner (like wire segments).

COPY: Copies the whole polygon.

NAME: If the polygon is located in a signal layer the name of the signal is changed.

Parameters

Width

Line width of the polygon edges. Also used for filling.

Layer

Polygons can be drawn into any layer. Polygons in signal layers belong to a signal and keep the distance defined in the design rules and net classes from other signals. Objects in the tRestrict layer are substracted from polygons in the Top layer (the same applies to bRestrict/Bottom). This allows you, for instance, to generate "negative" text on a ground area.

Pour

Fill mode (Solid [default], Hatch or Cutout).

Rank

Defines how polygons are subtracted from each other. Polygons with a lower 'rank' appear "first" and thus get subtracted from polygons with a higher 'rank'.
Valid ranks are `1..6`. Polygons with the same rank are checked against each other by the Design Rule Check. The rank parameter only has a meaning for polygons in signal layers (`1..16`) drawn in a board and will be ignored for any other polygons. The default is `1`.

Thermals

Defines how pads and smds are connected (On = thermals are generated [default], Off = no thermals).

Spacing

Distance between fill lines when Pour = Hatch (default: 50 Mil).

Isolate

Distance between polygon areas and other signals or objects in the Dimension layer (default: 0). If a particular polygon is given an Isolate value that exceeds that from the design rules and net classes, the larger value will be taken. See also Design Rules under **Distance** and **Supply**, respectively. **Note that if you give a polygon an Isolate value that exceeds that from the design rules and net classes, small gaps may result between the calculated polygon and objects belonging to the same signal as the polygon itself, which may lead to problems during manufacturing! It is therefore recommended to leave this parameter at 0, unless you know exactly what you are doing!**

Orphans

As a polygon automatically keeps a certain distance to other signals it can happen that the polygon is separated into a number of smaller polygons. If such a polygon has no electrical connection to any other (non-polygon) object of its signal, the user might want it to disappear. With the parameter Orphans = Off [default] these isolated zones will disappear. With Orphans = On they will remain. If a signal consists only of polygons and has no other electrically connected objects, all polygon parts will remain, independent of the setting of the Orphans parameter.

Under certain circumstances, especially with Orphans = Off, a polygon can disappear completely. In that case the polygon's original outlines will be displayed on the screen, to make it possible to delete or otherwise modify it. When going to the printer or CAM Processor these outlines will not be drawn in order to avoid short circuits. A polygon is also displayed with its original outlines if there are other non-polygon objects in the signal, but none of them is connected to the polygon.

Thermal dimensions

The width of the conducting path in the thermal symbol is calculated as follows:

- Pads: half the drill diameter of the pad
- Smds: half the smaller side of the smd
- at least the width of the polygon
- a maximum of twice the width of the polygon

Outlines data

The special signal name _OUTLINES_ gives a polygon certain properties that are used to generate outlines data (for example for milling prototype boards). This name should not be used otherwise.

Hatched polygons and airwires

Depending on the value of the *spacing* parameter, pads, smds, vias and wires inside a hatched polygon that are connected to the same signal as the polygon may "fall through" the raster and thus have airwires generated to indicate their connection to the signal.

When calculating whether such an object is actually solidly connected to the hatched polygon, it is reduced to several "control points". For a round pad, for instance, these would be the north, east, west and south point on the pad's circumference, while for a wire it's the two end points. A solid connection is considered to exist if there is at least one line in the calculated polygon (outline or hatch line) that runs through these points with its center line.

Thermal and annulus rings inside a hatched polygon that do not have solid contact to any of the polygon lines are not generated.

Polygon cutouts

The special pour style "Cutout" makes a polygon be subtracted from all other signal polygons within the same layer, independent of their Rank.

Only polygons in signal layers can have the pour style "Cutout".

The outlines of a cutout polygon are always drawn as dotted lines on the screen, even after the signal polygons have been calculated using RATSNEST.

The wire width of a cutout polygon is taken into account when subtracting it from other signal polygons. It may be arbitrarily small (even zero) without causing large amounts of CAM data (as opposed to "solid" polygons, where the wire width should not be too small).

PREFIX

Function

Defines the prefix for a symbol name.

Syntax

```
PREFIX prefix_string;
```

See also CONNECT, PACKAGE, VALUE

This command is used in the device editor mode to determine the initial characters of automatically generated symbol names when a symbol is placed in a schematic using the ADD command.

Example

```
PREFIX U;
```

If this command is used when editing, for example, a 7400 device, then gates which are later placed in a schematic using the ADD command will be allocated the names U1, U2, U3 in sequence. These names may be changed later with the NAME command.

PRINT

Function

Prints a drawing to the system printer.

Syntax

```
PRINT [factor] [-limit] [options] [;]
```

See also CAM Processor, printing to the system printer

The PRINT command prints the currently edited drawing to the system printer.

Colors and fill styles are used as set in the editor window. This can be changed with the `SOLID` and `BLACK` options. The color palette used for the printout is always that for white background.

If you want to print pads and vias "filled" (without the drill holes being visible), use the command

```
SET DISPLAY_MODE NODRILL;
```

Please note that polygons in boards will not be automatically calculated when printing via the PRINT command! Only the outlines will be drawn. To print polygons in their calculated shape you have to use the RATSNEST command before printing.

You can enter a `factor` to scale the output.

The `limit` parameter is the maximum number of pages you want the output to use. The number has to be preceded with a `'-'` to distinguish it from the `factor`. In case the drawing does not fit on the given number of pages, the `factor` will be reduced until it fits. Set this parameter to `-0` to allow any number of pages (and thus making sure the printout uses exactly the given scale factor).

If the PRINT command is not terminated with a `';'`, a print dialog will allow you to set print options. Note that options entered via the command line will not be stored permanently in the print setup unless they have been confirmed in the print dialog (i.e. if the command has not been terminated with a `';'`).

The following `options` exist:

`MIRROR`	mirrors the output
`ROTATE`	rotates the output by 90°
`UPSIDEDOWN`	rotates the drawing by 180°. Together with `ROTATE`, the drawing is rotated by a total of 270°
`BLACK`	ignores the color settings of the layers and prints everything in black
`SOLID`	ignores the fill style settings of the layers and prints everything in solid
`CAPTION`	prints a caption at the bottom of the page
`FILE`	prints the output into a file; the file name must immediately follow this option
`PRINTER`	prints to a specific printer; the printer name must immediately follow this option
`PAPER`	prints on the given paper size; the paper size must immediately follow this option
`SHEETS`	prints the given range of sheets; the range (from-to) must immediately follow this option
`WINDOW`	prints the currently visible window selection of the drawing
`PORTRAIT`	prints in portrait orientation
`LANDSCAPE`	prints in landscape orientation

If any of the `options` `MIRROR`...`CAPTION` is preceeded with a `'-'`, that option is turned off in case it is currently on (from a previous PRINT). A `'-'` by itself turns off all `options`.

Printing to a file

The `FILE` option can be used to print the output into a file. If this option is present, it must be immediately followed by the name of the output file.

If the output file name has an extension of `".pdf"` (case insensitive), a PDF file will be created. A PDF file can also be created by selecting "Print to File (PDF)" from the "Printer" combo box in the print dialog. Texts in a PDF file can be searched in a PDF viewer, as long as they are not using the vector font.

If the output file name has an extension of `".ps"` (case insensitive), a Postscript file will be created.

If the file name is only an `"*"` or `"*.ext"` (an asterisk followed by an extension, as in `"*.pdf"`, for instance), a file dialog will be opened that allows the user to select or enter the actual file name.

If the file name is only an extension, as in `".pdf"`, the output file name will be the same as the drawing file name, with the extension changed to the given string.

The file name may contain one or more of the following placeholders, which will be replaced with the respective string:

`%E`	the loaded file's extension (without the `'.'`)
`%N`	the loaded file's name (without path and

	extension)
`%P`	the loaded file's directory path (without file name)
`%%`	the character `'%'`

For example, the file name

```
%N.cmp.pdf
```

would create *`boardname`*`.cmp.pdf`.

If both the `FILE` and the `PRINTER` option are present, only the last one given will be taken into account

Printing to a given paper size

The `PAPER` option defines the size of the paper to print on. It must be immediately followed by one of the paper size names listed in the *Paper* combo box of the PRINT dialog, like `A4`, `Letter` etc. If a custom paper size shall be set, it has to be given in the format

```
Width x Height Unit
```

(without blanks), as in

```
PRINT PAPER 200x300mm
PRINT PAPER 8.0x11.5inch
```

Width and *Height* can be floating point numbers, and the *Unit* may be either `mm` or `inch` (the latter may be abbreviated as `in`). Paper names must be given in full, and are case insensitive. If both the `PRINTER` and `PAPER` option are used, the `PRINTER` option must be given first. Custom paper sizes may not work with all printers. They are mainly for use with Postscript or PDF output.

Printing a range of sheets

The `SHEETS` option can be used to print a range of sheets from a schematic. The range is given as two numbers, delimited by a `'-'`, as in `2-15`. Without this option, only the currently edited sheet is printed. To print all sheets, the range `ALL` can be used (which is case insensitive, but must be written in full). A range can also consist of just a single number, as in `42`, which will print exactly that sheet. If no schematic is loaded, this option has no meaning.

Examples

`PRINT`	opens the print dialog in which you can set print options
`PRINT;`	immediately prints the drawing with the default options
`PRINT - MIRROR BLACK SOLID;`	prints the drawing mirrored, with everything in black and solid
`PRINT 2.5 -1;`	prints the drawing enlarged by a factor of 2.5, but makes sure that it does not exceed **one** page
`PRINT FILE .pdf;`	prints the drawing into a PDF file with the same name as the drawing file
`PRINT SHEETS 2-15 FILE .pdf;`	prints the sheets 2 through 15 into a PDF file with the same name as the drawing file

QUIT

Function
Quits the program
Syntax
```
QUIT
```

This command ends the editing session. If any changes have been made but the drawing has not yet been saved, a popup menu will ask you if you want to save the drawing/library first.

You can also exit from EAGLE at any time by pressing `Alt+X`.

RATSNEST

Function
Calculates the shortest possible airwires and polygons.
Syntax
```
RATSNEST
RATSNEST signal_name ..
RATSNEST ! signal_name ..
```

See also SIGNAL, MOVE, POLYGON, RIPUP

The RATSNEST command assesses the airwire connections in order to achieve the shortest possible paths, for instance, after components have been moved. After reading a netlist via the SCRIPT command, it is also useful to use the RATSNEST command to optimize the length of airwires.

The RATSNEST command also calculates all polygons belonging to a signal. This is necessary in order to avoid the calculation of airwires for pads already connected through polygons. All of the calculated polygon areas are then being displayed in the "real mode". You can switch back to the faster "outline mode" with the RIPUP command.
The automatic calculation of the polygons can be turned off with

```
SET POLYGON_RATSNEST OFF;
```

Note that RATSNEST doesn't mark the board drawing as modified, since the calculated polygon data (if any) is not stored in the board, and the recalculated airwires don't really constitute a modification of the drawing.

Zero length airwires

If two or more wires of the same signal on different routing layers end at the same point without being connected through a pad or a via, a *zero length airwire* is generated, which will be displayed as an X-shaped cross in the Unrouted layer. The same applies to smds that belong to the same signal and are placed on opposite sides of the board.

Such *zero length airwires* can be picked up with the ROUTE command just like ordinary airwires. They may also be handled by placing a VIA at that point.

Making sure everything has been routed

If there is nothing left to be routed, the RATSNEST command will respond with the message

```
Ratsnest: Nothing to do!
```

Otherwise, if there are still airwires that have not been routed, the message

```
Ratsnest: xx airwires.
```

will be displayed, where `xx` gives the number of unrouted airwires.

Wildcards

If a `signal_name` parameter is given, the characters `'*'`, `'?'` and `'[]'` are *wildcards* and have the following meaning:

`*`	matches any number of any characters
`?`	matches exactly one character
`[...]`	matches any of the characters between the brackets

If any of these characters shall be matched exactly as such, it has to be enclosed in brackets. For example, `abc[*]ghi` would match `abc*ghi` and not `abcdefghi`.

A range of characters can be given as `[a-z]`, which results in any character in the range `'a'`...`'z'`.

Hiding selected airwires

Sometimes it may be useful to hide the airwires of selected signals, for instance if these will later be connected through a polygon. Typically this could be supply signals, which have a lot of airwires that will never be routed explicitly and just obscure the other signals' airwires.

To hide airwires the RATSNEST command can be given the exclamation mark (`'!'`), followed by a list of signals, as in

```
RATSNEST ! GND VCC
```

which would hide the airwires of the signals `GND` and `VCC`.
To have the airwires displayed again just enter the RATSNEST command without the `'!'` character, and the list of signals:

```
RATSNEST GND VCC
```

This will activate the display of the airwires of the signals `GND` and `VCC` and also recalculates them. You can also recalculate the airwires (and polygons) of particular signals this way.

The signal names may contain wildcards, and the two variants may be combined, as in

```
RATSNEST D* ! ?GND VCC
```

which would recalculate and display the airwires of all signals with names beginning with `'D'`, and hide the airwires of all the various GND signals (like AGND, DGND etc.) and the VCC signal. Note that the command is processed from left to right, so in case there is a DGND signal the example would first process it for display, but then hide its airwires.

To make sure all airwires are displayed enter

```
RATSNEST *
```

Note that the SIGNAL command will automatically make the airwires of a signal visible if a new airwire is created for that signal. The RIPUP command on the other hand will not change the state of hiding airwires if a wire of a signal is changed into an airwire.

Differential Pairs

Airwires for Differential Pair signals prefer open wire ends.

RECT

Function
Adds rectangles to a drawing.
Syntax
`RECT [orientation] ▯ ▯..`
Mouse keys
Center selects the layer.

See also CIRCLE

The RECT command is used to add rectangles to a drawing. The two points define two opposite corners of the rectangle. Pressing the center mouse button changes the layer to which the rectangle is to be added.

The `orientation` (see description in ADD) may be any angle in the range `R0`...`R359.9`. The `S` and `M` flags can't be used here. Note that the coordinates are always defined at an orientation of `R0`. The possibility of entering an `orientation` in the RECT command is mainly for use in scripts, where the rectangle data may have been derived through a User Language Program from the UL_RECTANGLE object. When entering a non-zero orientation interactively, the corners of the rectangle may not appear at the actual cursor position. Use the ROTATE command to interactively rotate a rectangle.

Not Part of Signals

Rectangles in the signal layers Top, Bottom, or Route2...15 don't belong to signals. Therefore the DRC reports errors if they overlap with wires, pads etc.

Restricted Areas

If used in the layers tRestrict, bRestrict, or vRestrict, the RECT command defines restricted areas for the Autorouter.

REDO

Function
Executes a command that was reversed by UNDO.
Syntax
`REDO;`

Keyboard

`F10: REDO` execute the REDO command.
`Shift+Alt+BS: REDO`

See also UNDO, Forward&Back Annotation

In EAGLE it is possible to reverse previous actions with the UNDO command. These actions can be executed again by the REDO command. UNDO and REDO operate with a command memory which exists back to the last EDIT, OPEN or REMOVE command.

UNDO/REDO is completely integrated within Forward&Back Annotation.

REMOVE

Function

Deletes files, devices, symbols, packages, and sheets.

Syntax

```
REMOVE name
REMOVE name.Sxx
```

See also OPEN, RENAME

Files

The REMOVE command is used to delete the file `name` if in board or schematic editing mode.

Devices, Symbols, Packages

The REMOVE command is used to delete the device, symbol or package "name" from the presently opened library. The name may include an extension (for example REMOVE name.pac). If the name is given without extension, you have to be in the respective mode to remove an object (i.e. editing a package if you want to remove packages).

Symbols and packages can be erased from a library only if not used by a device.

REMOVE in a library clears the UNDO buffer.

Sheets

The REMOVE command may also be used to delete a sheet from a schematic. The name of the presently loaded schematic can be omitted. The parameter xx represents the sheet number, for example:

```
REMOVE .S3
```

deletes sheet number 3 from the presently loaded schematic.

If you delete the currently loaded sheet, sheet number 1 will be loaded after the command has been executed. All sheets with a higher number than the one deleted will get a number reduced by one.

RENAME

Function

Renames symbols, devices or packages.

Syntax

```
RENAME old_name new_name;
```

See also OPEN

The RENAME command is used to change the name of a symbol, device or package. The appropriate library must have been opened by the OPEN command before.

The names may include extensions (for example RENAME name1.pac name2[.pac] - note that the extension is optional in the second parameter). If the first parameter is given without extension, you have to be in the respective mode to rename an object (i.e. editing a package if you want to rename packages).

RENAME clears the UNDO buffer.

REPLACE

Function

Replace a part.

Syntax

```
REPLACE •..
REPLACE device_name[@library_name] •..
REPLACE part_name device_name[@library_name] ..
REPLACE package_name •..
REPLACE element_name package_name ..
```

See also SET, UPDATE

The REPLACE command can be used to replace a part with a different device (even from a different library). The old and new device must be compatible, which means that their used gates and connected pins/pads must match, either by their names or their coordinates.

Without parameters the REPLACE command opens a dialog from which a device can be selected from all libraries that are currently in use. After such a device has been selected, subsequent mouse clicks on parts will replace those parts' devices with the selected one if possible.

If a `device_name` is given, that device will be used for the replace operation.

With both a `part_name` and a `device_name`, the device of the given part will be replaced (this is useful when working with scripts).

If a `library_name` is given and it contains blanks, the whole `device_name@library_name` needs to be enclosed in single quotes.

If only a board is being edited (without a schematic), or if elements in the board are being replaced that have no matching part in the schematic, the REPLACE command has two different modes that are chosen by the SET command.

The first mode (default) is activated by the command:

```
SET REPLACE_SAME NAMES;
```

In this mode the new package must have the same pad and smd names as the old one. It may be taken from a different library and it may contain additional pads and smds. The position of pads and smds is irrelevant.

The second mode is activated by the command

```
SET REPLACE_SAME COORDS;
```

In this mode, pads and smds of the new package must be placed at the same coordinates as in the old one (relative to the origin). Pad and smd names may be different. The new package may be taken from a different library and may contain additional pads and smds.

Pads of the old package connected with signals must be present in the new package. If this condition is true the new package may have less pads than the old one.

REPLACE functions only when the appropriate tOrigins/bOrigins layer is displayed.

If there is already a package with the same name (from the same library) in the drawing, and the library has been modified after the original object was added, an automatic library update will be started and you will be asked whether objects in the drawing shall be replaced with their new versions.

Note: A REPLACE operation automatically updates all involved library objects as necessary. This means that other parts (on other schematic sheets or in other locations on the board) may be changed, too. You should always run a Design Rule Check (DRC) and an Electrical Rule Check (ERC) after a REPLACE operation!

RIPUP

Function

Changes routed wires and vias into airwires.
Changes the display of polygons to "outlines".

Syntax

```
RIPUP;
RIPUP [ @ ] [ ! ] ▯..
RIPUP [ @ ] [ ! ] signal_name..
```

Mouse keys

Ctrl+Right rips up the group.

See also DELETE, GROUP, POLYGON, RATSNEST

The RIPUP command changes routed wires (tracks) into airwires. That can be done for:

- all signals (RIPUP;)
- all signals except certain ones (e.g. RIPUP ! GND VCC;)
- one or more signals (e.g. RIPUP D0 D1 D2;)
- certain segments (chosen with one or more mouse clicks)
- all polygons (RIPUP @;)
- all polygons of certain signals (e.g. RIPUP @ GND VCC;)
- all polygons except those of certain signals (e.g. RIPUP @ ! GND VCC;)

Selecting an airwire with RIPUP converts all adjacent routed wires and vias into airwires, up to the next pad, smd or airwire.

```
RIPUP signal_name..
```

rips up the complete signal "signal_name" (several signals may be listed, e.g. `RIPUP D0 D1 D2;`).

```
RIPUP ▯..
```

rips up segments selected by the mouse click up to the next pad/smd.

```
RIPUP;
```

removes only signals which are connected to elements (e.g. board crop marks are not affected). The same applies if RIPUP is used on a group.

Note: in all cases the RIPUP command only acts on objects that are in layers that are currently visible!

Wildcards

If a `signal_name` parameter is given, the characters `'*'`, `'?'` and `'[]'` are *wildcards* and have the following meaning:

`*`	matches any number of any characters
`?`	matches exactly one character
`[...]`	matches any of the characters between the brackets

If any of these characters shall be matched exactly as such, it has to be enclosed in brackets. For example, `abc[*]ghi` would match `abc*ghi` and not `abcdefghi`.

A range of characters can be given as `[a-z]`, which results in any character in the range `'a'`...`'z'`.

Polygons

If the RIPUP command with a name is applied to a signal which contains a polygon the polygon will be displayed with its outlines (faster screen redraw!). Use the RATSNEST command to have polygons displayed in the "real mode" again.

ROTATE

Function
: Rotates objects.

Syntax

```
ROTATE orientation ▯ ..
ROTATE orientation 'name' ..
```

Mouse keys

Ctrl+Right rotates the group.
Left&Drag rotates the object by any angle.
Ctrl+Right&Drag rotates the group by any angle.

See also ADD, MIRROR, MOVE, LOCK, GROUP

The ROTATE command is used to change the orientation of objects.

If `orientation` (see description in ADD) is given, that value will be added to the orientation of the selected object instead.

Prepending `orientation` with the character `'='` causes the value not to be added, but instead to be set absolutely.

Parts, pads, smds and pins can also be selected by their name, which is especially useful if the object is outside the currently shown window area. For example

```
ROTATE =MR90 IC1
```

would set the orientation of element IC1 to MR90, regardless of its previous setting.

Attributes of parts can be selected by entering the concatenation of part name and attribute name, as in `R5>VALUE`.

The quotes around `name` are necessary to distinguish it from an orientation parameter as in

```
ROTATE R45 'R1'
```

They can be left away if the context is clear.

You can use Click&Drag to rotate an object by any angle. Just click on the object and move the mouse (with the mouse button held down) away from the object. After having moved the mouse a short distance, the object will start rotating. Move the mouse until the desired angle has been reached and then release the mouse button. If, at some point, you decide to rather not rotate the object, you can press the ESCape key while still holding the mouse button pressed. The same operation can be applied to a group by using the right mouse button. The group will be rotated around the point where the right mouse button has been pressed down.

Parts cannot be rotated if they are locked, or if any of their connected pads would extend outside the allowed area (in case you are using a limited edition of EAGLE).

Elements

When rotating an element, wires (tracks) connected to the element are moved at the connection points (beware of short circuits!).

Elements can only be rotated if the appropriate tOrigins/bOrigins layer is visible.

Text

Text is always displayed so that it can be read from the bottom or from the right - even when rotated. Therefore after every two rotations it appears the same way, but the origin has moved from the lower left to the upper right corner. Remember this if a text appears to be unselectable!

If you want to have text that is printed "upside down", you can set the "Spin" flag for that text.

ROUTE

Function

Converts unrouted connections into routed wires (tracks).

Syntax

```
ROUTE [width] ▯ [curve | @radius] ▯..
ROUTE name ..
```

Mouse keys

Ctrl+Left starts routing at any given point along a wire or via.
Shift+Left starts routing with the same width as an existing wire.
Center selects the layer.
Right changes the wire bend style (see SET Wire_Bend).
Shift+Right reverses the direction of switching bend styles.
Ctrl+Right toggles between corresponding bend styles.
Shift+Left places a via at the end point.
Ctrl+Left when placing a wire end point defines arc radius.

See also AUTO, UNDO, WIRE, MITER, SIGNAL, SET, RATSNEST

The ROUTE command activates the manual router which allows you to convert airwires (unrouted connections) into real wires.

The first point selects an unrouted connection (a wire in the Unrouted layer) and replaces one end of it by a wire (track). The end which is closer to the mouse cursor will be taken. Now the wire can be moved around (see also WIRE). The right mouse button will change the wire bend and the center mouse button will change the layer. Please note that only those signal layers (1 through 16) are available that have been entered into the layer setup in the Design Rules.

When the final position of the wire is reached, a further click of the left mouse button will place the wire and a new wire segment will be attached to the cursor. If the `Shift` key is held down in such a situation, a Via will be generated at that point if this is possible and the airwire hasn't already been completely routed. The generated Via will have either the appropriate length or, if such a length can't be determined, will go from layer 1 through 16.

When the layer has been changed and a via-hole is thus necessary, it will be added automatically as the wire is placed. When the complete connection has been routed a 'beep' will be given and the next unrouted connection can be selected for routing.

Only the minimum necessary vias will be set (according to the layer setup in the Design Rules). It may happen that an already existing via of the same signal is extended accordingly, or that existing vias are combined to form a longer via if that's necessary to allow the desired layer change. If a via is placed at the start or end point, and there is an SMD pad at that location, the via will be a *micro via* if the current routing layer is one layer away from the SMD's layer (this applies only if micro vias have been enabled in the Design Rules).

While the ROUTE command is active the wire width can be entered from the keyboard.

If the *curve* or *@radius* parameter is given, an arc can be drawn as part of the track (see the detailed description in the WIRE command).

If the `Ctrl` key is pressed while selecting the starting point and there is no airwire at that point, a new airwire will be created automatically. The starting point of that airwire will be that point on the selected wire or via that is closest to the mouse cursor (possibly snapped to the nearest grid point). The far end of the airwire will dynamically point to a target segment that is different from the selected one. If the selected signal is already completely routed, the far end will point to the starting point instead. If the selected wire is an arc, the airwire will start at the closest end point of the wire.

If a `name` is given, the airwire of that signal that is closest to the mouse cursor is selected. If `name` could be interpreted as a *with*, *curve* or *@radius* it has to be written in single quotes.

Selecting the routing layer and wire width

When you select an airwire, the initial layer in which to route is determined by considering the objects at the starting point as follows:

- if there is an object in the current layer, the current layer is kept
- else one of the layers of the objects at that point will be taken

When selecting an airwire, the wire width for routing will be that defined by the Design Rules and the net class of the selected signal if the flag "Options/Set/Misc/Auto set route width and drill" is set. You can select a different width wile the airwire is attached to the cursor, and the track will be rerouted with the new width. The same applies to the via data.

When routing an airwire that starts at an already routed wire, the new wire's width is automatically adjusted to that of the existing wire if the `Shift` key is pressed when selecting the airwire.

Snap Function

The end point of the dynamically calculated airwire is always used as an additional snap point, even if it is off grid. If the remaining airwire has a length that is shorter than SNAP_LENGTH, the routed wire automatically snaps to the airwire's end point, and stays there until the mouse pointer is moved at least SNAP_LENGTH away from that point. The minimum distance for this snap function can be defined with the command

```
SET SNAP_LENGTH distance;
```

where "distance" is the snap radius in the current grid unit.

Follow-me Router

With the special wire bend styles `8` and `9`, the ROUTE command works as a "Follow-me" router. This means that the selected airwire will be routed fully automatically by the Autorouter.

Wire bend style `8` routes only the shorter side of the selected airwire, while `9` routes both sides. Once the automatic routing process is complete (which may take a while, so be patient), the airwire will be replaced by the actual routed wires and vias. If the routing couldn't be completed (for instance due to Design Rules restrictions), the cursor changes into a "forbidden" sign. With bend style `9` it is possible that only one side of the airwire can be routed, while the other side can't.

Whenever the mouse is moved, any previous result is discarded and a new calculation is started. Once the result is acceptable, just click the left mouse button to place it.

The Follow-me router works by marking the grid point at the current mouse position as a starting point, and uses the Autorouter to find a path from that point to any point along the signal segment at which the selected airwire ends (which is not necessarily the exact end point of the airwire). The starting point also considers the currently selected layer, so don't be surprised if the router places a via at that point. By changing the current layer you can influence the routing result.

The routing grid is taken from the actual grid setting at the time the airwire is selected.

The routing parameters (like cost factors, preferred directions etc.) are those defined in the dialog of the AUTO command.

The following particularities apply:

- The Follow-me router doesn't calculate the polygons. If you want them to be calculated, run the RATSNEST command first.
- Since the starting point has to be part of the routed track, the result may be a T-shaped connection, with an unnecessary wire reaching to the starting point. Simply move the mouse cursor towards the actual connection to avoid this.
- Both ends of the airwire are routed separately in bend style `9`, which may lead to wires and/or vias overlapping each other. Move the mouse cursor until such unwanted effects go away.
- Depending on the selected routing layer for the start point it may happen that unnecessary vias are created. Select a different routing layer to avoid this.
- If the maximum number of allowed vias is set to `0` in the Follow-me router parameters, and you change the layer while an airwire is attached to the mouse cursor, the router may place a via at the starting point of the short end of the selected airwire (if this is at all possible according to the Design Rules, restricted areas etc.).
- When in Follow-me mode, the right mouse button toggles between routing only the shorter end of the selected airwire, or both ends. To get back to manual routing you need to click on one of the bend style buttons, or enter the SET Wire_Bend command with a value smaller than `8`.
- The Follow-me router can only place round or octagonal shaped Vias, not square ones.
- The Miter parameter has no meaning in Follow-me mode.
- The parameters for the Follow-me router are stored together with the rest of the Autorouter parameters, but in a separate section. This is because basically the Follow-me parameters should behave like those of the "Route" section in the Autorouter parameters (in order to not obscure too much area), but also might have a tendency towards those of the optimize sections.
- If a board file containing Autorouter parameters is saved with this version of EAGLE and loaded into an older version, the Autorouter parameters may be reported as invalid by the older version, and it will use default values. You can save the Autorouter parameters into a *.ctl file and explicitly load them into the older version if necessary.
- The special mouse key functions Ctrl+Left (start routing at any given point along a wire or via), Shift+Left (place a via at the end point) and Ctrl+Left (define arc radius) don't work in Follow-me mode.

Differential Pair routing

Differential Pairs are signals that need to be routed in parallel and with a specific distance between them.

The following particularities apply:

- A Differential Pair consists of two signals that have the same name, only one ending with `_P` (the "positive" signal) and the other one with `_N` (the "negative" signal), for instance `CLOCK_P` and `CLOCK_N`. The two signals must also belong to the same net class.

- When selecting an airwire of a Differential Pair, both signals are routed in parallel. The distance between the two signals as well as the wire and via sizes are determined by the signals' net class. This is done independent of the setting of "Options/Set/Misc/Auto set route width and drill".
- If you don't want to route both signals, you can press the ESCape key to drop the second airwire.
- At the beginning of routing a Differential Pair (when the starting points of the airwires don't have the necessary distance, yet) signal wires are generated from the starting points to the current mouse cursor position, according to the current wire bend style. Note that there may be cases where these wires overlap, so please make sure you choose a proper point from where to start the actual parallel routing.
- The coordinates given while routing a Differential Pair form a "center line" along which the actual signal wires are placed left and right with the proper distance.
- Since the pads a Differential Pair is connected to typically don't have the same distance as used for the signal wires, you may have to route such signals from both ends. This means, you start at one part, route towards the other part, and then route the rest starting from the other part. This is necessary because only the first step in a routing sequence generates wires that start at positions that don't have a proper distance.
- If you route towards the wire end points of a Differential Pair in a different layer, and the wires are fully aligned, the proper vias will be generated automatically.
- The special mouse key functions Shift+Left (place a via at the end point) and Ctrl+Left (define arc radius) don't work in Differential Pair mode.
- When you start routing at any point of a signal (with Ctrl+Left) you can only route the selected signal, and not the Differential Pair this signal might be part of.
- Differential Pairs can only be routed fully manually. The Follow-me router and the Autorouter treat them like regular signals.

You can use the MEANDER command to balance the lengths of the two signals that form a differential pair.

RUN

Function

Executes a User Language Program.

Syntax

```
RUN file_name [argument ...]
```

See also SCRIPT

The RUN command starts the User Language Program from the file `file_name`.
The optional `argument` list is available to the ULP through the Builtin Variables `argc` and `argv`.

Started from a context menu the according object is assigned to a group. It can be identified with the builtin function ingroup() for further processing..

Running a ULP from a script file

If a ULP is executed from a script file and the program returns an integer value other than `0` (either because it has been terminated through a call to the `exit()` function or because the STOP button was clicked), execution of the script file will be terminated.

Editor commands resulting from running a ULP

A ULP can also use the `exit()` function with a `string` parameter to send a command string back to the editor window.

SCRIPT

Function

Executes a command file.

Syntax

```
SCRIPT file_name;
```

See also SET, MENU, ASSIGN, EXPORT, RUN

The SCRIPT command is used to execute sequences of commands that are stored in a script file. If SCRIPT is typed in at the keyboard and "file_name" has no extension, the program automatically uses ".scr".

Examples

`SCRIPT nofill`	executes nofill.scr
`SCRIPT myscr.`	executes myscr (no Suffix)
`SCRIPT myscr.old`	executes myscr.old

Please refer to the EXPORT command for different possibilities of script files.

If the SCRIPT command is selected with the mouse, a popup menu will show all of the files which have the extension ".scr" so that they can be selected and executed.

The SCRIPT command provides the ability to customize the program according to your own wishes. For instance:

- change the command menu
- assign keys
- load pc board shapes
- change colors

SCRIPT files contain EAGLE commands according to the syntax rules. Lines beginning with `'#'` are comment.

Continued Lines

SCRIPT files contain one or more commands in every line according to the syntax rules. The character '\' at the end of a command line ensures that the first word of the next line is not interpreted as a command. This feature allows you to avoid apostrophes in many cases.

Set Default Parameters

The SCRIPT file eagle.scr - if it exists in the project directory or in the script path - is executed each time a new drawing is loaded into an editor window (or when the drawing type is changed in a library).

Script Labels

The default SCRIPT file eagle.scr makes use of labels of the form

```
EDITOR:
```

where EDITOR is one of SCH, BRD, LBR, DEV, PAC and SYM. This ensures that only the appropriate section is executed in the editor. For example, when a new board is opened, only the section starting with `BRD:` is executed (until the next label if any). This also offers the possibility for editor specific menus using the MENU command. The label must be at the line beginning.

Execute Script Files in the Library Editor

All of the layers are recognized only if the library editor has previously been loaded.

SET

Function

Alters system parameters

Syntax

```
SET
SET options;
```

Parameters which affect the behavior of the program, the screen display, or the user interface can be specified with the SET command. The precise syntax is described below.

A dialog in which all the parameters can be set appears if the SET command is entered without parameters.

User Interface

Snap function

```
SET SNAP_LENGTH number;
```

This sets the limiting value for the snap function in the ROUTE command (using the current unit).
Default: 20 mil
If tracks are being laid with the ROUTE command to pads that are not on the grid, the snap function will ensure that a route will be laid to the pad within the snap-length.

```
SET CATCH_FACTOR value;
```

Defines the distance from the cursor up to which objects are taken into account when clicking with the mouse. The value is entered relative to the height (or width, whichever is smaller) of the presently visible part of the drawing. It applies to a zoom level that displays at least a range of 4 inch and inrceases logarithmically when zooming further in. A value of 0 turns this limitation off. Values < 1 are interpreted as factor, values ≥ 1 as percents.

Default: 0.05 (5%).
`SET SELECT_FACTOR value;`
This setting controls the distance from the cursor within which nearby objects will be suggested for selection. The value is entered relative to the height (or width, whichever is smaller) of the presently visible part of the drawing. Values < 1 are interpreted as factor, values ≥ 1 as percents.
Default: 0.02 (2%).

Menu contents

`SET USED_LAYERS name | number;`

Specifies the layers which will be shown in the associated EAGLE menus. See the example file `mylayers.scr`.
The layers Pads, Vias, Unrouted, Dimension, Drills and Holes will in any case remain in the menu, as will the schematic layers. Any used signal layers also remain in the menus.
`SET Used_Layers All` activates all layers.

```
SET WIDTH_MENU value..;
SET DIAMETER_MENU value..;
SET DRILL_MENU value..;
SET SMD_MENU value..;
SET SIZE_MENU value..;
SET ISOLATE_MENU value..;
SET SPACING_MENU value..;
SET MITER_MENU value..;
```

The content of the associated popup menus can be configured with the above command for the parameters *width* etc.. A maximum of 16 values is possible for each menu (16 value-pairs in the SMD menu). Without any values (as in `SET WIDTH_MENU;`) the program default values will be restored.
Example:

```
Grid Inch;
Set Width_Menu 0.1 0.2 0.3;
```

Context menus

`SET CONTEXT objecttype text commands;`

For selectable object types context menus (right mouse button) can be extended by arbitrary entries. *objecttype* is not case sensitive. *text* is the menu text, *commands* is the command sequence, that is executed after click on the menu entry. Empty spaces are possible if the expression is set into apostrophs. apostrophs inside have to be doubled (see TEXT). Example:
`SET CONTEXT Element Export 'run myexport.ulp';`
To the context menu for elements the entry *Export* is added, which starts an according ULP.
An existing userdefined entry can also be overwritten.
The settings are stored in the eaglerc file. The number of entries is unlimited.
Deletion of entries:
`SET CONTEXT objecttype;` deletes all entries for this object type. With `SET CONTEXT;` all user defined menu entries are deleted.
All selectable object types are supported. These are attribute, circle, dimension, element, frame, gate, hole, instance, junction, label, pad, pin, rectangle, smd, text, via and wire.

Bend angle for wires

`SET WIRE_BEND bend_nr;`

bend_nr can be one of:
`0`: Starting point - horizontal - vertical - end
`1`: Starting point - horizontal - 45° - end

	2: Starting point - end (straight connection)
	3: Starting point - 45° - horizontal - end
	4: Starting point - vertical - horizontal - end
	5: Starting point - arc - horizontal - end
	6: Starting point - horizontal - arc - end
	7: "Freehand" (arc that fits to wire at start, straight otherwise)
	8: Route short end of airwire in Follow-me router
	9: Route both ends of airwire in Follow-me router
	Note that 0, 1, 3 and 4 may contain additional miter wires (see MITER).
	`SET WIRE_BEND @ bend_nr ...;`
	Defines the bend angles that shall be actually used when switching with the right mouse button.
	`SET WIRE_BEND @;`
	Switches back to using all bend angles.

Beep on/off	`SET BEEP OFF | ON;`

Screen display

Color for grid lines	`SET COLOR_GRID color;`
Layer color	`SET COLOR_LAYER layer color;`
Fill pattern for layer	`SET FILL_LAYER layer fill;`
Grid parameters	`SET MIN_GRID_SIZE pixels;` The grid is only displayed if the grid size is greater than the set number of pixels.
Min. text size shown	`SET MIN_TEXT_SIZE size;` Text less than `size` pixels high is shown as a rectangle on the screen. The setting `0` means that all text will be displayed readably.
Net wire display	`SET NET_WIRE_WIDTH width;`

Pad display	`SET DISPLAY_MODE REAL | NODRILL;`
	REAL: Pads are displayed as they will be plotted.
	NODRILL: Pads are shown without drill hole.
	`SET PAD_NAMES OFF | ON;`
	Pad names are displayed/not displayed.

Bus line display	`SET BUS_WIRE_WIDTH width;`
DRC-Parameter	`SET DRC_FILL fill_name;`
Polygon calculation	`SET POLYGON_RATSNEST OFF | ON;` See POLYGON command.
Vector font	`SET VECTOR_FONT OFF | ON;` See TEXT command.
Cross-reference labels	`SET XREF_LABEL_FORMAT string;` See LABEL command.
Part cross-references	`SET XREF_PART_FORMAT string;` See TEXT command.

Mode parameters

Package check	`SET CHECK_CONNECTS OFF | ON;` The ADD command checks whether every pin has been connected to a pad (with

	CONNECT). This check can be switched off. Nevertheless, no board can be generated from a schematic if a device is found which does not have a package.
REPLACE mode	`SET REPLACE_SAME NAMES | COORDS;`
UNDO buffer on/off	`SET UNDO_LOG OFF | ON;`
Wire optimizing	`SET OPTIMIZING OFF | ON;` If set *on*, wires which lie in one line after a MOVE, ROUTE or SPLIT are subsumed into a single wire. See also OPTIMIZE.
Net wire termination	`SET AUTO_END_NET OFF | ON;` Automatic ending of nets or busses. See NET or BUS.
Automatic junctions	`SET AUTO_JUNCTION OFF | ON;` Automatic setting of junctions. See JUNCTION.
Automatic confirmation	`SET CONFIRM OFF | NO | YES;` Allows confirmation dialogs to be handled automatically (see below for details).

Colors

There are three *palettes* for black, white and colored background, respectively. Each palette has 64 color entries, which can be set to any ARGB value. The palette entry number 0 is used as the background color (in the "white" palette this entry cannot be modified, since this palette will also be used for printing, where the background is always white).

The color palettes can be modified either through the dialog under "Options/Set.../Colors" or by using the command

```
SET PALETTE index argb
```

where *index* is a number in the range 0..63 and *argb* is a hexadecimal value defining the Alpha, Red, Green and Blue components of the color, like 0xFFFFFF00 (which would result in a bright yellow). The alpha component defines how "opaque" the color is. A value of 0x00 means it is completely transparent (i.e. invisible), while 0xFF means it is totally opaque. The alpha component of the background color is always 0xFF. Note that the ARGB value must begin with "0x", otherwise it would be taken as a decimal number. You can use

```
SET PALETTE BLACK|WHITE|COLORED
```

to switch to the black, white or colored background palette, respectively. Note that there will be no automatic window refresh after this command, so you should do a WINDOW; command after this.

By default only the palette entries 0..15 are used and they contain the colors listed below.

The palette entries are grouped into "normal" and "highlight" colors. There are always 8 "normal" colors, followed by the corresponding 8 "highlight" colors. So colors 0..7 are "normal" colors, 8..15 are their "highlight" values, 16..23 are another 8 "normal" colors with 24..31 being their "highlight" values and so on. The "highlight" colors are used to visualize objects, for instance in the SHOW command.

`Color`, listed according to color numbers, which can be used instead of the color names. Used to specify colors:

0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	LGray
8	DGray
9	LBlue
10	LGreen
11	LCyan
12	LRed
13	LMagenta
14	Yellow
15	White

`Fill` specifies the style with which wires and rectangles in a particular layer are to be filled. This parameter can also be replaced with the number at the beginning of each line:

0	Empty
1	Solid
2	Line
3	LtSlash
4	Slash
5	BkSlash
6	LtBkSlash
7	Hatch
8	XHatch
9	Interleave
10	WideDot
11	CloseDot
12	Stipple1
13	Stipple2
14	Stipple3
15	Stipple4

Automatic Confirmation

At times EAGLE prompts the user with informational or warning messages, or requires a decision on how to proceed. This may be unwanted in automated processes (like script files). The command

```
SET CONFIRM YES
```

will automatically confirm every message dialog as if the user had clicked on the "positive" button ("OK" or "Yes"). The dialog itself isn't even presented to the user. Similarly, the command

```
SET CONFIRM NO
```

acts as if the user had clicked on the "negative" button ("No"), if such a button is present. Otherwise it just confirms the dialog.
Finally, the command

```
SET CONFIRM OFF
```

turns off automatic confirmation. If automatic confirmation is not turned off explicitly, it will automatically fall back to `OFF` the next time the editor window accepts some interactive input from the user.

You should not put a `SET CONFIRM YES` at the beginning of a script and then execute lots of commands "silently". It is better to explicitly put this around individual commands, as in

```
SET CONFIRM YES
REMOVE filename
SET CONFIRM OFF
```

Be extremely careful when using this command! Blindly confirming message dialogs may cause important messages to be overlooked. The result may then not be what the user expected.

EagleRc Parameters

Sometimes a small detail of functionality needs to be made adjustable, for instance because some users absolutely need to have it work differently. These parameters are not available in any dialogs, but can only be changed through an entry in the eaglerc file. In order to make this easier, any parameter that is not found amoung the keywords listed above will be looked up in the eaglerc parameters and can thus be changed using the SET command. Note that the parameter names must be written in full and exactly as listed below (case sensitive). The parameter value is typically '0' or '1', to turn the functionality 'off' or 'on', respectively. After changing any of these parameters that influence the way the screen display is drawn, a window refresh may be necessary.

Example

```
SET Option.DrawUnprocessedPolygonEdgesContinuous 1;
```

The following eaglerc parameters are available:

Cmd.Copy.ClassicEagleMode

In older versions of EAGLE the COPY command was used solely to copy objects within a drawing, as opposed to the Windows behavior, where COPY places a copy of the selected objects (i.e. the GROUP) into the system's clipboard. As of version 6, EAGLE's COPY command primarily behaves the same way as in other Windows applications, by putting a copy of the current group into the clipboard. The original functionality of copying selected objects, or copying library objects between libraries, is still fully available, which is especially important to keep existing scripts and ULPs working. What has also often irritated Windows users is that in EAGLE the CUT command has only copied the current group to the clipboard, but did not actually delete the group from the drawing. Since a CUT command that deletes the group would not be of much use in a board/schematic pair that is connected via forward-&backannotation, the CUT command has been removed from the main pulldown menu and the command button toolbar. It is still fully available from the command line or within scripts. Setting this parameter to '1' restores the old behavior of both the COPY and the CUT command. Note that this setting only takes effect the next time you open an editor window.

Cmd.Delete.WireJointsWithoutCtrl

If you insist on having the DELETE command delete wire joints without pressing the Ctrl key, you can set this parameter to '1'.

Cmd.Name.RenameEntireNetByDefault

If a net consists of more than one segment, the NAME command by default acts only upon the selected segment. By setting this parameter to '1' this can be changed to rename the entire net by default. This parameter also applies to busses.

Cmd.Name.RenameEntireSignalByDefault

If a signal contains a polygon, and the NAME command is applied to that polygon, by default only the polygon gets renamed. Setting this parameter to '1' makes the NAME command act upon the entire signal by default.

Cmd.Wire.IgnoreCtrlForRadiusMode

If you don't like the special mode in wire drawing commands that allows for the definition of an arc radius by pressing the Ctrl key when placing the wire, you can set this parameter to '1'. This will turn this feature off for all commands that draw wires.

ControlPanel.View.AutoOpenProjectFolder

The automatic opening of the project folder at program start (or when activating a project by clicking on its gray button) can be disabled by setting this parameter to '0'.

Erc.AllowUserOverrideConsistencyCheck

In order to handle board/schematic pairs that have only minor inconsistencies, the user can enable a dialog that allows him to force the editor to perform forward-/backannotation, even if the ERC detects that the files are inconsistent. This can be done by setting this parameter to '1'. **PLEASE NOTE THAT YOU ARE DOING THIS AT YOUR OWN RISK** - if the files get corrupted in the process, there may be nothing anybody can do to recover them. After all, the ERC **did** state that the files were inconsistent!

Interface.MouseButtonReleaseTimeout

The time (in milliseconds) within which a mouse button release that follows a mouse button press on a button (like, for instance, toolbar buttons) triggers the button's action, even if the mouse button release happened outside the button's area. Default is 500, set this to 0 to turn off this feature. If this parameter is 0 when the program is started, any change to it will only take effect the next time the program is started.

Interface.PreferredUnit

When displaying a numerical value in dialog input fields, the units are determined automatically, so that the representation with the least number of decimal digits is chosen. This can be controlled by setting this parameter to '0' for automatic unit determination (default), '1' for imperial units, and '2' for metric units.

Interface.UseCtrlForPanning

Panning is done by moving the mouse while holding the center mouse button (or mouse wheel) down. In older versions this was done by pressing the Ctrl key instead. If you want the old functionality back, you can set this parameter to '1'. Note, though, that the Ctrl key is now used for special functions in some commands, so when using these special functions (like selecting an object at its origin in MOVE) with this parameter enabled you may inadvertently pan your draw window.

Option.AutoLoadMatchingDrawingFile

If you have a board and schematic editor window open and load an other board (or schematic) in one of these windows, and if that other drawing has a matching schematic (or board), EAGLE asks whether that other drawing shall also be loaded. By setting this parameter to '1', this query can be suppressed, and EAGLE will always automatically load the other drawing.

Option.DrawUnprocessedPolygonEdgesContinuous

If you don't like the way unprocessed polygons display their edges (as dotted lines), you can set this parameter to '1'. The edges of unprocessed polygons will then be displayed as continuous lines, as was the case before version 5 (however, they will not be highlighted).

Option.LayerSequence

The internal layers are rendered in a sequence that mimics the actual layer stack, so that the result looks useful even on printers and PDF or Postscript files, where layers are not transparent. Sometimes user defined layers may need to be rendered before internal layers instead of after them. This parameter can be used to define the sequence in which layers are rendered. It consists of a string of layer numbers or layer ranges, followed by an optional 't' or 'b'.

123	renders layer 123
123t	renders layer 123 if the output is "viewed from top" (not mirrored)
123b	renders layer 123 if the output is "viewed from bottom" (mirrored)
123-140	renders layers 123 through 140 in the given sequence
140-123	renders layers 140 through 123 in the given sequence
*	inserts the default sequence of the internal layers
123b * 123t	makes layer 123 always be rendered first

Note that each layer is rendered only once, even if it is listed several times. The default sequence of the internal layers is
48t 49t 19t 47t 20t 46t 23 27 25 59 57 55 53 50 51 21 44t 45t 37 35 31 29 33 39 41 43t 18t 17t 1-16 17b 18b 43b 42 40 34 30 32 36 38 45b 44b 22 52 54 56 58 60 26 28 24 46b 20b 47b 19b 49b 48b 61-99.
When viewed from top, the layer sequence is rendered from right to left, while when viewed from bottom (mirrored) it is rendered from left to right. For instance, layer 48 (Document) is entered as 48t and 48b to always have it rendered as the last one. Layers 21 (tPlace) and 22 (bPlace), on the other hand, are listed only once, to have them rendered at the proper place, depending on whether the output is mirrored or not.
Any layers that are not explicitly mentioned in the layer sequence are rendered after the given sequence in ascending order.

Option.RatsnestLimit

The RATSNEST command processes all points of a signal, even if that signal is very complex (in previous versions it dropped wire end points from processing if the total number of connection points exceeded 254). This requires more memory when calculating the ratsnest. In case this is a problem on your system, you can revert to the original method by setting this parameter to '254'. The value given here is the number of connection points up to which all wire end points will be taken into account and thus limits the amount of memory used (processing will use up to the square of this value in bytes, so a value of 1024 will limit the used memory to 1MB). A value of '0' means there is no limit. A value of '1' will result in airwires being connected only to pads, smds and vias.

Option.RepositionMouseCursorAfterContextMenu

Normally EAGLE doesn't automatically position the mouse cursor. However, some users want the cursor to be repositioned to the point where it has been before a context menu in the drawing editor was opened. Set this parameter to '1' to get this functionality.

Option.ShowPartOrigins

The origins of parts in a schematic are indicated by small crosses. Set this parameter to '0' to turn this off.

Option.ShowTextOrigins

The origins of texts are indicated by small crosses. Set this parameter to '0' to turn this off.

Option.ToggleCtrlForGroupSelectionAndContextMenu

Since the context menu function on the right mouse button interferes with the selection of groups as it was done before version 5, a group is now selected with Ctrl plus right mouse button. If you want to have the old method of selecting groups back, you can can set this parameter to '1'. This will allow selecting groups with the right mouse button only and require Ctrl plus right mouse button for context menus.

Sch.Cmd.Add.AlwaysUseDeviceNameAsValue

Some users always want to use the device name as part value, even if the part needs a user supplied value. Those who want this can set this parameter to '1'.

Warning.PartHasNoUserDefinableValue

If you don't want the warning message about a part not having a user definable value, you can turn it off by setting this parameter to '0'.

Warning.SupplyPinAutoOverwriteGeneratedNetName

Some users don't want the warning message about a supply pin overwriting a generated net name. Setting this option to '1' disables that warning.

SHOW

Function

Highlights objects.

Syntax

```
SHOW ▯..
SHOW name..
SHOW @ name..
```

Mouse keys

Ctrl+Left toggles the show state of the selected object.

See also INFO

The SHOW command is used to highlight objects. Details are listed in the status bar. Complete signals and nets can be highlighted with the SHOW command. If a bus is selected, all nets belonging to that bus will also be highlighted.

Cross Probing

With active Forward&Back Annotation an object that is highlighted with the SHOW command in a board will also be highlighted in the schematic, and vice versa.

Different Objects

If you select different objects with the SHOW command every single object is highlighted separately. You can select more than one object for highlighting by pressing the Ctrl key when clicking on the objects. When you click on an object that is already highlighted with the Ctrl key pressed, that object will be displayed non-highlighted again.

If several names are entered in one line, all matching objects are highlighted at the same time.

Small Objects

If the @ character is given in the command line, a pointer rectangle is drawn around the shown object. This is helpful in locating small objects that wouldn't show up too well just through highlighting. If more than one object is shown, the rectangle is drawn around all the objects. It may be necessary to zoom out (or do a WINDOW FIT command) in order to see the pointer. If an object with the literal name @ shall be shown, the name must be enclosed in single quotes.

Wildcards

If a `name` parameter is given, the characters `'*'`, `'?'` and `'[]'` are *wildcards* and have the following meaning:

`*`	matches any number of any characters
`?`	matches exactly one character
`[...]`	matches any of the characters between the brackets

If any of these characters shall be matched exactly as such, it has to be enclosed in brackets. For example, `abc[*]ghi` would match `abc*ghi` and not `abcdefghi`.

A range of characters can be given as `[a-z]`, which results in any character in the range `'a'`...`'z'`.

The special pattern `[number..number]` forms a bus name range and is therefore not treated as a wildcard pattern in a schematic.

Objects on different Sheets

If an object given by name is not found on the current schematic sheet, a dialog is presented containing a list of sheets on which the object is found. If the object is not found on any sheet, the sheet number is '-' in this list. Note that this dialog only appears if any of the objects given by name (or wildcards) is not found on the current sheet. If all given objects are found on the current sheet, no dialog appears (even if some of the objects are also present on other sheets). Once the dialog appears, it contains all objects found, even those on the current sheet.

Examples

```
SHOW IC1
```

IC1 is highlighted and remains highlighted until the SHOW command is ended or a different name is entered.

```
SHOW IC*
```

Highlights all objects with names starting with "IC".

SIGNAL

Function

Defines signals.

Syntax

```
SIGNAL □ □..
SIGNAL signal_name □ □..
SIGNAL signal_name element_name pad_name..;
```

See also AUTO, ROUTE, NAME, CLASS, WIRE, RATSNEST, EXPORT

The SIGNAL command is used to define signals (connections between the various packages). The user must define a minimum of two element_name/pad_name pairs, as otherwise no airwire can be generated.

Mouse Input

To do that you select (with the mouse) the pads (or smds) of the elements to be connected, step by step. EAGLE displays the part signals as airwires in the Unrouted layer.

If input with signal_name the signal will be allocated the specified name.

Text Input

Signals may also be defined completely by text input (via keyboard or script file). The command

```
SIGNAL GND IC1 7 IC2 7 IC3 7;
```

connects pad 7 of IC1...3. In order to enter a whole netlist, a script file may be generated, with the extension *.scr. This file should include all of the necessary SIGNAL commands in the format shown above.

On-line Check

If the SIGNAL command is used to connect pads (or smds) that already belong to different signals, a popup menu will appear and ask the user if he wants to connect the signals together, and which name the signal should get.

Outlines data

The special signal name _OUTLINES_ gives a signal certain properties that are used to generate outlines data. This name should not be used otherwise.

SMASH

Function

Separates text variables and attributes from parts or elements.

Syntax

```
SMASH ▯..
SMASH name ..
```

Mouse keys

Ctrl+Right smashes the group.
Shift+Left reverses the text separation ("unsmashes" the part).
Ctrl+Shift+Right reverses the text separation for the group.

See also NAME, VALUE, TEXT, ATTRIBUTE

The SMASH command is used with parts or elements in order to separate the text parameters indicating name, value or attributes. The text may then be placed in a new and more convenient location with the MOVE command.

Parts and elements can also be selected by their name, which is especially useful if the object is outside the currently shown window area. Note that when selecting a multi-gate part in a schematic by name, you will need to enter the full instance name, consisting of part and gate name.

Use of the SMASH command allows the text to be treated like any other text, e.g.

CHANGE SIZE, ROTATE, etc., but the actual text may not be changed.

A "smashed" element can be made "unsmashed" by clicking on it with the `Shift` key pressed (and of course the SMASH command activated).

SMD

Function

Adds smd pads to a package.

Syntax

```
SMD [x_width y_width] [-roundness] [orientation] [flags]
['name'] ▯..
```

Mouse keys

Center selects the layer.
Right rotates the smd.
Shift+Right reverses the direction of rotating.

See also PAD, CHANGE, NAME, ROUTE, Design Rules

The SMD command is used to add pads for surface mount devices to a package. When the SMD command is active, an smd symbol is attached to the cursor. Pressing the left mouse button places an smd pad at the current position. Entering numbers changes the x- and y-width of the smd pad, which can be up to 0.51602 inch (13.1 mm). These parameters remain as defaults for successive SMD commands and can be changed with the CHANGE command. Pressing the center mouse button changes the layer onto which the smd pad will be drawn.

The `orientation` (see description in ADD) may be any angle in the range `R0`...`R359.9`. The `S` and `M` flags can't be used here.

Roundness

The `roundness` has to be entered as an integer number between `0` and `100`, with a negative sign to distinguish it from the width parameters. A value of `0` results in fully rectangular smds, while a value of `100` makes the corners of the smd fully round. The command

```
SMD 50 50 -100 '1' ▯
```

for example would create a completely round smd named '1' at the given mouseclick position. This can be used to create BGA (Ball Grid Array) pads.

Arbitrary Pad Shapes

If the standard smd pad shapes are not sufficient for a particular package, you can create arbitrary smd pad shapes by drawing a polygon around an smd pad, or by drawing wires that have one end connected to the smd pad.

The following conditions apply:

- A polygon is considered connected to an smd pad on the same layer if the center of the pad lies within the area defined by the center lines of the polygon wires.
- A wire is considered connected to an smd pad on the same layer if one of its end

points coincides with the center of the pad. Any wire connected to the other end of such a wire is also electrically connected to the pad.

- The DRC will not check such wires and polygons for minimum width.
- Only **one** polygon per pad is taken into account. If more than one polygon is connected to the same pad, they will cause DRC errors.
- Polygons connected to a pad will be ignored by the Autorouter when routing that signal. They will be considered obstacles when routing other signals.
- Wires connected to a pad will be handled like any other signal wires by the Autorouter, with the exception that they cannot be split.
- Solder stop and cream masks are only generated for the pad itself. If any additional solder stop or cream mask is required, it has to be drawn explicitly into the respective layer(s).
- When generating thermals, the additional polygon shape is taken into account.
- If a polygon or wire is connected to more than one pad within a package, only one of the pads will be considered electrically connected to the polygon or wire. The other pads will cause DRC errors, unless they are all connected to the same pin in a device.
- If a C-shaped polygon connected to a pad would cause a signal polygon in the board to have an "orphan" that lies within the C area, such an orphan will disappear regardless whether the signal polygon in question has its Orphans parameter set to on or off.

Names

SMD names are generated automatically and may be modified with the NAME command. Names may be included in the SMD command if enclosed in single quotes.

Flags

The following *flags* can be used to control the appearance of an smd:

`NOSTOP`	don't generate solder stop mask
`NOTHERMALS`	don't generate thermals
`NOCREAM`	don't generate cream mask

By default an smd automatically generates solder stop mask, cream mask and thermals as necessary. However, in special cases it may be desirable to have particular smds not do this. The above `NO...` flags can be used to suppress these features.
A newly started SMD command resets all flags to their defaults. Once a flag is given in the command line, it applies to all following smds placed within this SMD command.

Single Smds

Single smd pads in boards can only be used by defining a package with one smd.

Alter Package

It is not possible to add or delete smds in packages which are already used by a device, because this would change the pin/smd allocation defined with the CONNECT command.

SPLIT

Function

Splits wires and polygon edges into segments.

Syntax

```
SPLIT • [curve | @radius] •..
```

Mouse keys

Right changes the wire bend style (see SET Wire_Bend).
Shift+Right reverses the direction of switching bend styles.
Ctrl+Right toggles between corresponding bend styles.
Ctrl+Left when placing a wire end point defines arc radius.

Keyboard

`F8: SPLIT` activates the SPLIT command.

See also MITER, MOVE, OPTIMIZE, SET

The SPLIT command is used to split a wire (or segment) or a polygon edge into two segments in order, for example, to introduce a bend. This means you can split wires into parts that can be moved with the mouse during the SPLIT command. A mouseclick defines the point at which the wire is split. The shorter of the two new segments follows the current wire bend rules and may therefore itself become two segments (see SET Wire_Bend), the longer segment is a straight segment running to the next end point.

If the *curve* or *@radius* parameter is given, an arc can be drawn as part of the wire segment (see the detailed description in the WIRE command).

On completion of the SPLIT command, the segments are automatically rejoined if they are in line unless the command

```
SET OPTIMIZING OFF;
```

has previously been given, or the wire has been clicked at the same spot twice. In this case the split points remain and can be used, for example, to **reduce the width of a segment**. This is achieved by selecting the SPLIT command, marking the part of the wire which is to be reduced with two mouse clicks, and using the command

```
CHANGE WIDTH width
```

The segment is then clicked on to complete the change.

TECHNOLOGY

Function

Defines the possible *technology* parts of a device name.

Syntax

```
TECHNOLOGY name ..;
TECHNOLOGY -name ..;
TECHNOLOGY -* ..;
```

See also PACKAGE, ATTRIBUTE

This command is used in the device editor mode to define the possible *technology* parts of a device name. In the schematic or board editor the TECHNOLOGY command behaves exactly like "CHANGE TECHNOLOGY".

Exactly one of the names given in the TECHNOLOGY command will be used to replace the `'*'` in the device set name when an actual device is added to a schematic. The term *technology* stems from the main usage of this feature in creating different variations of the same basic device, which all have the same schematic symbol(s), the same package and the same pin/pad connections. They only differ in a part of their name, which for the classic TTL devices is related to their different technologies, like "L", "LS" or "HCT".

The TECHNOLOGY command can only be used if a package variant has been selected with the PACKAGE command.

If no `'*'` character is present in the device set name, the technology will be appended to the device set name to form the full device name. Note that the technology is processed before the package variant, so if the device set name contains neither a `'*'` nor a `'?'` character, the resulting device name will consist of *device_set_name*+*technology*+*package_variant*.

The names listed in the TECHNOLOGY command will be added to an already existing list of technologies for the current device. Starting a name with `'-'` will remove that name from the list of technologies. If a name shall begin with `'-'`, it has to be enclosed in single quotes. Using `-*` removes all technologies.

Only ASCII characters in the range 33..126 may be used in technologies (lowercase characters will be converted to uppercase).

The special "empty" technology can be entered as two single quotes (`''`, an empty string).

Note that the Technologies dialog contains all technologies from all devices in the loaded library, with the ones referenced by the current device checked.

Example

In a device named "`74*00`" the command

```
TECHNOLOGY -* '' L LS S HCT;
```

would first remove any existing technologies and then create the individual technology variants

```
7400
74L00
74LS00
74S00
74HCT00
```

TEXT

Function

Adds text to a drawing.

Syntax

```
TEXT any_text [orientation] ▯..
TEXT 'any_text' [orientation] ▯..
```

Mouse keys

Center selects the layer.
Right rotates the text.
Shift+Right reverses the direction of rotating.

See also CHANGE, MOVE, MIRROR, PIN, ROTATE, ATTRIBUTE

The TEXT command is used to add text to a library element or drawing. When entering several texts it is not necessary to invoke the command each time, as the text command remains active after placing text with the mouse.

Orientation

The orientation of the text may be defined by the TEXT command (orientation) using the usual definitions as listed in the ADD command (R0, R90 etc.). The right mouse button will change the rotation of the text and the center mouse button will change the current layer.

Text is always displayed so that it can be read from in front or from the right - even if rotated. Therefore after every two rotations it appears the same way, but the origin has moved from the lower left to the upper right corner. Remember this if a text appears to be unselectable.

The reading direction for vertical texts can be changed from "up" to "down" in the user interface dialog.

If you want to have text that is printed "upside down", you can set the "Spin" flag for that text.

Special Characters

If the text contains several successive blanks or a semicolon, the whole string has to be enclosed in single quotes. If the text contains single quotes then each one itself has to be enclosed in single quotes. If apostrophes are required in the text, each must be enclosed in single quotes.

Key Words

If the TEXT command is active and you want to type in a text that contains a string that can be mistaken for a command (e.g. "red" for "REDO") then this string has to be enclosed in single quotes.

Text Height

The height and thickness of characters can be changed with the CHANGE commands:

```
CHANGE SIZE text_size •..
CHANGE RATIO ratio •..
CHANGE LINEDISTANCE value •..
```

Maximum text height: 2 inches
Maximum text thickness: 0.51602 inch (13.1 mm)
Ratio: 0...31 (% of text height)
Line distance: 0...250 (% of text height).

Text Font

Texts can have three different fonts:

`Vector`	the program's internal vector font

`Proportional`	a proportional pixel font (usually 'Helvetica')
`Fixed`	a monospaced pixel font (usually 'Courier')

The text font can be changed with the CHANGE command:

```
CHANGE FONT VECTOR|PROPORTIONAL|FIXED ..
```

The program makes great efforts to output texts with fonts other than `Vector` as good as possible. However, since the actual font is drawn by the system's graphics interface, `Proportional` and `Fixed` fonts may be output with different sizes and/or lengths.

If you set the option "Always vector font" in the user interface dialog, all texts will always be displayed and printed using the builtin vector font, independent from the settings of the individual texts and for each drawing. This option is useful if the system doesn't display the other fonts correctly.
When creating a new board or schematic, the current setting of this option is stored in the drawing file. This makes sure that the drawing will be printed with the correct setting if it is transferred to somebody else who has a different setting of this option.
You can use the `SET VECTOR_FONT OFF|ON` command to change the setting in an existing board or schematic drawing. This means you can decide for the current drawing wether it shall always be displayed in vector font or not.

When creating output files with the CAM Processor, texts will always be drawn with `Vector` font. Other fonts are not supported.

If a text with a font other than `Vector` is subtracted from a signal polygon, only the surrounding rectangle is subtracted. Due to the above mentioned possible size/length problems, the actually printed font may exceed that rectangle. Therefore, if you need to subtract a text from a signal polygon it is recommended that you use the `Vector` font.

The *Ratio* parameter has no meaning for texts with fonts other than `Vector`.

Text Alignment

The text alignment defines where the origin shall be put within the text. There are nine different possible alignments, which consist of combinations of the keywords `left`, `bottom`, `center`, `right` and `top`. These keywords can be given in any sequence, but only the last one of left/right and top/bottom will be taken into account for the respective direction. The center keyword only applies to those directions where no other keyword has been given. The default is left and bottom.

`CHANGE ALIGN TOP CENTER; TEXT 'ABC' `	draws the text ABC with the origin at its top/center

Character Sets

Only the characters with ASCII codes below 128 are guaranteed to be printed correctly. Any characters above this may be system dependent and may yield different results with the various fonts.

Text Variables

Special texts in a symbol or package drawing, marked with the `'>'` character, will be replaced with actual values in a board or schematic:

`>NAME`	Component name (ev.+gate name) 1)
`>VALUE`	Comp. value/type 1)
`>PART`	Component name 2)
`>GATE`	Gate name 2)
`>XREF`	Part cross-reference 2)
`>CONTACT_XREF`	Contact cross-reference 2)
`>ASSEMBLY_VARIANT`	Name of the current assembly variant
`>DRAWING_NAME`	Drawing name
`>LAST_DATE_TIME`	Time of the last modification
`>PLOT_DATE_TIME`	Time of the plot creation
`>SHEETNR`	Sheet number of a schematic 3)
`>SHEETS`	Total number of sheets of a schematic 3)
`>SHEET`	equivalent to ">SHEETNR/>SHEETS" 3)

1) Only for package or symbol
2) Only for symbol
3) Only for symbol or schematic

The format in which a part cross-reference is displayed can be controlled through the "Xref part format" string, which is defined in the "Options/Set/Misc" dialog, or with the SET command. The following placeholders are defined, and can be used in any order:

`%S`	the sheet number
`%C`	the column on the sheet
`%R`	the row on the sheet

The default format string is `"/%S.%C%R"`. Apart from the defined placeholders you can also use any other ASCII characters.

Attributes

If a symbol or package drawing shall display an attribute of the actual part or element, a text with the name of that attribute, marked with the `'>'` character, can be used. By default, only the actual value of the given attribute will be displayed. If the attribute name is followed by one of the special characters `'='`, `'~'` or `'!'`, the actual display is as follows:

`>ABC`	`123`
`>ABC=`	`ABC = 123`
`>ABC~`	`ABC`
`>ABC!`	*nothing*

Note that for each attribute name there should be only one such text in any given symbol or package! If there is more than one such text in a symbol or package that all reference the same attribute name, only one of them will be displayed when the part using this symbol or package is smashed.

Overlined text

Text can be *overlined*, which is useful for instance for the names of inverted signals ("active low", see also NET, BUS and PIN). To do so, the text needs to be preceded with an exclamation mark (`'!'`), as in

```
!RESET
```

which would result in

$\overline{\texttt{RESET}}$

This is not limited to signal names, but can be used in any text. It is also possible to overline only part of a text, as in

```
!RST!/NMI
R/!W
```

which would result in

$\overline{\texttt{RST}}\texttt{/NMI}$

$\texttt{R/}\overline{\texttt{W}}$

Note that the second exclamation mark indicates the end of the overline. There can be any number of overlines in a text. If a text shall contain an exclamation mark that doesn't generate an overline, it needs to be escaped by a backslash. In order to keep the need for escaping exclamation marks at a minimum, an exclamation mark doesn't start an overline if it is the last character of a text, or if it is immediately followed by a blank, another exclamation mark, a double or single quote, or by a right parenthesis, bracket or brace. Any non-escaped exclamation mark or comma that appears after an exclamation mark that started an overline will end the overline (the comma as an overline terminator is necessary for busses).

UNDO

Function

Cancels previous commands.

Syntax

```
UNDO
UNDO LIST
```

Keyboard

`F9: UNDO` execute the UNDO command. `Alt+BS: UNDO`

See also REDO, SET, Forward&Back Annotation

The UNDO command allows you to cancel previously executed commands. This is especially useful if you have deleted things by accident. Multiple UNDO commands cancel the corresponding number of commands until the last EDIT, OPEN or REMOVE command is reached.

The UNDO command uses up memory space. If you are short of this you can switch off this function with the SET command

```
SET UNDO_LOG OFF;
```

UNDO/REDO is completely integrated within Forward&Back Annotation.

UNDO buffer dialog

The option LIST in the UNDO command opens a dialog that contains the entire contents of the undo buffer. You can navigate through the list of undo/redo steps by click&dragging the list delimiter, or by directly clicking on any given step you wish to go back or forward to. If there are several steps between the current delimiter position and the clicked list item, all steps in between will be executed in the proper sequence. Going upward in the list means doing UNDO, going downward results in REDO.
The icon at each list item indicates in which drawing this particular command has been executed.

If you confirm this dialog with "OK", the drawing will be left in the condition as selected in the list. If you cancel the dialog, it will be restored to the condition it had before the dialog was opened.

CAUTION: this is a very powerful tool! By going all the way back in the UNDO list (which can be done with a single mouse click) and executing any new command, the undo buffer will be truncated at that point, and there is no way back! So use this with care!

UPDATE

Function

Updates library objects.

Syntax

```
UPDATE
UPDATE;
UPDATE library_name..;
UPDATE package_name@library_name..;
UPDATE +@ | -@ [library_name..];
UPDATE old_library_name = new_library_name;
```

See also ADD, REPLACE

The UPDATE command checks the parts in a board or schematic against their respective library objects and automatically updates them if they are different. If UPDATE is invoked from the library editor, the packages within the loaded library will be updated from the given libraries.

If you activate the UPDATE command without a parameter, a file dialog will be presented to select the library from which to update.

If one ore more libraries are given, only parts from those libraries will be checked. The library names can be either a plain library name (like "ttl" or "ttl.lbr") or a full file name (like "/home/mydir/myproject/ttl.lbr" or "../lbr/ttl").

If a `library_name` contains blanks, it needs to be enclosed in single quotes.

Update in a board or schematic

If the command is terminated with a `';'`, but has no parameters, all parts will be checked.

If the first parameter is `'+@'`, the names of the given libraries (or all libraries, if none are given) will get a `'@'` character appended, followed by a number. This can be used to make sure the libraries contained in a drawing will not be modified when a part from a newer library with the same name is added to the drawing. Library names that already end with a `'@'` character followed by a number will not be changed.

If the first parameter is `'-@'`, the `'@'` character (followed by a number) of the given libraries (or all libraries, if none are given) will be stripped from the library name. This of course only works if there is no library with that new name already in the drawing.

Please note that "UPDATE +@;" followed by "UPDATE -@;" (and vice versa) does not necessarily result in the original set of library names, because the sequence in which the names are processed depends on the sequence in which the libraries are stored in the drawing file.

The libraries stored in a board or schematic drawing are identified only by their base name (e.g. "ttl"). When considering whether an update shall be performed, only the base name of the library file name will be taken into account. Libraries will be searched in the directories specified under "Libraries" in the directories dialog, from left to right. The first library of a given name that is found will be taken. Note that the library names stored in a drawing are handled case insensitive. It does not matter whether a specific library is currently "in use". If a library is not found, no update will be performed for that library and there will be no error message.

Using the UPDATE command in a schematic or board that are connected via active Forward&Back Annotation will act on both the schematic and the board.

At some point you may need to specify whether gates, pins or pads shall be mapped by their names or their coordinates. This is the case when the respective library objects have been renamed or moved. If too many modifications have been made (for example, if a pin has been both renamed and moved) the automatic update may not be possible. In that case you can either do the library modification in two steps (one for renaming, another for moving), or give the whole library object a different name.

When used with `old_library_name = new_library_name` (note that there has to be at least one blank before and after the `'='` character), the UPDATE command locates the library named *old_library_name* in the current board or schematic, and updates it with the contents of *new_library_name*. Note that *old_library_name* must be the pure library name, without any path, while *new_library_name* may be a full path name. If the update was performed successfully, the library in the current board/schematic file will also be renamed accordingly - therefore this whole operation is, of course, only possible if *new_library_name* has not yet been used in the current board or schematic.

Note: You should always run a Design Rule Check (DRC) and an Electrical Rule Check (ERC) after a library update has been performed in a board or a schematic!

Update in a library

The update in a library replaces all packages within that library with the versions from the given libraries.

By specifying the package name (package_name@library_name) you can have only a specific package be replaced.

USE

Function

Marks a library for use.

Syntax

```
USE
USE -*;
USE library_name..;
```

See also ADD, REPLACE

The USE command marks a library for later use with the ADD or REPLACE command.

If you activate the USE command without a parameter, a file dialog will appear that lets you select a library file. If a path for libraries has been defined in the "Options/Directories" dialog, the libraries from the first entry in this path are shown in the file dialog.

The special parameter `-*` causes all previously marked libraries to be dropped.

`library_name` can be the full name of a library or it can contain wildcards. If `library_name` is the name of a directory, all libraries from that directory will be marked.

The suffix `.lbr` can be omitted.

Note that when adding a device or package to a drawing, the complete library information for that object is copied into the drawing file, so that you don't need the library for changing the drawing later.

Changes in a library have no effect on existing drawings. See the UPDATE command if you want to update parts from modified libraries.

Using Libraries via the Control Panel

Libraries can be easily marked for use in the Control Panel by clicking on their activation icon (which changes its color to indicate that this library is being used), or by selecting "Use" from the library's context menu. Through the context menu of the "Libraries" entry in the Control Panel it is also possible to use *all* of the libraries or *none* of them.

Used Libraries and Projects

The libraries that are currently in use will be stored in the project file (if a project is currently open).

Examples

`USE`	opens the file dialog to choose a library
`USE -*;`	drops all previously marked libraries
`USE demo trans*;`	marks the library demo.lbr and all libraries with names matching trans*.lbr
`USE -* /eagle/lbr;`	first drops all previously marked libraries and then marks all libraries from the directory /eagle/lbr

VALUE

Function

Displays and changes values.

Syntax

```
VALUE ▯..
VALUE value ▯..
VALUE name value ..
VALUE ON;
VALUE OFF;
```

See also NAME, SMASH, VARIANT

In Boards and Schematics

Elements can be assigned a value, e.g. '1k' for a resistor or '10uF' for a capacitor. This is done with the VALUE command. The command selects an element and opens a popup menu that allows you to enter or to change a value.

If you type in a value before you select an element, then all of the subsequently selected elements receive this value. This is very useful if you want for instance a number of resistors to have the same value.

If the parameters name and value are specified, the element name gets the specified value.

The VALUE command can only be used in the default assembly variant. If you want to change the value of another assembly variant, you need to use the VARIANT command.

Example

```
VALUE R1 10k R2 100k
```

In this case more than one element has been assigned a value. This possibility can be used in script files:

```
VALUE R1   10k \
      R2  100k \
      R3  5.6k \
      C1  10uF \
      C2  22nF \
      ...
```

The '\' prevents the following line from being mistaken for an EAGLE key word.

In Device Mode

If the VALUE command is used in the device edit mode, the parameters ON and OFF may be used:

On: Permits the actual value to be changed in the schematic.

Off: Automatically enters the actual device name into the schematic (e.g.74LS00N). The user can only modify this value after a confirmation.

VARIANT

Function

Manages assembly variants.

Syntax

```
VARIANT
VARIANT name part_name [NO]POPULATE [ value [ technology ] ];
VARIANT [ + | - ] name;
```

See also VALUE, TECHNOLOGY

By default all parts of the schematic are populated on the board (provided they have a package). However, sometimes different variants of a design my require that some parts are not populated, or that they have different values or technologies than the default. The VARIANT command allows you to define which parts are actually populated in a given assembly variant, and to give them particular values and technologies.

name is the name of the variant. It is treated case-insensitive and must be enclosed in single quotes if it contains blanks or '+' or '-'.

If *part_name* followed by the keyword `POPULATE` or `NOPOPULATE` is given, a variant of the given *name* is created for that part, in which it will be marked as either "populated" or "not populated".

Parts that are not populated in the current assembly variant are indicated with an X drawn over their entire bounding rectangle in the schematic. In the board anything that is related to actually placing the part on the board (like placeplan, names, values etc.) is not drawn in such a case.

The optional *value* and *technology* (which is only applicable in a schematic) can be used to further refine the variant. A value may only be given if the part's device set has its "user value" parameter enabled. If only the technology shall be specified without using a different value, an empty string (`''`) can be entered for the value.

A new variant can be created by preceding `name` with a `'+'`. If a variant with that name already exists, nothing happens.

If the variant name is preceded with a `'-'`, the given variant will be deleted. If `name` is `'*'`, all variants will be deleted. Unless this command is used in a script, a confirmation prompt will ask the user whether this action should really be taken.

Giving only a variant name will switch the whole project to that variant. This means that all "populate" flags, values and technologies will appear as specified in that variant for each part. Using an empty string (`''`) here switches to the default assembly variant, which the same as if there were no variants at all. Note that when loading a drawing it is always in its default state, with no assembly variant selected.

If used without any parameters, a dialog will open that allows you to manage all assembly variants.

The name of the current assembly variant can be displayed by using the text variable `>ASSEMBLY_VARIANT`.

The commands ADD, CHANGE PACKAGE | TECHNOLOGY, REPLACE, UPDATE and VALUE can only be used if no assembly variant is active.

The COPY command doesn't copy assembly variants.

VIA

Function

Adds vias to a board.

Syntax

```
VIA ['signal_name'] [diameter] [shape] [layers] [flags] ..
```

See also SMD, CHANGE, DISPLAY, SET, PAD, Design Rules

When the VIA command is active, a via symbol is attached to the cursor. Pressing the left mouse button places a via at the current position. The via is added to a signal if it is placed on an existing signal wire. If you try to connect different signals, EAGLE will ask you if you really want to connect them.

Signal name

The `signal_name` parameter is intended mainly to be used in script files that read in generated data. If a `signal_name` is given, all subsequent vias will be added to that signal, and no automatic checks will be performed.
This feature should be used with great care because it could result in short circuits, if a via is placed in a way that it would connect wires belonging to different signals. Please run a Design Rule Check after using the VIA command with the `signal_name` parameter!

Via diameter

Entering a number changes the diameter of the via (in the actual unit) and the value remains in use for further vias. Via diameters can be up to 0.51602 inch (13.1 mm).

The drill diameter of the via is the same as the diameter set for pads. It can be changed with

```
CHANGE DRILL diameter
```

Shape

A via can have one of the following shapes:

Square
Round
Octagon

These shapes only apply to the outer layers (Top and Bottom). In inner layers the shape is always "round".

Vias generate drill symbols in the Drills layer and the solder stop mask in the tStop/bStop layers.

Like the diameter, the via shape can be entered while the VIA command is active, or it can be changed with the CHANGE command. The shape then remains valid for the next vias and pads.

Note that the actual shape and diameter of a via will be determined by the Design Rules of the board the via is used in.

Layers

The `layers` parameter defines the layers this via shall cover. The syntax is `from-to`, where 'from' and 'to' are the layer numbers that shall be covered. For instance `2-7` would create a via that goes from layer 2 to layer 7 (`7-2` would have the same meaning). If that exact via is not available in the layer setup of the Design Rules, the next longer via will be used (or an error message will be issued in case no such via can be set).

Flags

The following *flags* can be used to control the appearance of a via:

`STOP`	always generate solder stop mask

By default a via with a drill diameter that is less than or equal to the value of the Design Rules parameter "Masks/Limit" will not have a solder stop mask. The above `STOP` flag can be used to force a solder stop mask for a via.

WINDOW

Function

Zooms in and out of a drawing.

Syntax

```
WINDOW;
WINDOW □;
WINDOW □ □;
WINDOW □ □ □
WINDOW scale_factor
WINDOW FIT
WINDOW LAST
```

Mouse keys

Left&Drag defines a rectangular window (shortcut for "`□ □;`").

Keyboard

`Alt+F2: WINDOW FIT` Fit drawing on the screen
`F2: WINDOW;` Redraw screen
`F3: WINDOW 2` Zoom in by a factor of 2
`F4: WINDOW 0.5` Zoom out by a factor of 2
`F5: WINDOW (@);` Cursor pos. is new center (if a command is active)

The WINDOW command is used to zoom in and out of the drawing and to change the position of the drawing on the screen. The command can be used with up to three mouse clicks. If there are fewer, it must be terminated with a semicolon.

Refresh screen

If you use the WINDOW command followed by a semicolon, EAGLE redraws the screen without changing the center or the scale. This is useful if error messages cover part of the drawing.

New center

The WINDOW command with one point causes that point to become the center of a new screen display of the drawing. The scaling of the drawing remains the same. You can also use the sliders of the working area to move the visible area of the drawing. The function key F5 causes the current position of the cursor to be the new center.

Corner points

The WINDOW command with two points defines a rectangle with the specified points at opposite corners. The rectangle expands to fill the screen providing a close-up view of the specified portion of the drawing.

New center and zoom

You can use the WINDOW command with three points. The first point defines the new center of the drawing and the display becomes either larger or smaller, depending on the ratios of the spacing between the other points. In order to zoom in, the distance between point 1 and point 3 should be greater than the distance between point 1 and 2; to zoom out place point 3 between points 1 and 2.

Zoom in and out

```
WINDOW 2;
```

Makes the elements appear twice as large.

```
WINDOW 0.5;
```

Reduces the size of the elements by a factor of two.

You can specify an integer or real number as the argument to the WINDOW command to scale the view of the drawing by the amount entered. The center of the window remains the same.

The whole drawing

```
WINDOW FIT;
```

fits the entire drawing on the screen.

Back to the previous window

```
WINDOW LAST;
```

switches back to the previous window selection. A window selection is stored by every WINDOW command, except for zoom-only WINDOW commands and modifications of the window selection with the mouse.

Very large zoom factors

By default the maximum zoom factor is limited in such a way that an area of 1mm (about 40mil) in diameter will be shown using the full editor window. If you need to zoom in further, you can uncheck "Options/User interface/Limit zoom factor" and will then be able to zoom in all the way until the finest editor grid can be seen.

When zooming very far into a drawing, the following things may happen:

- Texts that are not using the vector font may not be shown if they are larger than the editor window.
- Circles and Arcs are approximated and therefore may not appear at their exact location (especially if they have a very small width).
- Whether or not the finest grid will be visible when zooming all the way in depends on your screen resolution, the editor window size and the value of "Options/Set/Misc/Min. visible grid size".

Parameter Aliases

Parameter aliases can be used to define certain parameter settings to the WINDOW command, which can later be referenced by a given name. The aliases can also be accessed by clicking on the "WINDOW Select" button and holding the mouse button pressed until the list pops up. A right click on the button also pops up the list.

The syntax to handle these aliases is:

`WINDOW =` ***`name parameters`***
: Defines the alias with the given *name* to expand to the given *parameters*. The *name* may consist of any number of letters, digits and underlines, and is treated case insensitive. It must begin with a letter or underline and may not be one of the option keywords.

`WINDOW =` ***`name`*** **`@`**
: Defines the alias with the given *name* to expand to the current window selection.

`WINDOW = ?`
: Asks the user to enter a name for defining an alias for the current window settings.

`WINDOW =` ***`name`***
: Allows the user to select a window that will be defined as an alias under the given *name*.

`WINDOW =` ***`name`*** **`;`**
: Deletes the alias with the given *name*.

`WINDOW` ***`name`***
: Expands the alias with the given *name* and executes the WINDOW command with the resulting set of parameters. The *name* may be abbreviated and there may be other parameters before and after the alias (even other aliases). Note that in case *name* is an abbreviation, aliases have precedence over other parameter names of the command.

Example:

```
WINDOW = MyWindow (0 0) (4 3);
```

Defines the alias "MyWindow" which, when used as in

```
WINDOW myw
```

will zoom to the given window area. Note the abbreviated use of the alias and the case insensitivity.

WIRE

Function

Adds wires (tracks) to a drawing.

Syntax

```
WIRE ['signal_name'] [width] □ □..
WIRE ['signal_name'] [width] [ROUND | FLAT] □ [curve | @radius]
□..
```

Mouse keys

Center selects the layer.
Right changes the wire bend style (see SET Wire_Bend).
Shift+Right reverses the direction of switching bend styles.
Ctrl+Left when starting a wire snaps it to the next existing wire end point.
Ctrl+Right toggles between corresponding bend styles.
Ctrl+Left when placing a wire end point defines arc radius.

See also MITER, SIGNAL, ROUTE, CHANGE, NET, BUS, DELETE, RIPUP, ARC

The WIRE command is used to add wires (tracks) to a drawing. The wire begins at the first point specified and runs to the second. Additional points draw additional wire segments. Two mouse clicks at the same position finish the wire and a new one can be started at the position of the next mouse click.

Depending on the currently active wire bend, one or two wire segments will be drawn between every two points. The wire bend defines the angle between the segments and can be changed with the right mouse button (holding the Shift key down while clicking the right mouse button reverses the direction in which the bend styles are gone through, and the Ctrl key makes it toggle between corresponding bend styles).

Pressing the center mouse button brings up a popup menu from which you may select the layer into which the wire will be drawn.

The special keywords `ROUND` and `FLAT`, as well as the *curve* parameter, can be used to draw an arc (see below).

Starting a WIRE with the Ctrl key pressed snaps the starting point of the new wire to the coordinates of the closest existing wire. This is especially useful if the existing wire is off grid. It also adjusts the current width, layer and style to those of the existing wire. If the current bend style is 7 ("Freehand"), the new wire will form a smooth continuation of the existing wire.

Signal name

The `signal_name` parameter is intended mainly to be used in script files that read in generated data. If a `signal_name` is given, all subsequent wires will be added to that signal and no automatic checks will be performed.
This feature should be used with great care because it could result in short circuits, if a wire is placed in a way that it would connect different signals. Please run a Design Rule Check after using the WIRE command with the `signal_name` parameter!

Wire Width

Entering a number after activating the WIRE command changes the width of the wire (in the present unit) which can be up to 0.51602 inch (13.1 mm).

The wire width can be changed with the command

```
CHANGE WIDTH width 
```

at any time.

Wire Style

Wires can have one of the following *styles*:

- Continuous
- LongDash
- ShortDash
- DashDot

The wire style can be changed with the CHANGE command.

Note that the DRC and Autorouter will always treat wires as "Continuous", even if their style is different. Wire styles are mainly for electrical and mechanical drawings and should not be used on signal layers. It is an explicit DRC error to use a non-continuous wire as part of a signal that is connected to any pad.

Signals in Top, Bottom, and Route Layers

Wires (tracks) in the layers Top, Bottom, and ROUTE2...15 are treated as signals. If you draw a wire in either of these layers starting from an existing signal, then all of the segments of this wire belong to that signal (only if the center of the wire is placed exactly onto the center of the existing wire or pad). If you finish this drawing operation with a wire segment connected to a different signal, then EAGLE will ask you if you want to connect the two signals.

Note that EAGLE treats each wire segment as a single object (e.g. when deleting a wire).

When the WIRE command is active the center mouse button can be used to change the layer on which the wire is drawn.

Do not use the WIRE command for nets, buses, and airwires. See NET, BUS and SIGNAL.

Drawing Arcs

Wires and arcs are basically the same objects, so you can draw an arc either by using the ARC command, or by adding the necessary parameters to the WIRE command. To make a wire an arc it needs either the *curve* parameter, which defines the "curvature" of the arc, or the *@radius* parameter, which defines the radius of the arc (note the `'@'`, which is necessary to be able to tell apart *curve* and *radius*).

The valid range for *curve* is `]-360`..`+360[` (without the limits +-360), and its value means what part of a full circle the arc consists of. A value of `90`, for instance, would result in a `90°` arc, while `180` would give you a semicircle. Full circles cannot be created this way (for this use the CIRCLE command). Positive values for *curve* mean that the arc is drawn in a mathematically positive sense (i.e. counterclockwise). If *curve* is `0`, the arc is a straight line ("no curvature"), which is actually a wire. Note that in order to distinguish the *curve* parameter from the *width* parameter, it always has to be given with a sign (`'+'` or `'-'`), even if it is a positive value.

As an example, the command

```
WIRE (0 0) +180 (0 10);
```

would draw a semicircle from the point (0 0) to (0 10), in counterclockwise direction.

If a *radius* is given, the arc will have that radius. Just like the *curve* parameter, *radius* also must have a sign in order to determine the arcs orientation. For example, the command

```
WIRE (0 0) @+100 (0 200);
```

would draw a semicircle from the point (0 0) to (0 200) (with a radius of 100), in counterclockwise direction. Note that if the end point is more than twice the radius away from the start point, a straight line will be drawn.

The arc radius can also be defined by placing the wire end point with the `Ctrl` key pressed (typically at the center of the circle on which the arc shall lie). In that case the point is not taken as an actual end point, but is rather used to set the radius of an arc. You can then move the cursor around and place an arc with the given radius (the right mouse button together with `Ctrl` will toggle the arc's orientation). If you move the cursor more than twice the radius away from the start point, a straight line will be drawn.

In order to be able to draw any arc with the WIRE command (which is especially important for generated script files), the keywords `ROUND` and `FLAT` are also allowed in the WIRE command. Note, though, that these apply only to actual arcs (straight wires always have round endings). By default, arcs created with the WIRE command have round endings.

WRITE

Function

Saves the current drawing or library.

Syntax

```
WRITE;
WRITE name
WRITE @name
```

The WRITE command is used to save a drawing or library. If 'name' is entered, EAGLE will save the file under the new name.

The file name may also be entered with a pathname if it is to be saved in another directory. If no pathname is given, the file is saved in the project directory.

If the new name is preceded with a `@`, the name of the loaded drawing will also be changed accordingly. The corresponding board/schematic will then also be saved automatically under this name and the UNDO buffer will be cleared.

If WRITE is selected from the menu, a popup window will appear asking for the name to use (current drawing name is default). This name may be edited and accepted by clicking the OK button. Pressing the ESCAPE key or clicking the CANCEL button cancels the WRITE command.

To assure consistency for Forward&Back Annotation between board and schematic drawings, the WRITE command has the following additional functionality:

- when a board/schematic is saved under the same name, the corresponding schematic/board is also saved if it has been modified
- when a board/schematic is saved under a different name, the user will be asked whether he also wants to save the schematic/board under that different name
- saving a drawing under a different name does not clear the "modified" flag

Generating Output

- Printing
- CAM Processor
- Outlines data

Printing

The parameters for printing to the system printer can be modified through the following three dialogs:

- Printing a Drawing
- Printing a Text
- Printer Page Setup

See also PRINT

Printing a Drawing

If you enter the PRINT command without a terminating `';'`, or select **Print** from the context menu of a drawing's icon in the Control Panel, you will be presented a dialog with the following options:

Paper

Selects which paper format to print on.

Orientation

Selects the paper orientation.

Preview

Turns the print preview on or off.

Mirror

Mirrors the output.

Rotate

Rotates the output by 90°.

Upside down

Rotates the drawing by 180°. Together with **Rotate** the drawing is rotated by a total of 270°.

Black

Ignores the color settings of the layers and prints everything in black.

Solid

Ignores the fill style settings of the layers and prints everything in solid.

Scale factor

Scales the drawing by the given value.

Page limit

Defines the maximum number of pages you want the output to use. In case the drawing does not fit on the given number of pages, the actual scale factor will be reduced until it fits. The default value of `0` means no limit.

All

All sheets of the schematic will be printed (this is the default when selecting **Print** from the context menu of a schematic drawing's icon).

From...to

Only the given range of sheets will be printed.

This

Only the sheet that is currently being edited will be printed (this is the default when using the PRINT command from a schematic editor window).

Printer...

Invokes the system printer dialog, which enables you to choose which printer to use and to set printer specific parameters.

PDF...

Creates a PDF (Portable Document Format) file with the given print settings.

The remaining options are used for the page setup.

Printing a Text

If you select **Print** from the context menu of a text file's icon in the Control Panel, or from the **File** menu of the Text Editor, you will be presented a dialog with the following options:

Wrap long lines

Enables wrapping lines that are too long to fit on the page width.

Printer...

Invokes the system printer dialog, which enables you to choose which printer to use and to set printer specific parameters.

PDF...

Creates a PDF (Portable Document Format) file with the given print settings.

The remaining options are used for the page setup.

Printer Page Setup

The Print dialog provides several options that are used to define how a drawing or text shall be placed on the paper.

Border

Defines the left, top, right and bottom borders. The values are either in millimeters or inches, depending on which unit results in fewer decimals.

The default border values are taken from the printer driver, and define the maximum drawing area your particular printer can handle. You can enter smaller values here, but your printer hardware may or may not be able to print arbitrarily close to the paper edges.

After changing the printer new hardware minimums may apply and your border values may be automatically enlarged as necessary to comply with the new printer. Note that the values will not be decreased automatically if a new printer would allow smaller values. To get the smallest possible border values you can enter `0` in each field, which will then be limited to the hardware minimum.

Calibrate

If you want to use your printer to produce production layout drawings, you may have to calibrate your printer to achieve an exact 1:1 reproduction of your layout.

The value in the **X** field is the calibration factor to use in the print head direction, while the value in the **Y** field is used to calibrate the paper feed direction.

IMPORTANT NOTE: When producing production layout drawings with your printer, always check the final print result for correct measurements!

The default values of `1` assume that the printer produces exact measurements in both directions.

Aligment

Defines the vertical and horizontal alignment of the drawing on the paper.

Caption

Activates the printing of a caption line, containing the time and date of the print as well as the file name.

If the drawing is mirrored, the word "mirrored" will appear in the caption, and if the scale factor is not `1.0` it will be added as **f=...** (the scale factor is given with 4 decimal digits, so even if **f=1.0000** appears in the caption the scale factor will not be *exactly* `1.0`).

CAM Processor

The CAM Processor allows you to output any combination of layers to a device or file.

The following help topics lead you through the necessary steps from selecting a data file to configuring the output device:

- Select the data file
- Select the output device type
- Select the output file
- Select the plot layers
- Adjust the device parameters
- Adjust the flag options

The CAM Processor allows you to combine several sets of parameter settings to form a CAM Processor Job, which can be used to produce a complete set of output files with a single click of a button.

See also printing to the system printer

Main CAM Menu

The *Main CAM Menu* is where you select which file to process, edit drill rack and aperture wheel files, and load or save job files.

File

Open	Board... open a board file for processing Drill rack... open a drill rack file for editing Wheel... open an aperture wheel file for editing Job... switch to an other job (or create a new one)
Save job...	save the current job
Close	close the CAM Processor window
Exit	exit from the program

Layer

Deselect all	deselect all layers

Show selected	show only the selected layers
Show all	show all layers

Window

Control Panel	switch to the Control Panel
1 Schematic - ...	switch to window number 1
2 Board - ...	switch to window number 2

Help

General help	opens a general help page
Contents	opens the help table of contents
CAM Processor	displays help for the CAM Processor
Job help	displays help about the Job mechanism
Device help	displays help about output devices

CAM Processor Job

A CAM Processor *Job* consists of several *Sections*, each of which defines a complete set of CAM Processor parameters and layer selections.

A typical CAM Processor job could for example have two sections, one that produces photoplotter data for the Top layer, and another that produces the data for the bottom layer.

Section

The *Section* selector shows the currently active job section. By pressing the button you can select any of the sections you have defined previously with the *Add* button.

Prompt

If you enter a text in this field, the CAM Processor will prompt you with this message before processing that particular job section. For example you might want to change the paper in your pen plotter for each plot, so the message could be "Please change paper!". Each job section can have its own prompt message, and if there is no message the section will be processed immediately.

Add

Click on the *Add* button to add a new section to the job. You will be asked for the name of that new job section. The new job section will be created with all parameters set to the values currently shown in the menu.
Please note that if you want to create a new job section, you should **first add** that new section and **then modify** the parameters. Otherwise, if you first modify the parameters of the current section and then add a new section, you will be prompted to confirm whether the modifications to the current section shall be saved or not.

Del

Use the *Del* button to delete the current job section. You will be prompted to confirm whether you really want to delete that section.

Process Section

The *Process Section* button processes the current job section, as indicated in the *Section* selector.

Process Job

The *Process Job* button processes the entire job by processing each section in turn, starting with the first section. What happens is the same as if you would select every single section with the *Section* selector and press the *Process Section* button for each section - just a lot more convenient!

Output Device

The *Output Device* defines the kind of output the CAM Processor is to produce. You can select from various device types, like photo plotters, drill stations etc.

Device

Clicking on the button of the Device selector opens a list of all available output devices.

Scale

On devices that can scale the output you can enter a scaling factor in this field. Values larger than `1` will produce an enlarged output, values smaller than `1` will shrink the output.

You can limit the size of the output to a given number of pages by entering a negative number in the Scale field. In that case the default scale factor will be 1.0 and will be decreased until the drawing just fits on the given number of pages. For example, entering "-2" into this field will produce a drawing that does not exceed two pages. Please note that for this mechanism to work you will have to make sure that the page width and height is set according to your output device. This setting can be adjusted in the Width and Height fields or by editing the file eagle.def.

File

You can either enter the name of the output file directly into this field, or click on the File button to open a dialog for the definition of the output file.
If you want to derive the output filename from the input data file, you can enter a partial filename (at least an extension, e.g. `.gbr`), in which case the rest of the filename will be taken from the input data filename.

Wheel

You can either enter the name of the aperture wheel file directly into this field, or click on

the Wheel button to open a file dialog to select from.
If you want to derive the output filename from the input data file, you can enter a partial filename (at least an extension, e.g. `.whl`), in which case the rest of the filename will be taken from the input data filename.

Rack

You can either enter the name of the drill rack file directly into this field, or click on the Rack button to open a file dialog to select from.
If you want to derive the output filename from the input data file, you can enter a partial filename (at least an extension, e.g. `.drl`), in which case the rest of the filename will be taken from the input data filename. Some drill devices (like EXCELLON, for instance) can automatically generate the necessary drill definitions, in which case this field is not present.

Device Parameters

Depending on the type of output device you have selected, there are several device specific parameters that allow you to adjust the output to your needs:

- Aperture Wheel File
- Aperture Emulation
- Aperture Tolerances
- Drill Rack File
- Drill Tolerances
- Offset
- Page Size
- Pen Data

Aperture Wheel File

A photoplotter usually needs to know which *apertures* are assigned to the codes used in the output file. These assignments are defined in an *Aperture Wheel File*.

Examples

```
D010    round     0.004
D040    square    0.004
D100    rectangle 0.060 x 0.075
D104    oval      0.030 x 0.090
D110    draw      0.004
```

Note that the file may contain several apertures that share the same D-code, as long as all of these have a type from draw or round, and have the same size. This can be used to map apertures that effectively result in the same drawing to a common D-code.

Aperture Emulation

If the item "Apertures" is selected, apertures not available are emulated with smaller apertures. If this item is not selected, no aperture emulation will be done at all.

Please note that aperture emulation can cause very long plot times (costs!).

Aperture Tolerances

If you enter tolerances for draw and/or flash apertures the CAM Processor uses apertures within the tolerances, provided the aperture with the exact value is not available.

Tolerances are entered in percent.

Please be aware that your design rules might not be kept when allowing tolerances!

Drill Rack File

If a drill station driver can't automatically generate the necessary drill definitions, it needs to know which *drill diameters* are assigned to the codes used in the output file. These assignments are defined in a *Drill Rack File*.

This file can be generated with the help of a User Language Program called drillcfg.ulp, that is stored in your EAGLE's ULP directory. Use the RUN command to start it.

Example

```
T01   0.010
T02   0.016
T03   0.032
T04   0.040
T05   0.050
T06   0.070
```

Drill Tolerances

If you enter tolerances for drills the CAM Processor uses drill diameters within the tolerances, provided the drill with the exact value is not available.

Tolerances are entered in percent.

Offset

Offset in x and y direction (inch, decimal number).

Can be used to position the origin of plotters at the lower left corner.

Printable Area

Height

Printable area in Y direction (inch).

Width

Printable area in X direction (inch).

Please note that the CAM Processor divides a drawing into several parts if the rectangle which includes all objects of the file (even the ones not printed) doesn't fit into the printable area.

Pen Data

Diameter

Pen diameter in mm. Is used for the calculation of lines when areas have to be filled.

Velocity

Pen velocity in cm/s for pen plotters which can be adjusted to different speeds.

The plotter default speed is selected with the value 0.

Defining Your Own Device Driver

The drivers for output devices are defined in the text file eagle.def. There you find details on how to define your own driver. It is advisable to copy the whole section of an existing driver of the same device category and to edit the parameters which are different.

Please use a text editor which doesn't place control characters into the file.

Output File

The *Output File* contains the data produced by the CAM Processor.

The following file names are commonly used:

```
-------------------------------------------------------
File   Layers                Meaning
-------------------------------------------------------
*.cmp  Top, Via, Pad         Component side
*.ly2  Route2, Via, Pad      Inner signal layer
*.ly3  Route3, Via, Pad      Inner signal layer
...                          ...
*.sol  Bot, Via, Pad         Solder side
*.plc  tPl, Dim, tName,      Silkscreen comp. side
*.pls  bPl, Dim, bName,      Silkscreen solder side
*.stc  tStop                 Solder stop mask comp. side
*.sts  bStop                 Solder stop mask sold. side
*.drd  Drills, Holes         Drill data for NC drill st.
-------------------------------------------------------
```

Placeholders

The output file name can either be entered directly, or can be dynamically composed using *placeholders*. A placeholder consists of a percentage character (`'%'`) followed by a letter. The following placeholders are defined:

`%D{xxx}`	a string that is inserted only into the data file name

`%E`	the loaded file's extension (without the `'.'`)
`%H`	the user's home directory
`%I{xxx}`	a string that is inserted only into the info file name
`%L`	the layer range for blind&buried vias (see below)
`%N`	the loaded file's name (without path and extension)
`%P`	the loaded file's directory path (without file name)
`%%`	the character `'%'`

For example, the output file definition

```
%N.cmp%I{.info}
```

would create *`boardname`*`.cmp` for the data file and *`boardname`*`.cmp.info` for the info file (in case the selected output device generates an info file).

Drill data with blind&buried vias

If the board contains blind or buried vias, the CAM Processor generates a separate drill file for each via length that is actually used in the board. The file names are built by adding the number of the start and end layer to the base file name, as in

`boardname``.drl.0104`

which would be the drill file for the layer stack 1-4. If you want to have the layer numbers at a different position, you can use the placeholder `%L`, as in

```
%N.%L.drl
```

which would result in

`boardname``.0104.drl`

The drill info file name is always generated without layer numbers, and any '.' before the `%L` will be dropped. Any previously existing files that would match the given drill file name pattern, but would not result from the current job, will be deleted before generating any new files. There will be one drill info file per job, which contains (amoung other information) a list of all generated drill data files.

Flag Options

Mirror

Mirror output. This option normally causes negative coordinates, therefore it should be used only if "pos. Coord." is selected, too.

Rotate

Rotate drawing by 90 degrees. This option normally causes negative coordinates, therefore it should be used only if "pos. Coord." is selected, too.

Upside down

Rotate the drawing by 180 degrees. Together with Rotate, the drawing is rotated by a total of 270 degrees. This option normally causes negative coordinates, therefore it should be used only if "pos. Coord." is selected, too.

pos. Coord

Offsets the output so that negative coordinates are eliminated and the drawing is referenced to the origin of the output device. This is advisable for devices which generate error messages if negative coordinates are detected.

Quickplot

Draft output which shows only the outlines of objects (subject to availability on the selected output device).

Optimize

Activates the optimization of the drawing sequence for plotters.

Fill pads

Pads will be filled. This function can be properly executed only with generic devices, like PostScript.
If this option is not selected, the drill holes of pads will be visible on the output.

Layers and Colors

Select the layer combination by clicking the check boxes in the *Layer* list.

If you have selected an output device that supports colors, you can enter the color number in the *Color* field of each layer.

The following layers and output file names are commonly used to create the output:

File	Layers	Meaning
*.cmp	Top, Via, Pad	Component side
*.ly2	Route2, Via, Pad	Inner signal layer
*.ly3	Route3, Via, Pad	Inner signal layer
...		...
*.sol	Bot, Via, Pad	Solder side
*.plc	tPl, Dim, tName,	Silkscreen comp. side
*.pls	bPl, Dim, bName,	Silkscreen solder side
*.stc	tStop	Solder stop mask comp. side
*.sts	bStop	Solder stop mask sold. side
*.drd	Drills, Holes	Drill data for NC drill st.

Outlines data

EAGLE can produce outlines data which can be used for milling prototype boards.

The User Language Program *outlines.ulp* implements the entire process necessary to do this. The following is a detailed description of what exactly has to be done to produce outlines data with EAGLE.

Preparing the board

Outlines data is produced by defining a POLYGON in the layer for which the outlines shall be calculated. This polygon must have the following properties:

- its name must be _OUTLINES_
- it must be the **only** object in the signal named _OUTLINES_
- its *Rank* must be `'6'`
- its *Width* must be the same as the diameter of the milling tool
- it must be large enough to cover the entire board area

If a polygon with these properties is present in your board, the RATSNEST command will calculate it in such a way that its *contours* correspond to the lines that have to be drawn by the milling tool to isolate the various signals from each other. The *fillings* of the calculated polygon define what has to be milled out if you want to completely remove all superfluous copper areas.

Extracting the data

The outlines data can be extracted from the board through a User Language Program. The *outlines.ulp* program that comes with EAGLE implements this entire process. If you want to write your own ULP you can use *outlines.ulp* as a starting point. See the help page for UL_POLYGON for details about how to retrieve the outlines data from a polygon object.

Milling tool diameter

The diameter of the milling tool (and thus the *Width* of the polygon) must be small enough to fit between any two different signals in order to be able to isolate them from each other. **Make sure you run a Design Rule Check (DRC) with all *Clearance* values for different signals set to at least the diameter of your milling tool!**

Non-zero values for the Isolate parameter can be used when working sequentially with different milling tool diameters in order to avoid areas that have already been milled.

Cleaning up

Make sure that you always delete the _OUTLINES_ polygon after generating the outlines data. Leaving this polygon in your drawing will cause short circuits since this special polygon does not adhere to the Design Rules!

Autorouter

The integrated Autorouter can be started from a board window with the AUTO command.

The Autorouter is also used as "Follow-me" router in the ROUTE command.

Please check your license to see whether you have access to the Autorouter module.

Design Checks

There are two integrated commands that allow you to check your design:

- Electrical Rule Check (ERC)
- Design Rule Check (DRC)

The ERC is performed in a schematic window, and checks the design for electrical consistency.

The DRC is performed in a board window, and checks the design for overlaps, distance violations etc.

Design Rules

Design Rules define all the parameters that the board layout has to follow.

The Design Rule Check checks the board against these rules and reports any violations.

The Design Rules of a board can be modified through the Design Rules dialog, which appears if the DRC command is selected without a terminating `';'`.

Newly created boards take their design rules from the file 'default.dru', which is searched for in the first directory listed in the "Options/Directories/Design rules" path. If no such file is present, the program's builtin default values apply.

Note regarding the values for **Clearance** and **Distance**: since the internal resolution of the coordinates is 1/10000mm, the DRC can only reliably report errors that are larger than 1/10000mm.

File

The *File* tab shows a description of the current set of Design Rules and allows you to *change* that description (this is strongly recommended if you define your own Design Rules). There are also buttons to *load* a different set of Design Rules from a disk file and to *save* the current Design Rules to disk.
Note that the Design Rules are stored within the board file, so they will be in effect if the board file is sent to a board house for production. The "Load..." and "Save as..." buttons are merely for copying a board's Design Rules to and from disk.

If the Design Rules have been modified, the name in the dialog's title will have trailing asterisk (`'*'`) to mark the Design Rules as modified. This mark will be removed once the Design Rules are explicitly written to disk, or a new set of Design Rules is loaded.

Layers

The *Layers* tab defines which signal layers the board actually uses, how thick the copper and isolation layers are, and what kinds of vias can be placed (note that this applies only to actual *vias*; so even if no via from layer 1 to 16 has been defined in the layer setup, *pads* will always be allowed).

The layer setup is defined by the string in the "Setup" field. This string consists of a sequence of layer numbers, separated by one of the characters `'*'` or `'+'`, where `'*'` stands for *core* material (also known as *FR4* or something similar) and `'+'` stands for *prepreg* (or any other kind of isolation material). The actual *core* and *prepreg* sequence has no meaning to EAGLE other than varying the color in the layer display at the top left corner of this tab (the actual multilayer setup always needs to be worked out with the board manufacturer). The vias are defined by enclosing a sequence of layers with `(...)`. So the setup string

```
(1*16)
```

would mean a two layer board, using layers 1 and 16 and vias going through the entire board (this is also the default value).
When building a multilayer board the setup could be something like

```
((1*2)+(15*16))
```

which is a four layer board with layer pairs 1/2 and 15/16 built on core material and vias drilled through them, and finally the two layer pairs pressed together with prepreg between them, and vias drilled all the way through the entire board.
Besides vias that go trough an entire layer stack (which are commonly referred to as *buried* vias in case they have no connection to the Top and Bottom layer) there can also be vias that are not drilled all the way through a layer stack, but rather end at a layer inside that stack. Such vias are known as *blind* vias and are defined in the "Setup" string by enclosing a sequence of layers with `[t:...:b]`, where *t* and *b* are the layers up to which that via will go from the top or bottom side, respectively. A possible setup with *blind* vias could be

```
[2:1+((2*3)+(14*15))+16:15]
```

which is basically the previous example, with two additional outer layers that are connected to the next inner layers by *blind* vias. It is also possible to have only one of the *t* or *b* parameters, so for instance

```
[2:1+((2*3)+(15*16))]
```

would also be a valid setup. Finally, *blind* vias are not limited to starting at the Top or Bottom layer, but may also be used in inner layer stacks, as in

```
[2:1+[3:2+(3*4)+5:4]+16:5]
```

A *blind* via from layer *a* to layer *b* also implements all possible *blind* vias from layer *a* to all layers between layers *a* and *b*, so

```
[3:1+2+(3*16)]
```

would allow *blind* vias from layer 1 to 2 as well as from 1 to 3.

Clearance

The *Clearance* tab defines the various minimum clearance values between objects in signal layers. These are usually absolute minimum values that are defined by the production process used and should be obtained from your board manufacturer.

The actual minimum clearance between objects that belong to different signals will also be influenced by the net classes the two signals belong to.

Note that a polygon in the special signal named _OUTLINES_ will be used to generate outlines data and as such will **not** adhere to these clearance values.

Distance

The *Distance* tab defines the minimum distance between objects in signal layers and the board dimensions, as well as that between any two drill holes. Note that only signals that are actually connected to at least one pad or smd are checked against the board dimensions. This allows edge markers to be drawn in the signal layer without generating DRC errors.

For compatibility with version 3.5x the following applies: If the minimum distance between copper and dimension is set to `0` objects in the Dimension layer will not be taken into account when calculating polygons (except for Holes, which are always taken into account). This also disables the distance check between copper and dimension objects.

Sizes

The *Sizes* tab defines the minimum width of any objects in signal layers and the minimum drill diameter. These are usually absolute minimum values that are defined by the production process used and should be obtained from your board manufacturer.
The actual minimum width of signal wires and drill diameter of vias will also be influenced by the Net Class the signal belongs to.

Restring

The *Restring* tab defines the width of the copper ring that has to remain after the pad or via has been drilled. Values are defined in percent of the drill diameter and there can be an absolute minimum and maximum limit. Restrings for pads can be different for the top, bottom and inner layers, while for vias they can be different for the outer and inner layers.
If the actual diameter of a pad (as defined in the library) or a via would result in a larger restring, that value will be used in the outer layers. Pads in library packages can have their diameter set to 0, so that the restring will be derived entirely from the drill diameter.

Shapes

The *Shapes* tab defines the actual shapes for smds and pads.
Smds are normally defined as rectangles in the library (with a "roundness" of 0), but if your design requires rounded smds you can specify the roundness factor here.
Pads are normally defined as octagons in the library (long octagons where this makes sense), and you can use the combo boxes to specify whether you want to have pads with the same shapes as defined in the library, or always square, round or octagonal. This can be set independently for the top and bottom layer.
If the "first" pad of a package has been marked as such in the library it will get the shape as defined in the third combo box (either round, square or octagonal, or no special shape).
The Elongation parameters define the appearance of pads with shape Long or Offset.

Supply

The *Supply* tab defines the Thermal isolation between pads and signal polygons.

Masks

The *Masks* tab defines the dimensions of solder stop and cream masks. They are given in percent of the smaller dimension of smds, pads and vias and can have an absolute minimum and maximum value.
Solder stop masks are generated for smds, pads and those vias that have a drill diameter that exceeds the given Limit parameter.
Cream masks are generated for smds only.

Misc

The *Misc* tab allows you to turn on a grid and angle check.

Cross-references

There are various methods that can be used to create cross-references in EAGLE schematic drawings. The following sections describe each of them.

- Cross-reference labels
- Part cross-references
- Contact cross-references

Cross-reference labels

A plain label can be used to make the name of a net visible in a schematic. If a label has the *xref* property activated, its behavior is changed so that it becomes a *cross-reference label*.

Cross-reference labels are typically placed at the right or left border of a schematic sheet, and indicate the next (or previous) sheet a particular net is used on. See the LABEL command for a detailed description of how this works.

Part cross-references

Electrical schematics often use electro-mechanical relays, consisting of a coil and one or more contact symbols. If the coil and contacts are distributed over various schematic sheets, it is useful to have each contact indicate which sheet its coil is on. This can be achieved by giving the coil gate in the device set an add level of *Must* (see the ADD command) and placing the text variable `'>XREF'` somewhere in the contacts' symbols (see the TEXT command).

When actually displayed, the `'>XREF'` text variable will be replaced with the sheet number, frame column and row (according to the part cross-reference format) of the *Must* gate of this device.

See Contact cross-references on how to display the contact locations on the coil's sheet.

Contact cross-references

On a multi-sheet electrical schematic with electro-mechanical relays that have their coils and contacts distributed over various sheets, it is useful to be able to see which sheets the individual contacts of a relay are on. EAGLE can automatically display this *contact cross-reference* for each relay coil if the following conditions are met.

The contact symbols need to contain the `'>XREF'` text variable in order to generate part cross-references.

The gate symbols shall be drawn in a way that the pins extend up and down, and that the origin is at the center of the symbol.

The first contact gate in the device set drawing shall be placed at an x-coordinate of 0, and its y-coordinate shall be high enough to make sure its lower pin is in the positive area, typically at 100mil. The rest of the contact gates shall be placed to the right of the first one, with their origins at the same y-coordinate as the first one. The coil gate can be placed at an arbitrary location.

In the schematic drawing the contact cross-reference will be shown at the same x-coordinate as the coil instance, and right below the y-coordinate defined by the text variable `'>CONTACT_XREF'`. This text variable can be defined either in a drawing frame symbol or directly on the sheet. If it is present in both, the one in the sheet is taken. The actual text will not be visible in the schematic sheet.

The graphical representation of the contact cross-reference consists of all the gates that have an `'>XREF'` text variable (except for the first *Must* gate, which is the coil and typically doesn't have this variable). The gates are rotated by 90 degrees and are shown from top to bottom at the same offsets as they have been drawn from left to right in the device set. Their sheet numbers and frame locations are displayed to the right of each gate that is actually used. Any other texts that have been defined in the symbol drawings will not be displayed when using these symbols for generating the contact cross-reference.

Note that the contact cross-reference can't be selected with the mouse. If you want to move it, move the coil instance and the contact cross-reference will automatically follow it. The contact cross-reference may get out of sync in case contact gates are invoked, moved, deleted or swapped, or if the `'>CONTACT_XREF'` text variable is modified. This will automatically be updated at the next window refresh.

User Language

The EAGLE User Language can be used to access the EAGLE data structures and to create a wide variety of output files.

To use this feature you have to write a User Language Program (ULP), and then execute it.

The following sections describe the EAGLE User Language in detail:

Syntax	lists the rules a ULP file has to follow
Data Types	defines the basic data types
Object Types	defines the EAGLE objects
Definitions	shows how to write a definition
Operators	lists the valid operators
Expressions	shows how to write expressions

Statements	defines the valid statements
Builtins	lists the builtin constants, functions etc.
Dialogs	shows how to implement a graphical frontent to a ULP

Writing a ULP

A User Language Program is a plain text file which is written in a C-like syntax. User Language Programs use the extension `.ulp`. You can create a ULP file with any text editor (provided it does not insert any additional control characters into the file) or you can use the builtin text editor.

A User Language Program consists of two major items, definitions and statements.

Definitions are used to define constants, variables and functions to be used by statements.

A simple ULP could look like this:

```
#usage "Add the characters in the word 'Hello'\n"
       "Usage: RUN sample.ulp"
// Definitions:
string hello = "Hello";
int count(string s)
{
  int c = 0;
  for (int i = 0; s[i]; ++i)
      c += s[i];
  return c;
}
// Statements:
output("sample") {
  printf("Count is: %d\n", count(hello));
  }
```

If the `#usage` directive is present, its value will be used in the Control Panel to display a description of the program.

If the result of the ULP shall be a specific command that shall be executed in the editor window, the `exit()` function can be used to send that command to the editor window.

Executing a ULP

User Language Programs are executed by the RUN command from an editor window's command line.

A ULP can return information on whether it has run successfully or not. You can use the `exit()` function to terminate the program and set the return value.

A return value of `0` means the ULP has ended "normally" (i.e. successfully), while any other value is considered as an abnormal program termination.

The default return value of any ULP is `0`.

When the RUN command is executed as part of a script file, the script is terminated if the ULP has exited with a return value other than `0`.

A special variant of the `exit()` function can be used to send a command to the editor window as a result of the ULP.

Syntax

The basic building blocks of a User Language Program are

- Whitespace
- Comments
- Directives
- Keywords
- Identifiers
- Constants
- Punctuators

All of these have to follow certain syntactical rules, which are described in their respective sections.

Whitespace

Before a User Language Program can be executed, it has to be read in from a file. During this read in process, the file contents is *parsed* into tokens and *whitespace*.

Any spaces (blanks), tabs, newline characters and comments are considered *whitespace* and are discarded.

The only place where ASCII characters representing *whitespace* are not discarded is within literal strings, like in

```
string s = "Hello World";
```

where the blank character between `'o'` and `'W'` remains part of the string.

If the final newline character of a line is preceded by a backslash (\), the backslash and newline character are both discarded, and the two lines are treated as one line:

```
"Hello \
World"
```

is parsed as `"Hello World"`

Comments

When writing a User Language Program it is good practice to add some descriptive text, giving the reader an idea about what this particular ULP does. You might also want to add your name (and, if available, your email address) to the ULP file, so that other people who use your program could contact you in case they have a problem or would like to suggest an improvement.

There are two ways to define a comment. The first one uses the syntax

```
/* some comment text */
```

which marks any characters between (and including) the opening `/*` and the closing `*/` as comment. Such comments may expand over more than one lines, as in

```
/* This is a
   multi line comment
```

```
*/
```

but they do not nest. The first `*/` that follows any `/*` will end the comment.

The second way to define a comment uses the syntax

```
int i; // some comment text
```

which marks any characters after (and including) the `//` and up to (but not including) the newline character at the end of the line as comment.

Directives

The following *directives* are available:

#include
#require
#usage

#include

A User Language Program can reuse code in other ULP files through the `#include` directive. The syntax is

```
#include "filename"
```

The file `filename` is first looked for in the same directory as the current source file (that is the file that contains the `#include` directive). If it is not found there, it is searched for in the directories contained in the ULP directory path.

The maximum include depth is 10.

Each `#include` directive is processed only **once**. This makes sure that there are no multiple definitions of the same variables or functions, which would cause errors.

Portability note

If *filename* contains a directory path, it is best to always use the **forward slash** as directory separator (even under Windows!). Windows drive letters should be avoided. This way a User Language Program will run on all platforms.

#require

Over time it may happen that newer versions of EAGLE implement new or modified User Language features, which can cause error messages when such a ULP is run from an older version of EAGLE. In order to give the user a dedicated message that this ULP requires at least a certain version of EAGLE, a ULP can contain the `#require` directive. The syntax is

```
#require version
```

The *version* must be given as a real constant of the form

```
V.RRrr
```

where `V` is the version number, `RR` is the release number and `rr` is the (optional) revision number (both padded with leading zeros if they are less than 10). For example, if a ULP requires at least EAGLE version 4.11r06 (which is the beta version that first implemented the `#require` directive), it could use

```
#require 4.1106
```

The proper directive for version 5.1.2 would be

```
#require 5.0102
```

#usage

Every User Language Program should contain information about its function, how to use it and maybe who wrote it.
The directive

```
#usage text [, text...]
```

implements a standard way to make this information available.

If the `#usage` directive is present, its `text` (which has to be a string constant) will be used in the Control Panel to display a description of the program.

In case the ULP needs to use this information in, for example, a dlgMessageBox(), the `text` is available to the program through the builtin constant `usage`.

Only the `#usage` directive of the main program file (that is the one started with the RUN command) will take effect. Therefore pure include files can (and should!) also have `#usage` directives of their own.

It is best to have the `#usage` directive at the beginning of the file, so that the Control Panel doesn't have to parse all the rest of the text when looking for the information to display.

If the usage information shall be made available in several langauges, the texts of the individual languages have to be separated by commas. Each of these texts has to start with the two letter code of the respective language (as delivered by the language() function), followed by a colon and any number of blanks. If no suitable text is found for the language used on the actual system, the first given text will be used (this one should generally be English in order to make the program accessible to the largest number of users).

Example

```
#usage "en: A sample ULP\n"
           "Implements an example that shows how to use the EAGLE User
Language\n"
           "Usage: RUN sample.ulp\n"
           "Author: john@home.org",
       "de: Beispiel eines ULPs\n"
           "Implementiert ein Beispiel das zeigt, wie man die EAGLE User
Language benutzt\n"
           "Aufruf: RUN sample.ulp\n"
           "Author: john@home.org"
```

Keywords

The following *keywords* are reserved for special purposes and must not be used as normal identifier names:

```
break
case
char
continue
default
do
else
enum
for
if
int
numeric
real
return
string
switch
void
while
```

In addition, the names of builtins and object types are also reserved and must not be used as identifier names.

Identifiers

An *identifier* is a name that is used to introduce a user defined constant, variable or function.

Identifiers consist of a sequence of letters (`a b c`..., `A B C`...), digits (`1 2 3`...) and underscores (_). The first character of an identifier **must** be a letter or an underscore.

Identifiers are case-sensitive, which means that

```
int Number, number;
```

would define two **different** integer variables.

The maximum length of an identifier is 100 characters, and all of these are significant.

Constants

Constants are literal data items written into a User Language Program. According to the different data types, there are also different types of constants.

- Character constants
- Integer constants
- Real constants
- String constants

Character Constants

A *character constant* consists of a single character or an escape sequence enclosed in single quotes, like

```
'a'
'='
'\n'
```

The type of a character constant is `char`.

Integer Constants

Depending on the first (and possibly the second) character, an *integer constant* is assumed to be expressed in different base values:

first	second	constant interpreted as
`0`	`1-7`	octal (base 8)
`0`	`x,X`	hexadecimal (base 16)
`1-9`		decimal (base 10)

The type of an integer constant is `int`.

Examples

`16`	decimal
`020`	octal
`0x10`	hexadecimal

Real Constants

A *real constant* follows the general pattern

```
[-]int.frac[e|E[±]exp]
```

which stands for

- optional sign
- decimal integer
- decimal point
- decimal fraction
- `e` or `E` and a signed integer exponent

You can omit either the decimal integer or the decimal fraction (but not both). You can omit either the decimal point or the letter `e` or `E` and the signed integer exponent (but not both).

The type of an real constant is `real`.

Examples

Constant	Value
`23.45e6`	23.45 x 10^6
`.0`	0.0
`0.`	0.0
`1.`	1.0
`-1.23`	-1.23
`2e-5`	2.0 x 10^-5
`3E+10`	3.0 x 10^10

`.09E34`	0.09 x 10^34

String Constants

A *string constant* consists of a sequence of characters or escape sequences enclosed in double quotes, like

```
"Hello world\n"
```

The type of a string constant is `string`.

String constants can be of any length (provided there is enough free memory available).

String constants can be concatenated by simply writing them next to each other to form larger strings:

```
string s = "Hello" " world\n";
```

It is also possible to extend a string constant over more than one line by escaping the newline character with a backslash (\):

```
string s = "Hello \
world\n";
```

Escape Sequences

An *escape sequence* consists of a backslash (\), followed by one or more special characters:

Sequence	Value
`\a`	audible bell
`\b`	backspace
`\f`	form feed
`\n`	new line
`\r`	carriage return
`\t`	horizontal tab
`\v`	vertical tab
`\\`	backslash
`\'`	single quote
`\"`	double quote
`\O`	`O` = up to 3 octal digits
`\xH`	`H` = up to 2 hex digits

Any character following the initial backslash that is not mentioned in this list will be treated as that character (without the backslash).

Escape sequences can be used in character constants and string constants.

Examples

```
'\n'
"A tab\tinside a text\n"
"Ring the bell\a\n"
```

Punctuators

The *punctuators* used in a User Language Program are

`[]`	Brackets
`()`	Parentheses
`{}`	Braces
`,`	Comma
`;`	Semicolon
`:`	Colon
`=`	Equal sign

Other special characters are used as operators in a ULP.

Brackets

Brackets are used in array definitions

```
int ai[];
```

in array subscripts

```
n = ai[2];
```

and in string subscripts to access the individual characters of a string

```
string s = "Hello world";
char c = s[2];
```

Parentheses

Parentheses group expressions (possibly altering normal operator precedence), isolate conditional expressions, and indicate function calls and function parameters:

```
d = c * (a + b);
if (d == z) ++x;
func();
void func2(int n) { ... }
```

Braces

Braces indicate the start and end of a compound statement:

```
if (d == z) {
   ++x;
   func();
   }
```

and are also used to group the values of an array initializer:

```
int ai[] = { 1, 2, 3 };
```

Comma

The *comma* separates the elements of a function argument list or the parameters of a function call:

```
int func(int n, real r, string s) { ... }
int i = func(1, 3.14, "abc");
```

It also delimits the values of an array initializer:

```
int ai[] = { 1, 2, 3 };
```

and it separates the elements of a variable definition:

```
int i, j, k;
```

Semicolon

The *semicolon* terminates a statement, as in

```
i = a + b;
```

and it also delimits the init, test and increment expressions of a for statement:

```
for (int n = 0; n < 3; ++n) {
    func(n);
    }
```

Colon

The *colon* indicates the end of a label in a switch statement:

```
switch (c) {
  case 'a': printf("It was an 'a'\n"); break;
  case 'b': printf("It was a  'b'\n"); break;
  default:  printf("none of them\n");
  }
```

Equal Sign

The *equal sign* separates variable definitions from initialization lists:

```
int i = 10;
char c[] = { 'a', 'b', 'c' };
```

It is also used as an assignment operator.

Data Types

A User Language Program can define variables of different types, representing the different kinds of information available in the EAGLE data structures.

The four basic data types are

`char`	for single characters
`int`	for integral values
`real`	for floating point values

`string`	for textual information

Besides these basic data types there are also high level Object Types, which represent the data structures stored in the EAGLE data files.

The special data type `void` is used only as a return type of a function, indicating that this function does **not** return any value.

char

The data type `char` is used to store single characters, like the letters of the alphabet, or small unsigned numbers.

A variable of type `char` has a size of 8 bit (one byte), and can store any value in the range `0..255`.

See also Operators, Character Constants

int

The data type `int` is used to store signed integral values, like the coordinates of an object.

A variable of type `int` has a size of 32 bit (four byte), and can store any value in the range `-2147483648..2147483647`.

See also Integer Constants

real

The data type `real` is used to store signed floating point values, like the grid distance.

A variable of type `real` has a size of 64 bit (eight byte), and can store any value in the range `±2.2e-308..±1.7e+308` with a precision of 15 digits.

See also Real Constants

string

The data type `string` is used to store textual information, like the name of a part or net.

A variable of type `string` is not limited in it's size (provided there is enough memory available).

Variables of type `string` are defined without an explicit *size*. They grow automatically as necessary during program execution.

The elements of a `string` variable are of type `int` and can be accessed individually by using `[index]`. The first character of a `string` has the index `0`:

```
string s = "Layout";
printf("Third char is: %c\n", s[2]);
```

This would print the character `'y'`. Note that `s[2]` returns the **third** character of `s`!

A lossless conversion to `char` is possible for standard ASCII strings:

```
string s = "Layout";
```

```
char c = s[2];
```

See also Operators, Builtin Functions, String Constants

Implementation details

The data type `string` is actually implemented like native C-type zero terminated strings. Looking at the following variable definition

```
string s = "abcde";
```

`s[4]` is the character `'e'`, and `s[5]` is the character `'\0'`, or the integer value `0x00`. This fact may be used to determine the end of a string without using the `strlen()` function, as in

```
for (int i = 0; s[i]; ++i) {
    // do something with s[i]
    }
```

It is also perfectly ok to "cut off" part of a string by "punching" a zero character into it:

```
string s = "abcde";
s[3] = 0;
```

This will result in `s` having the value `"abc"`. Note that everything following the zero character will actually be gone, and it won't come back by restoring the original character. The same applies to any other operation that sets a character to 0, for instance --s[3].

Type Conversions

The result type of an arithmetic expression, such as `a + b`, where `a` and `b` are different arithmetic types, is equal to the "larger" of the two operand types.

Arithmetic types are `char`, `int` and `real` (in that order). So if, e.g. `a` is of type `int` and `b` is of type `real`, the result of the expression `a + b` would be `real`.

See also Typecast

Typecast

The result type of an arithmetic expression can be explicitly converted to a different arithmetic type by applying a *typecast* to it.

The general syntax of a typecast is

```
type(expression)
```

where `type` is one of `char`, `int` or `real`, and `expression` is any arithmetic expression.

When typecasting a `real` expression to `int`, the fractional part of the value is truncated!

See also Type Conversions

Object Types

The EAGLE data structures are stored in three binary file types:

- Library (*.lbr)
- Schematic (*.sch)
- Board (*.brd)

These data files contain a hierarchy of objects. In a User Language Program you can access these hierarchies through their respective builtin access statements:

```
library(L) { ... }
schematic(S) { ... }
board(B) { ... }
```

These access statements set up a context within which you can access all of the objects contained in the library, schematic or board.

The properties of these objects can be accessed through *members*.

There are two kinds of members:

- Data members
- Loop members

Data members immediately return the requested data from an object. For example, in

```
board(B) {
  printf("%s\n", B.name);
  }
```

the data member *name* of the board object *B* returns the board's name.
Data members can also return other objects, as in

```
board(B) {
  printf("%f\n", B.grid.size);
  }
```

where the board's *grid* data member returns a grid object, of which the *size* data member then returns the grid's size.

Loop members are used to access multiple objects of the same kind, which are contained in a higher level object:

```
board(B) {
  B.elements(E) {
    printf("%-8s %-8s\n", E.name, E.value);
    }
  }
```

This example uses the board's *elements()* loop member function to set up a loop through all of the board's elements. The block following the `B.elements(E)` statement is executed in turn for each element, and the current element can be referenced inside the block through the name `E`.

Loop members process objects in alpha-numerical order, provided they have a name.

A loop member function creates a variable of the type necessary to hold the requested objects. You are free to use any valid name for such a variable, so the above example might also be written as

```
board(MyBoard) {
  MyBoard.elements(TheCurrentElement) {
    printf("%-8s %-8s\n", TheCurrentElement.name, TheCurrentElement.value);
    }
  }
```

and would do the exact same thing. The scope of the variable created by a loop member function is limited to the statement (or block) immediately following the loop function call.

Object hierarchy of a Library:

```
LIBRARY
  GRID
  LAYER
  DEVICESET
    DEVICE
    GATE
  PACKAGE
    CONTACT
      PAD
      SMD
    CIRCLE
    HOLE
    RECTANGLE
    FRAME
    DIMENSION
    TEXT
    WIRE
    POLYGON
      WIRE
  SYMBOL
    PIN
    CIRCLE
    RECTANGLE
    FRAME
    DIMENSION
    TEXT
    WIRE
    POLYGON
      WIRE
```

Object hierarchy of a Schematic:

```
SCHEMATIC
  GRID
  LAYER
  LIBRARY
  ATTRIBUTE
  VARIANTDEF
  PART
    ATTRIBUTE
    VARIANT
  SHEET
    CIRCLE
    RECTANGLE
    FRAME
    DIMENSION
    TEXT
    WIRE
    POLYGON
      WIRE
    INSTANCE
      ATTRIBUTE
    BUS
      SEGMENT
        LABEL
          TEXT
          WIRE
        WIRE
    NET
      SEGMENT
        JUNCTION
        PINREF
        TEXT
        WIRE
```

Change note from version 5 to version 6, compatibility

- Since version 6 the instance is in the hierarchy no longer below the part but below the sheet.
- The part is no longer below the sheet, but below the schematic.

For compatibility reasons the access by the according member functions is further supported, but the behaviour of the Object Functions reflects the new hierarchy.

Object hierarchy of a Board:

```
BOARD
  GRID
  LAYER
  LIBRARY
  ATTRIBUTE
  VARIANTDEF
  CIRCLE
  HOLE
  RECTANGLE
  FRAME
  DIMENSION
  TEXT
  WIRE
  POLYGON
    WIRE
  ELEMENT
    ATTRIBUTE
    VARIANT
  SIGNAL
    CONTACTREF
    POLYGON
      WIRE
    VIA
    WIRE
```

UL_ARC

Data members

`angle1`	real (start angle, `0.0`...`359.9`)
`angle2`	real (end angle, `0.0`...`719.9`)
`cap`	int (`CAP_...`)
`layer`	int
`radius`	int
`width`	int
`x1, y1`	int (starting point)
`x2, y2`	int (end point)
`xc, yc`	int (center point)

See also UL_WIRE

Constants

`CAP_FLAT`	flat arc ends
`CAP_ROUND`	round arc ends

Note

Start and end angles are defined mathematically positive (i.e. counterclockwise), with `angle1` < `angle2`. In order to assure this condition, the start and end point of an UL_ARC may be exchanged with respect to the UL_WIRE the arc has been derived from.

Example

```
board(B) {
  B.wires(W) {
    if (W.arc)
       printf("Arc: (%d %d), (%d %d), (%d %d)\n",
              W.arc.x1, W.arc.y1, W.arc.x2, W.arc.y2, W.arc.xc, W.arc.yc);
    }
  }
```

UL_AREA

Data members

`x1, y1`	int (lower left corner)
`x2, y2`	int (upper right corner)

See also UL_BOARD, UL_DEVICE, UL_PACKAGE, UL_SHEET, UL_SYMBOL

A UL_AREA is an abstract object which gives information about the area covered by an object. For a UL_PACKAGE or UL_SYMBOL in a UL_ELEMENT or UL_INSTANCE context, respectively, the area is given in absolute drawing coordinates, including the offset of the element or instance.

Example

```
board(B) {
  printf("Area: (%d %d), (%d %d)\n",
          B.area.x1, B.area.y1, B.area.x2, B.area.y2);
  }
```

UL_ATTRIBUTE

Data members

`constant`	int (0=variable, i.e. allows overwriting, 1=constant - see note)
`defaultvalue`	string (see note)
`display`	int (`ATTRIBUTE_DISPLAY_FLAG_...`)
`name`	string
`text`	UL_TEXT (see note)
`value`	string

See also UL_DEVICE, UL_PART, UL_INSTANCE, UL_ELEMENT

Constants

`ATTRIBUTE_DISPLAY_FLAG_OFF`	nothing is displayed
`ATTRIBUTE_DISPLAY_FLAG_VALUE`	value is displayed
`ATTRIBUTE_DISPLAY_FLAG_NAME`	name is displayed

A UL_ATTRIBUTE can be used to access the *attributes* that have been defined in the library for a device, or assigned to a part in the schematic or board.

Note

`display` contains a bitwise or'ed value consisting of `ATTRIBUTE_DISPLAY_FLAG_...` and defines which parts of the attribute are actually drawn. This value is only valid if `display` is used in a UL_INSTANCE or UL_ELEMENT context.

In a UL_ELEMENT context `constant` only returns an actual value if f/b annotation is active, otherwise it returns 0.

The `defaultvalue` member returns the value as defined in the library (if different from the actual value, otherwise the same as `value`). In a UL_ELEMENT context `defaultvalue` only returns an actual value if f/b annotation is active, otherwise an empty string is returned.

The `text` member is only available in a UL_INSTANCE or UL_ELEMENT context and returns a UL_TEXT object that contains all the text parameters. The value of this text object is the string as it will be displayed according to the UL_ATTRIBUTE's 'display' parameter. If called from a different context, the data of the returned UL_TEXT object is undefined.

For global attributes only `name` and `value` are defined.

Example

```
schematic(SCH) {
  SCH.parts(P) {
    P.attributes(A) {
      printf("%s = %s\n", A.name, A.value);
      }
    }
  }
schematic(SCH) {
  SCH.attributes(A) { // global attributes
    printf("%s = %s\n", A.name, A.value);
    }
  }
```

UL_BOARD

Data members

`alwaysvectorfont`	int (`ALWAYS_VECTOR_FONT_..., see note`)
`area`	UL_AREA
`description`	string
`grid`	UL_GRID
`headline`	string
`name`	string (see note)
`verticaltext`	int (`VERTICAL_TEXT_...`)

Loop members

`attributes()`	UL_ATTRIBUTE (see note)
`circles()`	UL_CIRCLE
`classes()`	UL_CLASS
`dimensions()`	UL_DIMENSION
`elements()`	UL_ELEMENT
`frames()`	UL_FRAME
`holes()`	UL_HOLE
`layers()`	UL_LAYER
`libraries()`	UL_LIBRARY
`polygons()`	UL_POLYGON
`rectangles()`	UL_RECTANGLE
`signals()`	UL_SIGNAL
`texts()`	UL_TEXT
`variantdefs()`	UL_VARIANTDEF
`wires()`	UL_WIRE

See also UL_LIBRARY, UL_SCHEMATIC

Constants

`ALWAYS_VECTOR_FONT_GUI`	alwaysvectorfont is set in the user interface dialog
`ALWAYS_VECTOR_FONT_PERSISTENT`	alwaysvectorfont is set persistent in this board

`VERTICAL_TEXT_UP`	reading direction for vertical texts: up
`VERTICAL_TEXT_DOWN`	reading direction for vertical texts: down

Note

The value returned by `alwaysvectorfont` can be used in boolean context or can be masked with the `ALWAYS_VECTOR_FONT_...` constants to determine the source of this setting, as in

```
if (B.alwaysvectorfont) {
   // alwaysvectorfont is set in general
   }
if (B.alwaysvectorfont & ALWAYS_VECTOR_FONT_GUI) {
   // alwaysvectorfont is set in the user interface
   }
```

The `name` member returns the full file name, including the directory.

The `attributes()` loop member loops through the *global* attributes.

Example

```
board(B) {
  B.elements(E) printf("Element: %s\n", E.name);
  B.signals(S)  printf("Signal: %s\n", S.name);
  }
```

UL_BUS

Data members

`name`	string (BUS_NAME_LENGTH)

Loop members

`segments()`	UL_SEGMENT

See also UL_SHEET

Constants

`BUS_NAME_LENGTH`	max. length of a bus name (obsolete - as from version 4 bus names can have any length)

Example

```
schematic(SCH) {
  SCH.sheets(SH) {
    SH.busses(B) printf("Bus: %s\n", B.name);
    }
  }
```

UL_CIRCLE

Data members

`layer`	int
`radius`	int
`width`	int
`x, y`	int (center point)

See also UL_BOARD, UL_PACKAGE, UL_SHEET, UL_SYMBOL

Example

```
board(B) {
  B.circles(C) {
    printf("Circle: (%d %d), r=%d, w=%d\n",
           C.x, C.y, C.radius, C.width);
    }
  }
```

UL_CLASS

Data members

`clearance[number]`	int (see note)
`drill`	int
`name`	string (see note)
`number`	int
`width`	int

See also Design Rules, UL_NET, UL_SIGNAL, UL_SCHEMATIC, UL_BOARD

Note

The `clearance` member returns the clearance value between this net class and the net class with the given number. If the number (and the square brackets) is ommitted, the net class's own clearance value is returned. If a number is given, it must be between 0 and the number of this net class.

If the `name` member returns an empty string, the net class is not defined and therefore not in use by any signal or net.

Example

```
board(B) {
  B.signals(S) {
    printf("%-10s %d %s\n", S.name, S.class.number, S.class.name);
    }
  }
```

UL_CONTACT

Data members

`name`	string (`CONTACT_NAME_LENGTH`)
`pad`	UL_PAD
`signal`	string
`smd`	UL_SMD
`x, y`	int (center point, see note)

Loop members

`polygons()`	UL_POLYGON (of arbitrary pad shapes)
`wires()`	UL_WIRE (of arbitrary pad shapes)

See also UL_PACKAGE, UL_PAD, UL_SMD, UL_CONTACTREF, UL_PINREF

Constants

`CONTACT_NAME_LENGT`	max. recommended length of a contact name (used in formatted

H	output only)

Note

The `signal` data member returns the signal this contact is connected to (only available in a board context).

The coordinates (`x, y`) of the contact depend on the context in which it is called:

- if the contact is derived from a UL_LIBRARY context, the coordinates of the contact will be the same as defined in the package drawing
- in all other cases, they will have the actual values from the board

Example

```
library(L) {
  L.packages(PAC) {
    PAC.contacts(C) {
      printf("Contact: '%s', (%d %d)\n",
             C.name, C.x, C.y);
      }
    }
  }
```

UL_CONTACTREF

Data members

`contact`	UL_CONTACT
`element`	UL_ELEMENT
`route`	int (`CONTACT_ROUTE_...`)
`routetag`	string (see note)

See also UL_SIGNAL, UL_PINREF

Constants

`CONTACT_ROUTE_ALL`	must explicitly route to all contacts
`CONTACT_ROUTE_ANY`	may route to any contact

Note

If `route` has the value `CONTACT_ROUTE_ANY`, the `routetag` data member returns an additional tag which describes a group of `contactref`s belonging to the same pin.

Example

```
board(B) {
  B.signals(S) {
    printf("Signal '%s'\n", S.name);
    S.contactrefs(C) {
      printf("\t%s, %s\n", C.element.name, C.contact.name);
      }
    }
  }
```

UL_DEVICE

Data members

`activetechnology`	string (see note)
`area`	UL_AREA
`description`	string
`headline`	string
`library`	string
`name`	string (`DEVICE_NAME_LENGTH`)
`package`	UL_PACKAGE (see note)
`prefix`	string (`DEVICE_PREFIX_LENGTH`)
`technologies`	string (see note)
`value`	string ("On" or "Off")

Loop members

`attributes()`	UL_ATTRIBUTE (see note)
`gates()`	UL_GATE

See also UL_DEVICESET, UL_LIBRARY, UL_PART

Constants

`DEVICE_NAME_LENGTH`	max. recommended length of a device name (used in formatted output only)
`DEVICE_PREFIX_LENGTH`	max. recommended length of a device prefix (used in formatted output only)

All members of UL_DEVICE, except for `name` and `technologies`, return the same values as the respective members of the UL_DEVICESET in which the UL_DEVICE has been defined. The `name` member returns the name of the package variant this device has been created for using the PACKAGE command. When using the `description` text keep in mind that it may contain newline characters (`'\n'`).

Note

The value returned by the `activetechnology` member depends on the context in which it is called:

- if the device is derived from the deviceset that is currently edited in the library editor window, the active technology, set by the TECHNOLOGY command, will be returned
- if the device is derived from a UL_PART, the actual technology used by the part will be returned
- otherwise an empty string will be returned.

The `package` data member returns the package that has been assigned to the device through a PACKAGE command. It can be used as a boolean function to check whether a package has been assigned to a device (see example below).

The value returned by the `technologies` member depends on the context in which it is called:

- if the device is derived from a UL_DEVICESET, `technologies` will return a string containing all of the device's technologies, separated by blanks

- if the device is derived from a UL_PART, only the actual technology used by the part will be returned.

The `attributes()` loop member takes an additional parameter that specifies for which technology the attributes shall be delivered (see the second example below).

Examples

```
library(L) {
  L.devicesets(S) {
    S.devices(D) {
      if (D.package)
         printf("Device: %s, Package: %s\n", D.name, D.package.name);
      D.gates(G) {
        printf("\t%s\n", G.name);
        }
      }
    }
  }

library(L) {
  L.devicesets(DS) {
    DS.devices(D) {
      string t[];
      int n = strsplit(t, D.technologies, ' ');
      for (int i = 0; i < n; i++) {
          D.attributes(A, t[i]) {
            printf("%s = %s\n", A.name, A.value);
            }
          }
      }
    }
  }
```

UL_DEVICESET

Data members

`activedevice`	UL_DEVICE (see note)
`area`	UL_AREA
`description`	string
`headline`	string (see note)
`library`	string
`name`	string (`DEVICE_NAME_LENGTH`)
`prefix`	string (`DEVICE_PREFIX_LENGTH`)
`value`	string ("On" or "Off")

Loop members

`devices()`	UL_DEVICE
`gates()`	UL_GATE

See also UL_DEVICE, UL_LIBRARY, UL_PART

Constants

`DEVICE_NAME_LENGTH`	max. recommended length of a device name (used in formatted output only)
`DEVICE_PREFIX_LENGT`	max. recommended length of a device prefix (used in formatted

H	output only)

Note

If a deviceset is currently edited in a library editor window, the `activedevice` member returns the active device, selected by a PACKAGE command. It can be used as a boolean function to check the availability of such an `activedevice` (see example below).

The `description` member returns the complete descriptive text as defined with the DESCRIPTION command, while the `headline` member returns only the first line of the description, without any HTML tags. When using the `description` text keep in mind that it may contain newline characters (`'\n'`).

Example

```
library(L) {
  L.devicesets(D) {
    printf("Device set: %s, Description: %s\n", D.name, D.description);
    D.gates(G) {
      printf("\t%s\n", G.name);
      }
    }
  }

if (deviceset)
   deviceset(DS) {
     if (DS.activedevice)
        printf("Active Device: %s\n", DS.activedevice.name);
     }
```

UL_DIMENSION

Data members

`dtype`	int (`DIMENSION_...`)
`layer`	int
`extlength`	int
`extoffset`	int
`extwidth`	int
`precision`	int
`ratio`	int
`size`	int
`unit`	int (`GRID_UNIT_...`)
`visible`	int (`unit`, 0=off, 1=on)
`width`	int
`x1, y1`	int (first reference point)
`x2, y2`	int (second reference point)
`x3, y3`	int (alignment reference point)

Loop members

`texts()`	UL_TEXT
`wires()`	UL_WIRE

See also UL_BOARD, UL_GRID, UL_PACKAGE, UL_SHEET, UL_SYMBOL

Constants

DIMENSION_PARALLEL	linear dimension with parallel measurement line
DIMENSION_HORIZONTAL	linear dimension with horizontal measurement line
DIMENSION_VERTICAL	linear dimension with vertical measurement line
DIMENSION_RADIUS	radial dimension
DIMENSION_DIAMETER	diameter dimension
DIMENSION_ANGLE	angle dimension
DIMENSION_LEADER	an arbitrary pointer

Note

The `texts()` and `wires()` loop members loop through all the texts and wires the dimension consists of.

Example

```
board(B) {
  B.dimensions(D) {
    printf("Dimension: (%d %d), (%d %d), (%d %d)\n",
           D.x1, D.y1, D.x2, D.y2, D.x3, D.y3);
    }
  }
```

UL_ELEMENT

Data members

angle	real (0.0...359.9)
attribute[]	string (see note)
column	string (see note)
locked	int
mirror	int
name	string (ELEMENT_NAME_LENGTH)
package	UL_PACKAGE
populate	int (0=do not populate, 1=populate)
row	string (see note)
smashed	int (see note)
spin	int
value	string (ELEMENT_VALUE_LENGTH)
x, y	int (origin point)

Loop members

attributes()	UL_ATTRIBUTE
texts()	UL_TEXT (see note)
variants()	UL_VARIANT

See also UL_BOARD, UL_CONTACTREF

Constants

ELEMENT_NAME_LENGTH	max. recommended length of an element name (used in formatted output only)

`ELEMENT_VALUE_LENGTH`	max. recommended length of an element value (used in formatted output only)

Note

The `attribute[]` member can be used to query a UL_ELEMENT for the value of a given attribute (see the second example below). The returned string is empty if there is no attribute by the given name, or if this attribute is explicitly empty.

The `texts()` member only loops through those texts of the element that have been detached using **SMASH**, and through the visible texts of any attributes assigned to this element. To process all texts of an element (e.g. when drawing it), you have to loop through the element's own `texts()` member as well as the `texts()` member of the element's package.

`angle` defines how many degrees the element is rotated counterclockwise around its origin.

The `column` and `row` members return the column and row location within the frame in the board drawing. If there is no frame in the drawing, or the element is placed outside the frame, a `'?'` (question mark) is returned.

The `smashed` member tells whether the element is smashed. This function can also be used to find out whether there is a detached text parameter by giving the name of that parameter in square brackets, as in `smashed["VALUE"]`. This is useful in case you want to select such a text with the MOVE command by doing `MOVE R5>VALUE`. Valid parameter names are "NAME" and "VALUE", as well as the names of any user defined attributes. They are treated case insensitive, and they may be preceded by a `'>'` character.

Examples

```
board(B) {
  B.elements(E) {
    printf("Element: %s, (%d %d), Package=%s\n",
           E.name, E.x, E.y, E.package.name);
    }
  }

board(B) {
  B.elements(E) {
    if (E.attribute["REMARK"])
       printf("%s: %s\n", E.name, E.attribute["REMARK"]);
    }
  }
```

UL_FRAME

Data members

`columns`	int (`-127`...`127`)
`rows`	int (`-26`...`26`)
`border`	int (`FRAME_BORDER_...`)
`layer`	int
`x1, y1`	int (lower left corner)
`x2, y2`	int (upper right corner)

Loop members

`texts()`	UL_TEXT
`wires()`	UL_WIRE

See also UL_BOARD, UL_PACKAGE, UL_SHEET, UL_SYMBOL

Constants

`FRAME_BORDER_BOTTOM`	bottom border is drawn
`FRAME_BORDER_RIGHT`	right border is drawn
`FRAME_BORDER_TOP`	top border is drawn
`FRAME_BORDER_LEFT`	left border is drawn

Note

`border` contains a bitwise or'ed value consisting of `FRAME_BORDER_...` and defines which of the four borders are actually drawn.

The `texts()` and `wires()` loop members loop through all the texts and wires the frame consists of.

Example

```
board(B) {
  B.frames(F) {
    printf("Frame: (%d %d), (%d %d)\n",
           F.x1, F.y1, F.x2, F.y2);
    }
  }
```

UL_GATE

Data members

`addlevel`	int (`GATE_ADDLEVEL_...`)
`name`	string (`GATE_NAME_LENGTH`)
`swaplevel`	int
`symbol`	UL_SYMBOL
`x, y`	int (origin point, see note)

See also UL_DEVICE

Constants

`GATE_ADDLEVEL_MUST`	must
`GATE_ADDLEVEL_CAN`	can
`GATE_ADDLEVEL_NEXT`	next
`GATE_ADDLEVEL_REQUEST`	request
`GATE_ADDLEVEL_ALWAYS`	always
`GATE_NAME_LENGTH`	max. recommended length of a gate name (used in formatted output only)

Note

The coordinates of the origin point (x, y) are always those of the gate's position within the device, even if the UL_GATE has been derived from a UL_INSTANCE.

Example

```
library(L) {
  L.devices(D) {
    printf("Device: %s, Package: %s\n", D.name, D.package.name);
    D.gates(G) {
      printf("\t%s, swaplevel=%d, symbol=%s\n",
             G.name, G.swaplevel, G.symbol.name);
      }
    }
  }
```

UL_GRID

Data members

`distance`	real
`dots`	int (0=lines, 1=dots)
`multiple`	int
`on`	int (0=off, 1=on)
`unit`	int (`GRID_UNIT_...`)
`unitdist`	int (`GRID_UNIT_...`)

See also UL_BOARD, UL_LIBRARY, UL_SCHEMATIC, Unit Conversions

Constants

`GRID_UNIT_MIC`	microns
`GRID_UNIT_MM`	millimeter
`GRID_UNIT_MIL`	mil
`GRID_UNIT_INCH`	inch

Note

`unitdist` returns the grid unit that was set to define the actual grid size (returned by `distance`), while `unit` returns the grid unit that is used to display values or interpret user input.

Example

```
board(B) {
  printf("Gridsize=%f\n", B.grid.distance);
  }
```

UL_HOLE

Data members

`diameter[layer]`	int (see note)
`drill`	int

`drillsymbol`	int
`x, y`	int (center point)

See also UL_BOARD, UL_PACKAGE

Note

`diameter[]` is only defined vor layers `LAYER_TSTOP` and `LAYER_BSTOP` and returns the diameter of the solder stop mask in the given layer.

`drillsymbol` returns the number of the drill symbol that has been assigned to this drill diameter (see the manual for a list of defined drill symbols). A value of `0` means that no symbol has been assigned to this drill diameter.

Example

```
board(B) {
  B.holes(H) {
    printf("Hole: (%d %d), drill=%d\n",
           H.x, H.y, H.drill);
    }
  }
```

UL_INSTANCE

Data members

`angle`	real (`0`, `90`, `180` and `270`)
`column`	string (see note)
`gate`	UL_GATE
`mirror`	int
`name`	string (`INSTANCE_NAME_LENGTH`)
`part`	UL_PART
`row`	string (see note)
`sheet`	int (0=unused, >0=sheet number)
`smashed`	int (see note)
`value`	string (`PART_VALUE_LENGTH`)
`x, y`	int (origin point)

Loop members

`attributes()`	UL_ATTRIBUTE (see note)
`texts()`	UL_TEXT (see note)
`xrefs()`	UL_GATE (see note)

See also UL_PINREF

Constants

`INSTANCE_NAME_LENGTH`	max. recommended length of an instance name (used in formatted output only)
`PART_VALUE_LENGTH`	max. recommended length of a part value (instances do not have a value of their own!)

Note

The `attributes()` member only loops through those attributes that have been explicitly assigned to this instance (including *smashed* attributes).

The `texts()` member only loops through those texts of the instance that have been detached using **SMASH**, and through the visible texts of any attributes assigned to this instance. To process all texts of an instance, you have to loop through the instance's own `texts()` member as well as the `texts()` member of the instance's gate's symbol. If attributes have been assigned to an instance, `texts()` delivers their texts in the form as they are currently visible.

The `column` and `row` members return the column and row location within the frame on the sheet on which this instance is invoked. If there is no frame on that sheet, or the instance is placed outside the frame, a `'?'` (question mark) is returned. These members can only be used in a sheet context.

The `smashed` member tells whether the instance is smashed. This function can also be used to find out whether there is a detached text parameter by giving the name of that parameter in square brackets, as in `smashed["VALUE"]`. This is useful in case you want to select such a text with the MOVE command by doing `MOVE R5>VALUE`. Valid parameter names are "NAME", "VALUE", "PART" and "GATE", as well as the names of any user defined attributes. They are treated case insensitive, and they may be preceded by a `'>'` character.

The `xrefs()` member loops through the contact cross-reference gates of this instance. These are only of importance if the ULP is going to create a drawing of some sort (for instance a DXF file).

Example

```
schematic(S) {
  S.parts(P) {
    printf("Part: %s\n", P.name);
    P.instances(I) {
      if (I.sheet != 0)
         printf("\t%s used on sheet %d\n", I.name, I.sheet);
      }
    }
  }
```

UL_JUNCTION

Data members

`diameter`	int
`x, y`	int (center point)

See also UL_SEGMENT

Example

```
schematic(SCH) {
  SCH.sheets(SH) {
    SH.nets(N) {
      N.segments(SEG) {
        SEG.junctions(J) {
          printf("Junction: (%d %d)\n", J.x, J.y);
          }
        }
      }
    }
  }
```

UL_LABEL

Data members

`angle`	real (0.0...359.9)
`layer`	int
`mirror`	int
`spin`	int
`text`	UL_TEXT
`x, y`	int (origin point)
`xref`	int (0=plain, 1=cross-reference)

Loop members

`wires()`	UL_WIRE (see note)

See also UL_SEGMENT

Note

If `xref` returns a non-zero value, the `wires()` loop member loops through the wires that form the flag of a cross-reference label. Otherwise it is an empty loop.

The `angle`, `layer`, `mirror` and `spin` members always return the same values as those of the UL_TEXT object returned by the `text` member. The `x` and `y` members of the text return slightly offset values for cross-reference labels (non-zero `xref`), otherwise they also return the same values as the UL_LABEL.

`xref` is only meaningful for net labels. For bus labels it always returns 0.

Example

```
sheet(SH) {
  SH.nets(N) {
    N.segments(S) {
      S.labels(L) {
        printf("Label: %d %d '%s'\n", L.x, L.y, L.text.value);
        }
      }
    }
  }
```

UL_LAYER

Data members

`color`	int
`fill`	int
`name`	string (`LAYER_NAME_LENGTH`)
`number`	int
`used`	int (0=unused, 1=used)
`visible`	int (0=off, 1=on)

See also UL_BOARD, UL_LIBRARY, UL_SCHEMATIC

Constants

`LAYER_NAME_LENGTH`	max. recommended length of a layer name (used in formatted output only)
`LAYER_TOP`	layer numbers
`LAYER_BOTTOM`	
`LAYER_PADS`	
`LAYER_VIAS`	
`LAYER_UNROUTED`	
`LAYER_DIMENSION`	
`LAYER_TPLACE`	
`LAYER_BPLACE`	
`LAYER_TORIGINS`	
`LAYER_BORIGINS`	
`LAYER_TNAMES`	
`LAYER_BNAMES`	
`LAYER_TVALUES`	
`LAYER_BVALUES`	
`LAYER_TSTOP`	
`LAYER_BSTOP`	
`LAYER_TCREAM`	
`LAYER_BCREAM`	
`LAYER_TFINISH`	
`LAYER_BFINISH`	
`LAYER_TGLUE`	
`LAYER_BGLUE`	
`LAYER_TTEST`	
`LAYER_BTEST`	
`LAYER_TKEEPOUT`	
`LAYER_BKEEPOUT`	
`LAYER_TRESTRICT`	
`LAYER_BRESTRICT`	
`LAYER_VRESTRICT`	
`LAYER_DRILLS`	
`LAYER_HOLES`	
`LAYER_MILLING`	
`LAYER_MEASURES`	
`LAYER_DOCUMENT`	
`LAYER_REFERENCE`	
`LAYER_TDOCU`	
`LAYER_BDOCU`	
`LAYER_NETS`	
`LAYER_BUSSES`	

`LAYER_PINS`	
`LAYER_SYMBOLS`	
`LAYER_NAMES`	
`LAYER_VALUES`	
`LAYER_INFO`	
`LAYER_GUIDE`	
`LAYER_USER`	lowest number for user defined layers (100)

Example

```
board(B) {
  B.layers(L) printf("Layer %3d %s\n", L.number, L.name);
  }
```

UL_LIBRARY

Data members

`description`	string (see note)
`grid`	UL_GRID
`headline`	string
`name`	string (`LIBRARY_NAME_LENGTH`, see note)

Loop members

`devices()`	UL_DEVICE
`devicesets()`	UL_DEVICESET
`layers()`	UL_LAYER
`packages()`	UL_PACKAGE
`symbols()`	UL_SYMBOL

See also UL_BOARD, UL_SCHEMATIC

Constants

`LIBRARY_NAME_LENGTH`	max. recommended length of a library name (used in formatted output only)

The `devices()` member loops through all the package variants and technologies of all UL_DEVICESETs in the library, thus resulting in all the actual device variations available. The `devicesets()` member only loops through the UL_DEVICESETs, which in turn can be queried for their UL_DEVICE members.

Note

The `description` member returns the complete descriptive text as defined with the DESCRIPTION command, while the `headline` member returns only the first line of the description, without any HTML tags. When using the `description` text keep in mind that it may contain newline characters (`'\n'`). The `description` and `headline` information is only available within a library drawing, not if the library is derived form a UL_BOARD or UL_SCHEMATIC context.

If the library is derived form a UL_BOARD or UL_SCHEMATIC context, `name` returns the pure library name (without path or extension). Otherwise it returns the full library file name.

Example

```
library(L) {
  L.devices(D)     printf("Dev: %s\n", D.name);
  L.devicesets(D)  printf("Dev: %s\n", D.name);
  L.packages(P)    printf("Pac: %s\n", P.name);
  L.symbols(S)     printf("Sym: %s\n", S.name);
  }
schematic(S) {
  S.libraries(L) printf("Library: %s\n", L.name);
  }
```

UL_NET

Data members

`class`	UL_CLASS
`column`	string (see note)
`name`	string (`NET_NAME_LENGTH`)
`row`	string (see note)

Loop members

`pinrefs()`	UL_PINREF (see note)
`segments()`	UL_SEGMENT (see note)

See also UL_SHEET, UL_SCHEMATIC

Constants

`NET_NAME_LENGTH`	max. recommended length of a net name (used in formatted output only)

Note

The `pinrefs()` loop member can only be used if the net is in a schematic context.
The `segments()` loop member can only be used if the net is in a sheet context.

The `column` and `row` members return the column and row locations within the frame on the sheet on which this net is drawn. Since a net can extend over a certain area, each of these functions returns two values, separated by a blank. In case of `column` these are the left- and rightmost columns touched by the net, and in case of `row` it's the top- and bottommost row.

When determining the column and row of a net on a sheet, first the column and then the row within that column is taken into account. Here XREF labels take precedence over normal labels, which again take precedence over net wires.

If there is no frame on that sheet, `"? ?"` (two question marks) is returned. If any part of the net is placed outside the frame, either of the values may be `'?'` (question mark). These members can only be used in a sheet context.

Example

```
schematic(S) {
  S.nets(N) {
    printf("Net: %s\n", N.name);
    // N.segments(SEG) will NOT work here!
    }
  }
schematic(S) {
  S.sheets(SH) {
    SH.nets(N) {
      printf("Net: %s\n", N.name);
      N.segments(SEG) {
        SEG.wires(W) {
          printf("\tWire: (%d %d) (%d %d)\n",
                 W.x1, W.y1, W.x2, W.y2);
          }
        }
      }
    }
  }
```

UL_PACKAGE

Data members

`area`	UL_AREA
`description`	string
`headline`	string
`library`	string
`name`	string (`PACKAGE_NAME_LENGTH`)

Loop members

`circles()`	UL_CIRCLE
`contacts()`	UL_CONTACT
`dimensions()`	UL_DIMENSION
`frames()`	UL_FRAME
`holes()`	UL_HOLE
`polygons()`	UL_POLYGON (see note)
`rectangles()`	UL_RECTANGLE
`texts()`	UL_TEXT (see note)
`wires()`	UL_WIRE (see note)

See also UL_DEVICE, UL_ELEMENT, UL_LIBRARY

Constants

`PACKAGE_NAME_LENGTH`	max. recommended length of a package name (used in formatted output only)

Note

The `description` member returns the complete descriptive text as defined with the DESCRIPTION command, while the `headline` member returns only the first line of the description, without any HTML tags. When using the `description` text keep in mind that it may contain newline characters (`'\n'`).

If the UL_PACKAGE is derived from a UL_ELEMENT, the `texts()` member only loops through the non-detached texts of that element.

Polygons and wires belonging to contacts with arbitrary pad shapes are in UL_BOARD context only available through the loop members `polygons()` and `wires()` of this contact.

Example

```
library(L) {
  L.packages(PAC) {
    printf("Package: %s\n", PAC.name);
    PAC.contacts(C) {
      if (C.pad)
         printf("\tPad: %s, (%d %d)\n",
                 C.name, C.pad.x, C.pad.y);
      else if (C.smd)
         printf("\tSmd: %s, (%d %d)\n",
                 C.name, C.smd.x, C.smd.y);
      }
    }
  }
board(B) {
  B.elements(E) {
    printf("Element: %s, Package: %s\n", E.name, E.package.name);
    }
  }
```

UL_PAD

Data members

`angle`	real (`0.0`...`359.9`)
`diameter[layer]`	int
`drill`	int
`drillsymbol`	int
`elongation`	int
`flags`	int (`PAD_FLAG_...`)
`name`	string (`PAD_NAME_LENGTH`)
`shape[layer]`	int (`PAD_SHAPE_...`)
`signal`	string
`x, y`	int (center point, see note)

See also UL_PACKAGE, UL_CONTACT, UL_SMD

Constants

`PAD_FLAG_STOP`	generate stop mask
`PAD_FLAG_THERMALS`	generate thermals
`PAD_FLAG_FIRST`	use special "first pad" shape
`PAD_SHAPE_SQUARE`	square
`PAD_SHAPE_ROUND`	round
`PAD_SHAPE_OCTAGON`	octagon
`PAD_SHAPE_LONG`	long
`PAD_SHAPE_OFFSET`	offset
`PAD_NAME_LENGTH`	max. recommended length of a pad name (same as `CONTACT_NAME_LENGTH`)

Note

The parameters of the pad depend on the context in which it is accessed:

- if the pad is derived from a UL_LIBRARY context, the coordinates (`x, y`) and `angle` will be the same as defined in the package drawing
- in all other cases, they will have the actual values from the board

The diameter and shape of the pad depend on the layer for which they shall be retrieved, because they may be different in each layer depending on the Design Rules. If one of the layers LAYER_TOP...LAYER_BOTTOM, LAYER_TSTOP or LAYER_BSTOP is given as the index to the diameter or shape data member, the resulting value will be calculated according to the Design Rules. If LAYER_PADS is given, the raw value as defined in the library will be returned.

`drillsymbol` returns the number of the drill symbol that has been assigned to this drill diameter (see the manual for a list of defined drill symbols). A value of `0` means that no symbol has been assigned to this drill diameter.

`angle` defines how many degrees the pad is rotated counterclockwise around its center.

`elongation` is only valid for shapes PAD_SHAPE_LONG and PAD_SHAPE_OFFSET and defines how many percent the long side of such a pad is longer than its small side. This member returns 0 for any other pad shapes.

The value returned by `flags` must be masked with the `PAD_FLAG_...` constants to determine the individual flag settings, as in

```
if (pad.flags & PAD_FLAG_STOP) {
   ...
   }
```

Note that if your ULP just wants to draw the objects, you don't need to check these flags explicitly. The `diameter[]` and `shape[]` members will return the proper data; for instance, if `PAD_FLAG_STOP` is set, `diameter[LAYER_TSTOP]` will return `0`, which should result in nothing being drawn in that layer. The `flags` member is mainly for ULPs that want to create script files that create library objects.

Example

```
library(L) {
  L.packages(PAC) {
    PAC.contacts(C) {
      if (C.pad)
         printf("Pad: '%s', (%d %d), d=%d\n",
                C.name, C.pad.x, C.pad.y, C.pad.diameter[LAYER_BOTTOM]);
      }
    }
  }
```

UL_PART

Data members

`attribute[]`	string (see note)
`device`	UL_DEVICE

`deviceset`	UL_DEVICESET
`name`	string (`PART_NAME_LENGTH`)
`populate`	int (0=do not populate, 1=populate)
`value`	string (`PART_VALUE_LENGTH`)

Loop members

`attributes()`	UL_ATTRIBUTE (see note)
`instances()`	UL_INSTANCE (see note)
`variants()`	UL_VARIANT

See also UL_SCHEMATIC, UL_SHEET

Constants

`PART_NAME_LENGTH`	max. recommended length of a part name (used in formatted output only)
`PART_VALUE_LENGTH`	max. recommended length of a part value (used in formatted output only)

Note

The `attribute[]` member can be used to query a UL_PART for the value of a given attribute (see the second example below). The returned string is empty if there is no attribute by the given name, or if this attribute is explicitly empty.

When looping through the `attributes()` of a UL_PART, only the `name`, `value`, `defaultvalue` and `constant` members of the resulting UL_ATTRIBUTE objects are valid.

If the part is in a sheet context, the `instances()` loop member loops only through those instances that are actually used on that sheet. If the part is in a schematic context, all instances are looped through.

Example

```
schematic(S) {
  S.parts(P) printf("Part: %s\n", P.name);
  }

schematic(SCH) {
  SCH.parts(P) {
    if (P.attribute["REMARK"])
       printf("%s: %s\n", P.name, P.attribute["REMARK"]);
    }
  }
```

UL_PIN

Data members

`angle`	real (`0`, `90`, `180` and `270`)
`contact`	UL_CONTACT (deprecated, see note)
`direction`	int (`PIN_DIRECTION_...`)

`function`	int (`PIN_FUNCTION_FLAG_...`)
`length`	int (`PIN_LENGTH_...`)
`name`	string (`PIN_NAME_LENGTH`)
`net`	string (see note)
`route`	int (`CONTACT_ROUTE_...`)
`swaplevel`	int
`visible`	int (`PIN_VISIBLE_FLAG_...`)
`x, y`	int (connection point)

Loop members

`circles()`	UL_CIRCLE
`contacts()`	UL_CONTACT (see note)
`texts()`	UL_TEXT
`wires()`	UL_WIRE

See also UL_SYMBOL, UL_PINREF, UL_CONTACTREF

Constants

`PIN_DIRECTION_NC`	not connected
`PIN_DIRECTION_IN`	input
`PIN_DIRECTION_OUT`	output (totem-pole)
`PIN_DIRECTION_IO`	in/output (bidirectional)
`PIN_DIRECTION_OC`	open collector
`PIN_DIRECTION_PWR`	power input pin
`PIN_DIRECTION_PAS`	passive
`PIN_DIRECTION_HIZ`	high impedance output
`PIN_DIRECTION_SUP`	supply pin
`PIN_FUNCTION_FLAG_NONE`	no symbol
`PIN_FUNCTION_FLAG_DOT`	inverter symbol
`PIN_FUNCTION_FLAG_CLK`	clock symbol
`PIN_LENGTH_POINT`	no wire
`PIN_LENGTH_SHORT`	0.1 inch wire
`PIN_LENGTH_MIDDLE`	0.2 inch wire
`PIN_LENGTH_LONG`	0.3 inch wire
`PIN_NAME_LENGTH`	max. recommended length of a pin name (used in formatted output only)
`PIN_VISIBLE_FLAG_OFF`	no name drawn
`PIN_VISIBLE_FLAG_PAD`	pad name drawn
`PIN_VISIBLE_FLAG_PIN`	pin name drawn
`CONTACT_ROUTE_ALL`	must explicitly route to all contacts
`CONTACT_ROUTE_ANY`	may route to any contact

Note

The `contacts()` loop member loops through the contacts that have been assigned to the pin through a CONNECT command.

The `contact` data member returns the contact that has been assigned to the pin through a CONNECT command. ***This member is deprecated! It will work for backwards compatibility and as long as only one pad has been connected to the pin, but will cause a***

runtime error when used with a pin that is connected to more than one pad.

The coordinates (and layer, in case of an SMD) of the contact returned by the `contact` data member depend on the context in which it is called:

- if the pin is derived from a UL_PART that is used on a sheet, and if there is a corresponding element on the board, the resulting contact will have the coordinates as used on the board
- in all other cases, the coordinates of the contact will be the same as defined in the package drawing

The `name` data member always returns the name of the pin as it was defined in the library, with any `'@'` character for pins with the same name left intact (see the PIN command for details).
The `texts` loop member, on the other hand, returns the pin name (if it is visible) in the same way as it is displayed in the current drawing type.

The `net` data member returns the name of the net to which this pin is connected (only available in a schematic context).

Example

```
library(L) {
  L.symbols(S) {
    printf("Symbol: %s\n", S.name);
    S.pins(P) {
      printf("\tPin: %s, (%d %d)", P.name, P.x, P.y);
      if (P.direction == PIN_DIRECTION_IN)
         printf(" input");
      if ((P.function & PIN_FUNCTION_FLAG_DOT) != 0)
         printf(" inverted");
      printf("\n");
      }
    }
  }
```

UL_PINREF

Data members

`instance`	UL_INSTANCE
`part`	UL_PART
`pin`	UL_PIN

See also UL_SEGMENT, UL_CONTACTREF

Example

```
schematic(SCH) {
  SCH.sheets(SH) {
    printf("Sheet: %d\n", SH.number);
    SH.nets(N) {
      printf("\tNet: %s\n", N.name);
      N.segments(SEG) {
        SEG.pinrefs(P) {
          printf("connected to: %s, %s, %s\n",
                 P.part.name, P.instance.name, P.pin.name);
```

```
            }
          }
        }
      }
    }
```

UL_POLYGON

Data members

`isolate`	int
`layer`	int
`orphans`	int (0=off, 1=on)
`pour`	int (`POLYGON_POUR_...`)
`rank`	int
`spacing`	int
`thermals`	int (0=off, 1=on)
`width`	int

Loop members

`contours()`	UL_WIRE (see note)
`fillings()`	UL_WIRE
`wires()`	UL_WIRE

See also UL_BOARD, UL_PACKAGE, UL_SHEET, UL_SIGNAL, UL_SYMBOL

Constants

`POLYGON_POUR_SOLID`	solid
`POLYGON_POUR_HATCH`	hatch
`POLYGON_POUR_CUTOUT`	cutout

Note

The `contours()` and `fillings()` loop members loop through the wires that are used to draw the calculated polygon if it is part of a signal and the polygon has been calculated by the RATSNEST command. The `wires()` loop member always loops through the polygon wires as they were drawn by the user. For an uncalculated signal polygon `contours()` does the same as `wires()`, and `fillings()` does nothing.

If the `contours()` loop member is called without a second parameter, it loops through all of the contour wires, regardless whether they belong to a positive or a negative polygon. If you are interested in getting the positive and negative contour wires separately, you can call `contours()` with an additional integer parameter (see the second example below). The sign of that parameter determines whether a positive or a negative polygon will be handled, and the value indicates the index of that polygon. If there is no polygon with the given index, the statement will not be executed. Another advantage of this method is that you don't need to determine the beginning and end of a particular polygon yourself (by comparing coordinates). For any given index, the statement will be executed for all the wires of that polygon. With the second parameter `0` the behavior is the same as without a second parameter.

Polygon width

When using the `fillings()` loop member to get the fill wires of a solid polygon, make sure the *width* of the polygon is not zero (actually it should be quite a bit larger than zero, for example at least the hardware resolution of the output device you are going to draw on). **Filling a polygon with zero width may result in enormous amounts of data, since it will be calculated with the smallest editor resolution of 1/10000mm!**

Partial polygons

A calculated signal polygon may consist of several distinct parts (called *positive* polygons), each of which can contain extrusions (*negative* polygons) resulting from other objects being subtracted from the polygon. Negative polygons can again contain other positive polygons and so on.

The wires looped through by `contours()` always start with a positive polygon. To find out where one partial polygon ends and the next one begins, simply store the (x1,y1) coordinates of the first wire and check them against (x2,y2) of every following wire. As soon as these are equal, the last wire of a partial polygon has been found. It is also guaranteed that the second point (x2,y2) of one wire is identical to the first point (x1,y1) of the next wire in that partial polygon.

To find out where the "inside" and the "outside" of the polygon lays, take any contour wire and imagine looking from its point (x1,y1) to (x2,y2). The "inside" of the polygon is always on the right side of the wire. Note that if you simply want to draw the polygon you won't need all these details.

Example

```
board(B) {
  B.signals(S) {
    S.polygons(P) {
      int x0, y0, first = 1;
      P.contours(W) {
        if (first) {
           // a new partial polygon is starting
           x0 = W.x1;
           y0 = W.y1;
           }
        // ...
        // do something with the wire
        // ...
        if (first)
           first = 0;
        else if (W.x2 == x0 && W.y2 == y0) {
           // this was the last wire of the partial polygon,
           // so the next wire (if any) will be the first wire
           // of the next partial polygon
           first = 1;
           }
        }
      }
    }
  }

board(B) {
  B.signals(S) {
    S.polygons(P) {
      // handle only the "positive" polygons:
      int i = 1;
      int active;
      do {
         active = 0;
         P.contours(W, i) {
           active = 1;
           // do something with the wire
           }
         i++;
         } while (active);
      }
    }
  }
```

UL_RECTANGLE

Data members

`angle`	real (`0.0`...`359.9`)
`layer`	int
`x1, y1`	int (lower left corner)
`x2, y2`	int (upper right corner)

See also UL_BOARD, UL_PACKAGE, UL_SHEET, UL_SYMBOL

`angle` defines how many degrees the rectangle is rotated counterclockwise around its center. The center coordinates are given by `(x1+x2)/2` and `(y1+y2)/2`.

Example

```
board(B) {
  B.rectangles(R) {
    printf("Rectangle: (%d %d), (%d %d)\n",
           R.x1, R.y1, R.x2, R.y2);
    }
  }
```

UL_SCHEMATIC

Data members

`alwaysvectorfont`	int (`ALWAYS_VECTOR_FONT_..., see note`)
`description`	string
`grid`	UL_GRID
`headline`	string
`name`	string (see note)
`verticaltext`	int (`VERTICAL_TEXT_...`)
`xreflabel`	string

Loop members

`attributes()`	UL_ATTRIBUTE (see note)
`classes()`	UL_CLASS
`layers()`	UL_LAYER

`libraries()`	UL_LIBRARY
`nets()`	UL_NET
`parts()`	UL_PART
`sheets()`	UL_SHEET
`variantdefs()`	UL_VARIANTDEF

See also UL_BOARD, UL_LIBRARY

Constants

`ALWAYS_VECTOR_FONT_GUI`	alwaysvectorfont is set in the user interface dialog
`ALWAYS_VECTOR_FONT_PERSISTENT`	alwaysvectorfont is set persistent in this schematic
`VERTICAL_TEXT_UP`	reading direction for vertical texts: up
`VERTICAL_TEXT_DOWN`	reading direction for vertical texts: down

Note

The value returned by `alwaysvectorfont` can be used in boolean context or can be masked with the `ALWAYS_VECTOR_FONT_...` constants to determine the source of this setting, as in

```
if (sch.alwaysvectorfont) {
   // alwaysvectorfont is set in general
   }
if (sch.alwaysvectorfont & ALWAYS_VECTOR_FONT_GUI) {
   // alwaysvectorfont is set in the user interface
   }
```

The `name` member returns the full file name, including the directory.

The `xreflabel` member returns the format string used to display cross-reference labels.

The `attributes()` loop member loops through the *global* attributes.

Example

```
schematic(S) {
  S.parts(P) printf("Part: %s\n", P.name);
  }
```

UL_SEGMENT

Loop members

`junctions()`	UL_JUNCTION (see note)
`labels()`	UL_LABEL
`pinrefs()`	UL_PINREF (see note)
`texts()`	UL_TEXT (deprecated, see note)
`wires()`	UL_WIRE

See also UL_BUS, UL_NET

Note

The `junctions()` and `pinrefs()` loop members are only available for net segments.

The `texts()` loop member was used in older EAGLE versions to loop through the labels of a segment, and is only present for compatibility. It will not deliver the text of cross-reference labels at the correct position. Use the `labels()` loop member to access a segment's labels.

Example

```
schematic(SCH) {
  SCH.sheets(SH) {
    printf("Sheet: %d\n", SH.number);
    SH.nets(N) {
      printf("\tNet: %s\n", N.name);
      N.segments(SEG) {
        SEG.pinrefs(P) {
          printf("connected to: %s, %s, %s\n",
                 P.part.name, P.instance.name, P.pin.name);
        }
      }
    }
  }
}
```

UL_SHEET

Data members

`area`	UL_AREA
`description`	string
`headline`	string
`number`	int

Loop members

`busses()`	UL_BUS
`circles()`	UL_CIRCLE
`dimensions()`	UL_DIMENSION
`frames()`	UL_FRAME
`instances()`	UL_INSTANCE
`nets()`	UL_NET
`polygons()`	UL_POLYGON
`rectangles()`	UL_RECTANGLE
`texts()`	UL_TEXT
`wires()`	UL_WIRE

See also UL_SCHEMATIC

Example

```
schematic(SCH) {
  SCH.sheets(S) {
    printf("Sheet: %d\n", S.number);
    }
  }
```

UL_SIGNAL

Data members

airwireshidden	int
class	UL_CLASS
name	string (SIGNAL_NAME_LENGTH)

Loop members

contactrefs()	UL_CONTACTREF
polygons()	UL_POLYGON
vias()	UL_VIA
wires()	UL_WIRE

See also UL_BOARD

Constants

SIGNAL_NAME_LENGTH	max. recommended length of a signal name (used in formatted output only)

Example

```
board(B) {
  B.signals(S) printf("Signal: %s\n", S.name);
  }
```

UL_SMD

Data members

angle	real (0.0...359.9)
dx[layer], dy[layer]	int (size)
flags	int (SMD_FLAG_...)
layer	int (see note)
name	string (SMD_NAME_LENGTH)
roundness	int (see note)
signal	string
x, y	int (center point, see note)

See also UL_PACKAGE, UL_CONTACT, UL_PAD

Constants

SMD_FLAG_STOP	generate stop mask
SMD_FLAG_THERMALS	generate thermals
SMD_FLAG_CREAM	generate cream mask
SMD_NAME_LENGTH	max. recommended length of an smd name (same as CONTACT_NAME_LENGTH)

Note

The parameters of the smd depend on the context in which it is accessed:

- if the smd is derived from a UL_LIBRARY context, the coordinates (`x, y`), `angle`, `layer` and `roundness` of the smd will be the same as defined in the package drawing

- in all other cases, they will have the actual values from the board

If the `dx` and `dy` data members are called with an optional layer index, the data for that layer is returned according to the Design Rules. Valid layers are LAYER_TOP, LAYER_TSTOP and LAYER_TCREAM for an smd in the Top layer, and LAYER_BOTTOM, LAYER_BSTOP and LAYER_BCREAM for an smd in the Bottom layer, respectively.

`angle` defines how many degrees the smd is rotated counterclockwise around its center.

The value returned by `flags` must be masked with the `SMD_FLAG_...` constants to determine the individual flag settings, as in

```
if (smd.flags & SMD_FLAG_STOP) {
   ...
   }
```

Note that if your ULP just wants to draw the objects, you don't need to check these flags explicitly. The `dx[]` and `dy[]` members will return the proper data; for instance, if `SMD_FLAG_STOP` is set, `dx[LAYER_TSTOP]` will return `0`, which should result in nothing being drawn in that layer. The `flags` member is mainly for ULPs that want to create script files that create library objects.

Example

```
library(L) {
  L.packages(PAC) {
    PAC.contacts(C) {
      if (C.smd)
         printf("Smd: '%s', (%d %d), dx=%d, dy=%d\n",
                 C.name, C.smd.x, C.smd.y, C.smd.dx, C.smd.dy);
      }
    }
  }
```

UL_SYMBOL

Data members

`area`	UL_AREA
`description`	string
`headline`	string
`library`	string
`name`	string (`SYMBOL_NAME_LENGTH`)

Loop members

`circles()`	UL_CIRCLE
`dimensions()`	UL_DIMENSION
`frames()`	UL_FRAME
`rectangles()`	UL_RECTANGLE
`pins()`	UL_PIN
`polygons()`	UL_POLYGON
`texts()`	UL_TEXT (see note)
`wires()`	UL_WIRE

See also UL_GATE, UL_LIBRARY

Constants

SYMBOL_NAME_LENGTH	max. recommended length of a symbol name (used in formatted output only)

Note

If the UL_SYMBOL is derived from a UL_INSTANCE, the `texts()` member only loops through the non-detached texts of that instance.

Example

```
library(L) {
  L.symbols(S) printf("Sym: %s\n", S.name);
  }
```

UL_TEXT

Data members

align	int (ALIGN_...)
angle	real (0.0...359.9)
font	int (FONT_...)
layer	int
linedistance	int
mirror	int
ratio	int
size	int
spin	int
value	string
x, y	int (origin point)

Loop members

wires()	UL_WIRE (see note)

See also UL_BOARD, UL_PACKAGE, UL_SHEET, UL_SYMBOL

Constants

FONT_VECTOR	vector font
FONT_PROPORTIONAL	proportional font
FONT_FIXED	fixed font
ALIGN_BOTTOM_LEFT	bottom/left aligned
ALIGN_BOTTOM_CENTER	bottom/center aligned
ALIGN_BOTTOM_RIGHT	bottom/right aligned
ALIGN_CENTER_LEFT	center/left aligned
ALIGN_CENTER	centered
ALIGN_CENTER_RIGHT	center/right aligned
ALIGN_TOP_LEFT	top/left aligned
ALIGN_TOP_CENTER	top/center aligned
ALIGN_TOP_RIGHT	top/right aligned

Note

The `wires()` loop member always accesses the individual wires the text is composed of when using the vector font, even if the actual font is not `FONT_VECTOR`.

If the UL_TEXT is derived from a UL_ELEMENT or UL_INSTANCE context, the member values will be those of the actual text as located in the board or sheet drawing.

Example

```
board(B) {
  B.texts(T) {
    printf("Text: %s\n", T.value);
    }
  }
```

UL_VARIANTDEF

Data members

`name`	string

See also UL_VARIANT, UL_SCHEMATIC, UL_BOARD

Example

```
schematic(SCH) {
  SCH.variantdefs(VD) {
    printf("Variant: '%s'\n", VD.name);
    }
  }
```

UL_VARIANT

Data members

`populate`	int (0=do not populate, 1=populate)
`value`	string
`technology`	string
`variantdef`	UL_VARIANTDEF

See also UL_VARIANTDEF, UL_PART, UL_ELEMENT

Example

```
schematic(SCH) {
  SCH.parts(P) {
    P.variants(V) {
      printf("%s: %spopulate\n", V.variantdef.name, V.populate ? "" : "do not
");
      }
    }
  }
```

UL_VIA

Data members

`diameter[layer]`	int
`drill`	int
`drillsymbol`	int
`end`	int
`flags`	int (`VIA_FLAG_...`)
`shape[layer]`	int (`VIA_SHAPE_...`)
`start`	int
`x, y`	int (center point)

See also UL_SIGNAL

Constants

`VIA_FLAG_STOP`	always generate stop mask
`VIA_SHAPE_SQUARE`	square
`VIA_SHAPE_ROUND`	round
`VIA_SHAPE_OCTAGON`	octagon

Note

The diameter and shape of the via depend on the layer for which they shall be retrieved, because they may be different in each layer depending on the Design Rules. If one of the layers LAYER_TOP...LAYER_BOTTOM, LAYER_TSTOP or LAYER_BSTOP is given as the index to the diameter or shape data member, the resulting value will be calculated according to the Design Rules. If LAYER_VIAS is given, the raw value as defined in the via will be returned.

Note that `diameter` and `shape` will always return the diameter or shape that a via would have in the given layer, even if that particular via doesn't cover that layer (or if that layer isn't used in the layer setup at all).

`start` and `end` return the layer numbers in which that via starts and ends. The value of `start` will always be less than that of `end`.

`drillsymbol` returns the number of the drill symbol that has been assigned to this drill diameter (see the manual for a list of defined drill symbols). A value of `0` means that no symbol has been assigned to this drill diameter.

Example

```
board(B) {
  B.signals(S) {
    S.vias(V) {
      printf("Via: (%d %d)\n", V.x, V.y);
      }
    }
  }
```

UL_WIRE

Data members

`arc`	UL_ARC
`cap`	int (`CAP_...`)
`curve`	real
`layer`	int
`style`	int (`WIRE_STYLE_...`)
`width`	int
`x1, y1`	int (starting point)
`x2, y2`	int (end point)

Loop members

`pieces()`	UL_WIRE (see note)

See also UL_BOARD, UL_PACKAGE, UL_SEGMENT, UL_SHEET, UL_SIGNAL, UL_SYMBOL, UL_ARC

Constants

`CAP_FLAT`	flat arc ends
`CAP_ROUND`	round arc ends
`WIRE_STYLE_CONTINUOUS`	continuous
`WIRE_STYLE_LONGDASH`	long dash
`WIRE_STYLE_SHORTDASH`	short dash
`WIRE_STYLE_DASHDOT`	dash dot

Wire Style

A UL_WIRE that has a *style* other than `WIRE_STYLE_CONTINUOUS` can use the `pieces()` loop member to access the individual segments that constitute for example a dashed wire. If `pieces()` is called for a UL_WIRE with `WIRE_STYLE_CONTINUOUS`, a single segment will be accessible which is just the same as the original UL_WIRE. The `pieces()` loop member can't be called from a UL_WIRE that itself has been returned by a call to `pieces()` (this would cause an infinite recursion).

Arcs at Wire level

Arcs are basically wires, with a few additional properties. At the first level arcs are treated exactly the same as wires, meaning they have a start and an end point, a width, layer and wire style. In addition to these an arc, at the wire level, has a *cap* and a *curve* parameter. *cap* defines whether the arc endings are round or flat, and *curve* defines the "curvature" of the arc. The valid range for *curve* is `-360`..`+360`, and its value means what part of a full circle the arc consists of. A value of `90`, for instance, would result in a `90°` arc, while `180` would give you a semicircle. The maximum value of `360` can only be reached theoretically, since this would mean that the arc consists of a full circle, which, because the start and end points have to lie on the circle, would have to have an infinitely large diameter. Positive values for *curve* mean that the arc is drawn in a mathematically positive sense (i.e. counterclockwise). If *curve* is `0`, the arc is a straight line ("no curvature"), which is actually a wire.

The *cap* parameter only has a meaning for actual arcs, and will always return `CAP_ROUND` for a straight wire.

Whether or not an UL_WIRE is an arc can be determined by checking the boolean return value of the `arc` data member. If it returns `0`, we have a straight wire, otherwise an arc. If `arc` returns a non-zero value it may be further dereferenced to access the UL_ARC specific parameters start and end angle, radius and center point. Note that you may only need these additional parameters if you are going to draw the arc or process it in other ways where the actual shape is important.

Example

```
board(B) {
  B.wires(W) {
    printf("Wire: (%d %d) (%d %d)\n",
           W.x1, W.y1, W.x2, W.y2);
    }
  }
```

Definitions

The data items to be used in a User Language Program must be defined before they can be used.

There are three kinds of definitions:

- Constant Definitions
- Variable Definitions
- Function Definitions

The scope of a *constant* or *variable* definition goes from the line in which it has been defined to the end of the current block, or to the end of the User Language Program, if the definition appeared outside any block.

The scope of a *function* definition goes from the closing brace (`}`) of the function body to the end of the User Language Program.

Constant Definitions

Constants are defined using the keyword `enum`, as in

```
enum { a, b, c };
```

which would define the three constants `a`, `b` and `c`, giving them the values `0`, `1` and `2`, respectively.

Constants may also be initialized to specific values, like

```
enum { a, b = 5, c };
```

where `a` would be `0`, `b` would be `5` and `c` would be `6`.

Variable Definitions

The general syntax of a *variable definition* is

```
[numeric] type identifier [= initializer][, ...];
```

where `type` is one of the data or object types, `identifier` is the name of the variable, and `initializer` is a optional initial value.

Multiple variable definitions of the same `type` are separated by commas (`,`).

If `identifier` is followed by a pair of brackets (`[]`), this defines an array of variables of the given `type`. The size of an array is automatically adjusted at runtime.

The optional keyword `numeric` can be used with string arrays to have them sorted alphanumerically by the sort() function.

By default (if no `initializer` is present), data variables are set to `0` (or `""`, in case of a string), and object variables are "invalid".

Examples

`int i;`	defines an int variable named `i`
`string s = "Hello";`	defines a string variable named `s` and initializes it to `"Hello"`
`real a, b = 1.0, c;`	defines three real variables named `a`, `b` and `c`, initializing `b` to the value `1.0`
`int n[] = { 1, 2, 3 };`	defines an array of int, initializing the first three elements to `1`, `2` and `3`
`numeric string names[];`	defines a string array that can be sorted alphanumerically
`UL_WIRE w;`	defines a UL_WIRE object named `w`

The members of array elements of object types can't be accessed directly:

```
UL_SIGNAL signals[];
...
UL_SIGNAL s = signals[0];
printf("%s", s.name);
```

Function Definitions

You can write your own User Language functions and call them just like the Builtin Functions.

The general syntax of a *function definition* is

```
type identifier(parameters)
{
  statements
}
```

where `type` is one of the data or object types, `identifier` is the name of the function, `parameters` is a list of comma separated parameter definitions, and `statements` is a sequence of statements.

Functions that do not return a value have the type `void`.

A function must be defined **before** it can be called, and function calls can not be recursive (a function cannot call itself).

The statements in the function body may modify the values of the parameters, but this will not have any effect on the arguments of the function call.

Execution of a function can be terminated by the `return` statement. Without any `return` statement the function body is executed until it's closing brace (`}`).

A call to the `exit()` function will terminate the entire User Language Program.

The special function `main()`

If your User Language Program contains a function called `main()`, that function will be explicitly called as the main function, and it's return value will be the return value of the program.

Command line arguments are available to the program through the global Builtin Variables `argc` and `argv`.

Example

```
int CountDots(string s)
{
  int dots = 0;
  for (int i = 0; s[i]; ++i)
      if (s[i] == '.')
         ++dots;
  return dots;
}
string dotted = "This.has.dots...";
output("test") {
  printf("Number of dots: %d\n",
                 CountDots(dotted));
  }
```

Operators

The following table lists all of the User Language operators, in order of their precedence (*Unary* having the highest precedence, *Comma* the lowest):

Unary	`! ~ + - ++ --`
Multiplicative	`* / %`
Additive	`+ -`
Shift	`<< >>`
Relational	`< <= > >=`
Equality	`== !=`
Bitwise AND	`&`
Bitwise XOR	`^`
Bitwise OR	`|`
Logical AND	`&&`
Logical OR	`||`
Conditional	`?:`
Assignment	`= *= /= %= += -= &= ^= |= <<= >>=`
Comma	`,`

Associativity is **left to right** for all operators, except for *Unary*, *Conditional* and *Assignment*, which are **right to left** associative.

The normal operator precedence can be altered by the use of parentheses.

Bitwise Operators

Bitwise operators work only with data types `char` and `int`.

Unary	
`~`	Bitwise (1's) complement
Binary	
`<<`	Shift left
`>>`	Shift right
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
Assignment	
`&=`	Assign bitwise AND
`^=`	Assign bitwise XOR
`|=`	Assign bitwise OR
`<<=`	Assign left shift
`>>=`	Assign right shift

Logical Operators

Logical operators work with expressions of any data type.

Unary	
`!`	Logical NOT
Binary	
`&&`	Logical AND
`||`	Logical OR

Using a `string` expression with a logical operator checks whether the string is empty.

Using an Object Type with a logical operator checks whether that object contains valid data.

Comparison Operators

Comparison operators work with expressions of any data type, except Object Types.

`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to
`==`	Equal to
`!=`	Not equal to

Evaluation Operators

Evaluation operators are used to evaluate expressions based on a condition, or to group a sequence of expressions and have them evaluated as one expression.

`?:`	Conditional
`,`	Comma

The *Conditional* operator is used to make a decision within an expression, as in

```
int a;
// ...code that calculates 'a'
string s = a ? "True" : "False";
```

which is basically the same as

```
int a;
string s;
// ...code that calculates 'a'
if (a)
   s = "True";
else
   s = "False";
```

but the advantage of the conditional operator is that it can be used in an expression.

The *Comma* operator is used to evaluate a sequence of expressions from left to right, using the type and value of the right operand as the result.

Note that arguments in a function call as well as multiple variable declarations also use commas as delimiters, but in that case this is **not** a comma operator!

Arithmetic Operators

Arithmetic operators work with data types `char`, `int` and `real` (except for `++`, `--`, `%` and `%=`).

Unary	
`+`	Unary plus
`-`	Unary minus
`++`	Pre- or postincrement
`--`	Pre- or postdecrement
Binary	
`*`	Multiply
`/`	Divide
`%`	Remainder (modulus)
`+`	Binary plus
`-`	Binary minus
Assignment	
`=`	Simple assignment
`*=`	Assign product
`/=`	Assign quotient
`%=`	Assign remainder (modulus)
`+=`	Assign sum
`-=`	Assign difference

See also String Operators

String Operators

String operators work with data types `char`, `int` and `string`. The left operand must always be of type `string`.

Binary	
`+`	Concatenation

Assignment

=	Simple assignment
+=	Append to string

The **+** operator concatenates two strings, or adds a character to the end of a string and returns the resulting string.

The **+=** operator appends a string or a character to the end of a given string.

See also Arithmetic Operators

Expressions

An *expression* can be one of the following:

- Arithmetic Expression
- Assignment Expression
- String Expression
- Comma Expression
- Conditional Expression
- Function Call

Expressions can be grouped using parentheses, and may be recursive, meaning that an expression can consist of subexpressions.

Arithmetic Expression

An *arithmetic expression* is any combination of numeric operands and an arithmetic operator or a bitwise operator.

Examples

```
a + b
c++
m << 1
```

Assignment Expression

An *assignment expression* consists of a variable on the left side of an assignment operator, and an expression on the right side.

Examples

```
a = x + 42
b += c
s = "Hello"
```

String Expression

A *string expression* is any combination of string and char operands and a string operator.

Examples

```
s + ".brd"
t + 'x'
```

Comma Expression

A *comma expression* is a sequence of expressions, delimited by the comma operator

Comma expressions are evaluated left to right, and the result of a comma expression is the type and value of the rightmost expression.

Example

```
i++, j++, k++
```

Conditional Expression

A *conditional expression* uses the conditional operator to make a decision within an expression.

Example

```
int a;
// ...code that calculates 'a'
string s = a ? "True" : "False";
```

Function Call

A *function call* transfers the program flow to a user defined function or a builtin function. The formal parameters defined in the function definition are replaced with the values of the expressions used as the actual arguments of the function call.

Example

```
int p = strchr(s, 'b');
```

Statements

A *statement* can be one of the following:

- Compound Statement
- Control Statement
- Expression Statement
- Builtin Statement
- Constant Definition
- Variable Definition

Statements specify the flow of control as a User Language Program executes. In absence of specific control statements, statements are executed sequentially in the order of appearance in the ULP file.

Compound Statement

A *compound statement* (also known as *block*) is a list (possibly empty) of statements enclosed in matching braces (`{}`). Syntactically, a block can be considered to be a single statement, but it also controls the scoping of identifiers. An identifier declared within a block has a scope starting at the point of declaration and ending at the closing brace.

Compound statements can be nested to any depth.

Expression Statement

An *expression statement* is any expression followed by a semicolon.

An expression statement is executed by evaluating the expression. All side effects of this evaluation are completed before the next statement is executed. Most expression statements are assignments or function calls.

A special case is the *empty statement*, consisting of only a semicolon. An empty statement does nothing, but it may be useful in situations where the ULP syntax expects a statement but your program does not need one.

Control Statements

Control statements are used to control the program flow.

Iteration statements are

`do...while`
`for`
`while`

Selection statements are

`if...else`
`switch`

Jump statements are

`break`
`continue`
`return`

break

The *break* statement has the general syntax

```
break;
```

and immediately terminates the **nearest** enclosing do...while, for, switch or while statement. This also applies to *loop members* of object types.

Since all of these statements can be intermixed and nested to any depth, take care to ensure that your `break` exits from the correct statement.

continue

The *continue* statement has the general syntax

```
continue;
```

and immediately transfers control to the test condition of the **nearest** enclosing do...while, while, or for statement, or to the increment expression of the **nearest** enclosing for statement.

Since all of these statements can be intermixed and nested to any depth, take care to ensure that your `continue` affects the correct statement.

do...while

The *do...while* statement has the general syntax

```
do statement while (condition);
```

and executes the `statement` until the `condition` expression becomes zero.

The `condition` is tested **after** the first execution of `statement`, which means that the statement is always executed at least one time.

If there is no `break` or `return` inside the `statement`, the `statement` must affect the value of the `condition`, or `condition` itself must change during evaluation in order to avoid an endless loop.

Example

```
string s = "Trust no one!";
int i = -1;
do {
   ++i;
   } while (s[i]);
```

for

The *for* statement has the general syntax

```
for ([init]; [test]; [inc]) statement
```

and performs the following steps:

1. If an initializing expression `init` is present, it is executed.
2. If a `test` expression is present, it is executed. If the result is nonzero (or if there is no `test` expression at all), the `statement` is executed.
3. If an `inc` expression is present, it is executed.
4. Finally control returns to step 2.

If there is no `break` or `return` inside the `statement`, the `inc` expression (or the `statement`) must affect the value of the `test` expression, or `test` itself must change during evaluation in order to avoid an endless loop.

The initializing expression `init` normally initializes one or more loop counters. It may also define a new variable as a loop counter. The scope of such a variable is valid until the end of the active block.

Example

```
string s = "Trust no one!";
int sum = 0;
for (int i = 0; s[i]; ++i)
    sum += s[i]; // sums up the characters in s
```

if...else

The *if...else* statement has the general syntax

```
if (expression)
   t_statement
[else
   f_statement]
```

The conditional `expression` is evaluated, and if its value is nonzero the `t_statement` is executed. Otherwise the `f_statement` is executed in case there is an `else` clause.

An `else` clause is always matched to the last encountered `if` without an `else`. If this is not what you want, you need to use braces to group the statements, as in

```
if (a == 1) {
   if (b == 1)
      printf("a == 1 and b == 1\n");
   }
else
   printf("a != 1\n");
```

return

A function with a return type other than `void` must contain at least one *return* statement with the syntax

```
return expression;
```

where `expression` must evaluate to a type that is compatible with the function's return type. The value of `expression` is the value returned by the function.

If the function is of type `void`, a `return` statement without an `expression` can be used to return from the function call.

switch

The *switch* statement has the general syntax

```
switch (sw_exp) {
  case case_exp: case_statement
  ...
  [default: def_statement]
  }
```

and allows for the transfer of control to one of several `case`-labeled statements, depending on the value of `sw_exp` (which must be of integral type).

Any `case_statement` can be labeled by one or more `case` labels. The `case_exp` of each `case` label must evaluate to a constant integer which is unique within it's enclosing `switch` statement.

There can also be at most one `default` label.

After evaluating `sw_exp`, the `case_exp` are checked for a match. If a match is found, control passes to the `case_statement` with the matching `case` label.

If no match is found and there is a `default` label, control passes to `def_statement`. Otherwise none of the statements in the `switch` is executed.

Program execution is not affected when `case` and `default` labels are encountered. Control simply passes through the labels to the following statement.

To stop execution at the end of a group of statements for a particular `case`, use the break statement.

Example

```
string s = "Hello World";
int vowels = 0, others = 0;
for (int i = 0; s[i]; ++i)
    switch (toupper(s[i])) {
      case 'A':
      case 'E':
      case 'I':
      case 'O':
      case 'U': ++vowels;
                break;
      default: ++others;
      }
printf("There are %d vowels in '%s'\n", vowels, s);
```

while

The *while* statement has the general syntax

```
while (condition) statement
```

and executes the `statement` as long as the `condition` expression is not zero.

The `condition` is tested **before** the first possible execution of `statement`, which means that the statement may never be executed if `condition` is initially zero.

If there is no `break` or `return` inside the `statement`, the `statement` must affect the value of the `condition`, or `condition` itself must change during evaluation in order to avoid an endless loop.

Example

```
string s = "Trust no one!";
int i = 0;
while (s[i])
      ++i;
```

Builtins

Builtins are *Constants*, *Variables*, *Functions* and *Statements* that provide additional information and allow for data manipulations.

- Builtin Constants
- Builtin Variables
- Builtin Functions
- Builtin Statements

Builtin Constants

Builtin constants are used to provide information about object parameters, such as maximum recommended name length, flags etc.

Many of the object types have their own **Constants** section which lists the builtin constants for that particular object (see e.g. UL_PIN).

The following builtin constants are defined in addition to the ones listed for the various object types:

`EAGLE_VERSION`	EAGLE program version number (int)
`EAGLE_RELEASE`	EAGLE program release number (int)
`EAGLE_SIGNATURE`	a string containing EAGLE program name, version and copyright information
`EAGLE_PATH`	a string containing the complete path of the EAGLE executable
`EAGLE_DIR`	a string containing the directory of the EAGLE installation (`$EAGLEDIR`)
`EAGLE_HOME`	a string containing the user's home directory when starting EAGLE (`$HOME`)
`OS_SIGNATURE`	a string containing a signature of the operating system (e.g. Mac..., Windows... or Linux)
`REAL_EPSILON`	the minimum positive real number such that `1.0 + REAL_EPSILON != 1.0`
`REAL_MAX`	the largest possible real value
`REAL_MIN`	the smallest possible (positive!) real value the smallest representable number is `-REAL_MAX`
`INT_MAX`	the largest possible int value
`INT_MIN`	the smallest possible int value
`PI`	the value of "pi" (3.14..., real)
`usage`	a string containing the text from the `#usage` directive

These builtin constants contain the directory paths defined in the directories dialog, with any of the special variables (`$HOME` and `$EAGLEDIR`) replaced by their actual values. Since each path can consist of several directories, these constants are string arrays with an individual directory in each member. The first empty member marks the end of the path:

`path_lbr[]`	Libraries

`path_dru[]`	Design Rules
`path_ulp[]`	User Language Programs
`path_scr[]`	Scripts
`path_cam[]`	CAM Jobs
`path_epf[]`	Projects

When using these constants to build a full file name, you need to use a directory separator, as in

```
string s = path_lbr[0] + '/' + "mylib.lbr";
```

The libraries that are currently in use through the USE command:

`used_libraries[]`

Builtin Variables

Builtin variables are used to provide information at runtime.

`int argc`	number of arguments given to the RUN command
`string argv[]`	arguments given to the RUN command (`argv[0]` is the full ULP file name)

Builtin Functions

Builtin functions are used to perform specific tasks, like printing formatted strings, sorting data arrays or the like.

You may also write your own functions and use them to structure your User Language Program.

The builtin functions are grouped into the following categories:

- Character Functions
- File Handling Functions
- Mathematical Functions
- Miscellaneous Functions
- Network Functions
- Printing Functions
- String Functions
- Time Functions
- Object Functions
- XML Functions

Alphabetical reference of all builtin functions:

- abs()
- acos()
- asin()
- atan()
- ceil()
- cfgget()
- cfgset()

- clrgroup()
- country()
- cos()
- exit()
- exp()
- fdlsignature()
- filedir()
- fileerror()
- fileext()
- fileglob()
- filename()
- fileread()
- filesetext()
- filesize()
- filetime()
- floor()
- frac()
- inch2u()
- ingroup()
- isalnum()
- isalpha()
- iscntrl()
- isdigit()
- isgraph()
- islower()
- isprint()
- ispunct()
- isspace()
- isupper()
- isxdigit()
- language()
- log()
- log10()
- lookup()
- max()
- mic2u()
- mil2u()
- min()
- mm2u()
- neterror()
- netget()
- netpost()
- palette()
- pow()

- printf()
- round()
- setgroup()
- setvariant()
- sin()
- sort()
- sprintf()
- sqrt()
- status()
- strchr()
- strjoin()
- strlen()
- strlwr()
- strrchr()
- strrstr()
- strsplit()
- strstr()
- strsub()
- strtod()
- strtol()
- strupr()
- strxstr()
- system()
- t2day()
- t2dayofweek()
- t2hour()
- t2minute()
- t2month()
- t2second()
- t2string()
- t2year()
- tan()
- time()
- tolower()
- toupper()
- trunc()
- u2inch()
- u2mic()
- u2mil()
- u2mm()
- variant()
- xmlattribute()
- xmlattributes()
- xmlelement()

- xmlelements()
- xmltags()
- xmltext()

Character Functions

Character functions are used to manipulate single characters.

The following character functions are available:

- isalnum()
- isalpha()
- iscntrl()
- isdigit()
- isgraph()
- islower()
- isprint()
- ispunct()
- isspace()
- isupper()
- isxdigit()
- tolower()
- toupper()

is...()

Function

Check whether a character falls into a given category.

Syntax

```
int isalnum(char c);
int isalpha(char c);
int iscntrl(char c);
int isdigit(char c);
int isgraph(char c);
int islower(char c);
int isprint(char c);
int ispunct(char c);
int isspace(char c);
int isupper(char c);
int isxdigit(char c);
```

Returns

The `is...` functions return nonzero if the given character falls into the category, zero otherwise.

Character categories

`isalnum`	letters (`A` to `Z` or `a` to `z`) or digits (`0` to `9`)
`isalpha`	letters (`A` to `Z` or `a` to `z`)
`iscntrl`	delete characters or ordinary control characters (`0x7F` or `0x00` to `0x1F`)

`isdigit`	digits (`0` to `9`)
`isgraph`	printing characters (except space)
`islower`	lowercase letters (`a` to `z`)
`isprint`	printing characters (`0x20` to `0x7E`)
`ispunct`	punctuation characters (`iscntrl` or `isspace`)
`isspace`	space, tab, carriage return, new line, vertical tab, or formfeed (`0x09` to `0x0D`, `0x20`)
`isupper`	uppercase letters (`A` to `Z`)
`isxdigit`	hex digits (`0` to `9`, `A` to `F`, `a` to `f`)

Example

```
char c = 'A';
if (isxdigit(c))
   printf("%c is hex\n", c);
else
   printf("%c is not hex\n", c);
```

to...()

Function

Convert a character to upper- or lowercase.

Syntax

```
char tolower(char c);
char toupper(char c);
```

Returns

The `tolower` function returns the converted character if `c` is uppercase. All other characters are returned unchanged.
The `toupper` function returns the converted character if `c` is lowercase. All other characters are returned unchanged.

See also strupr, strlwr

File Handling Functions

Filename handling functions are used to work with file names, sizes and timestamps.

The following file handling functions are available:

- fileerror()
- fileglob()
- filedir()
- fileext()
- filename()
- fileread()
- filesetext()
- filesize()
- filetime()

See output() for information about how to write into a file.

fileerror()

Function

Returns the status of I/O operations.

Syntax

`int fileerror();`

Returns

The `fileerror` function returns `0` if everything is ok.

See also output, printf, fileread

`fileerror` checks the status of any I/O operations that have been performed since the last call to this function and returns `0` if everything was ok. If any of the I/O operations has caused an error, a value other than `0` will be returned.

You should call `fileerror` before any I/O operations to reset any previous error state, and call it again after the I/O operations to see if they were successful.

When `fileerror` returns a value other than `0` (thus indicating an error) a proper error message has already been given to the user.

Example

```
fileerror();
output("file.txt", "wt") {
  printf("Test\n");
  }
if (fileerror())
   exit(1);
```

fileglob()

Function

Perform a directory search.

Syntax

`int fileglob(string &array[], string pattern);`

Returns

The `fileglob` function returns the number of entries copied into `array`.

See also dlgFileOpen(), dlgFileSave()

`fileglob` performs a directory search using `pattern`.

`pattern` may contain `'*'` and `'?'` as wildcard characters. If `pattern` ends with a `'/'`, the contents of the given directory will be returned.

Names in the resulting `array` that end with a `'/'` are directory names.

The `array` is sorted alphabetically, with the directories coming first.

The special entries `'.'` and `'..'` (for the current and parent directories) are never returned in the `array`.

If `pattern` doesn't match, or if you don't have permission to search the given directory, the resulting `array` will be empty.

Note for Windows users

The directory delimiter in the `array` is always a **forward slash**. This makes sure User Language Programs will work platform independently. In the `pattern` the **backslash** (`'\'`) is also treated as a directory delimiter.

Sorting filenames under Windows is done case insensitively.

Example

```
string a[];
int n = fileglob(a, "*.brd");
```

Filename Functions

Function

Split a filename into its separate parts.

Syntax

```
string filedir(string file);
string fileext(string file);
string filename(string file);
string filesetext(string file, string newext);
```

Returns

`filedir` returns the directory of `file` (including the drive letter under Windows).
`fileext` returns the extension of `file`.
`filename` returns the file name of `file` (including the extension).
`filesetext` returns `file` with the extension set to `newext`.

See also Filedata Functions

Example

```
if (board) board(B) {
  output(filesetext(B.name, ".out")) {
    ...
    }
  }
```

Filedata Functions

Function

Gets the timestamp and size of a file.

Syntax

```
int filesize(string filename);
int filetime(string filename);
```

Returns

`filesize` returns the size (in byte) of the given file.
`filetime` returns the timestamp of the given file in a format to be used with the time functions.

See also time, Filename Functions

Example

```
board(B)
  printf("Board: %s\nSize: %d\nTime: %s\n",
         B.name, filesize(B.name),
         t2string(filetime(B.name)));
```

File Input Functions

File input functions are used to read data from files.

The following file input is available:

- fileread()

See output() for information about how to write into a file.

fileread()

Function
Reads data from a file.

Syntax
`int fileread(`*`dest`*`, string file);`

Returns
`fileread` returns the number of objects read from the file.
The actual meaning of the return value depends on the type of `dest`.

See also lookup, strsplit, fileerror

If `dest` is a character array, the file will be read as raw binary data and the return value reflects the number of bytes read into the character array (which is equal to the file size).

If `dest` is a string array, the file will be read as a text file (one line per array member) and the return value will be the number of lines read into the string array. Newline characters will be stripped.

If `dest` is a string, the entire file will be read into that string and the return value will be the length of that string (which is not necessarily equal to the file size, if the operating system stores text files with "cr/lf" instead of a "newline" character).

Example

```
char b[];
int nBytes = fileread(b, "data.bin");
string lines[];
int nLines = fileread(lines, "data.txt");
string text;
int nChars = fileread(text, "data.txt");
```

Mathematical Functions

Mathematical functions are used to perform mathematical operations.

The following mathematical functions are available:

- abs()
- acos()
- asin()
- atan()
- ceil()
- cos()
- exp()
- floor()
- frac()
- log()
- log10()
- max()
- min()
- pow()
- round()
- sin()
- sqrt()
- trunc()
- tan()

Error Messages

If the arguments of a mathematical function call lead to an error, the error message will show the actual values of the arguments. Thus the statements

```
real x = -1.0;
real r = sqrt(2 * x);
```

will lead to the error message

```
Invalid argument in call to 'sqrt(-2)'
```

Absolute, Maximum and Minimum Functions

Function

Absolute, maximum and minimum functions.

Syntax

```
type abs(type x);
type max(type x, type y);
type min(type x, type y);
```

Returns

`abs` returns the absolute value of `x`.
`max` returns the maximum of `x` and `y`.
`min` returns the minimum of `x` and `y`.

The return type of these functions is the same as the (larger) type of the arguments. `type` must be one of `char`, `int` or `real`.

Example

```
real x = 2.567, y = 3.14;
printf("The maximum is %f\n", max(x, y));
```

Rounding Functions

Function

Rounding functions.

Syntax

```
real ceil(real x);
real floor(real x);
real frac(real x);
real round(real x);
real trunc(real x);
```

Returns

`ceil` returns the smallest integer not less than `x`.
`floor` returns the largest integer not greater than `x`.
`frac` returns the fractional part of `x`.
`round` returns `x` rounded to the nearest integer.
`trunc` returns the integer part of `x`.

Example

```
real x = 2.567;
printf("The rounded value of %f is %f\n", x, round(x));
```

Trigonometric Functions

Function

Trigonometric functions.

Syntax

```
real acos(real x);
real asin(real x);
real atan(real x);
real cos(real x);
real sin(real x);
real tan(real x);
```

Returns

`acos` returns the arc cosine of `x`.
`asin` returns the arc sine of `x`.
`atan` returns the arc tangent of `x`.
`cos` returns the cosine of `x`.
`sin` returns the sine of `x`.
`tan` returns the tangent of `x`.

Constants

`PI`	the value of "pi"

(3.14...)

Note

Angles are given in radian.

Example

```
real x = PI / 2;
printf("The sine of %f is %f\n", x, sin(x));
```

Exponential Functions

Function
Exponential Functions.

Syntax

```
real exp(real x);
real log(real x);
real log10(real x);
real pow(real x, real y);
real sqrt(real x);
```

Returns

`exp`	returns the exponential *e* to the power of `x`.
`log`	returns the natural logarithm of `x`.
`log10`	returns the base 10 logarithm of `x`.
`pow`	returns the value of `x` to the power of `y`.
`sqrt`	returns the square root of `x`.

Example

```
real x = 2.1;
printf("The square root of %f is %f\n", x, sqrt(x));
printf("The 3rd root of %f is %f\n", x, pow(x, 1.0/3));
```

Miscellaneous Functions

Miscellaneous functions are used to perform various tasks.

The following miscellaneous functions are available:

- country()
- exit()
- fdlsignature()
- language()
- lookup()
- palette()
- sort()
- status()
- system()
- Configuration Parameters

- Unit Conversions

Configuration Parameters

Function

Store and retrieve configuration parameters.

Syntax

```
string cfgget(string name[, string default]);
void cfgset(string name, string value);
```

Returns

`cfgget` returns the value of the parameter stored under the given `name`. If no such parameter has been stored, yet, the value of the optional `default` is returned (or an empty string, if no `default` is given).

The `cfgget` function retrieves values that have previously been stored with a call to `cfgset()`.

The `cfgset` function sets the parameter with the given `name` to the given `value`.

The valid characters for `name` are `'A'-'Z'`, `'a'-'z'`, `'0'-'9'`, `'.'` and `'_'`. Parameter names are case sensitive.

The parameters are stored in the user's eaglerc file. To ensure that different User Language Programs don't overwrite each other's parameters in case they use the same parameter names, it is recommended to put the name of the ULP at the beginning of the parameter name. For example, a ULP named `mytool.ulp` that uses a parameter named `MyParam` could store that parameter under the name

```
mytool.MyParam
```

Because the configuration parameters are stored in the eaglerc file, which also contains all of EAGLE's other user specific parameters, it is also possible to access the EAGLE parameters with `cfgget()` and `cfgset()`. In order to make sure no ULP parameters collide with any EAGLE parameters, the EAGLE parameters must be prefixed with `"EAGLE:"`, as in

```
EAGLE:Option.XrefLabelFormat
```

Note that there is no documentation of all of EAGLE's internal parameters and how they are stored in the eaglerc file. Also, be very careful when changing any of these parameters! As with the eaglerc file itself, you should only manipulate these parameters if you know what you are doing! Some EAGLE parameters may require a restart of EAGLE for changes to take effect.

In the eaglerc file the User Language parameters are stored with the prefix `"ULP:"`. Therefore this prefix may be optionally put in front of User Language parameter names, as in

```
ULP:mytool.MyParam
```

Example

```
string MyParam = cfgget("mytool.MyParam", "SomeDefault");
MyParam = "OtherValue";
```

```
cfgset("mytool.MyParam", MyParam);
```

country()

Function

Returns the country code of the system in use.

Syntax

```
string country();
```

Returns

`country` returns a string consisting of two uppercase characters that identifies the country used on the current system. If no such country setting can be determined, the default "US" will be returned.

See also language

Example

```
dlgMessageBox("Your country code is: " + country());
```

exit()

Function

Exits from a User Language Program.

Syntax

```
void exit(int result);
void exit(string command);
```

See also RUN

The `exit` function terminates execution of a User Language Program.
If an integer `result` is given it will be used as the return value of the program.
If a string `command` is given, that command will be executed as if it were entered into the command line immediately after the RUN command. In that case the return value of the ULP is set to `EXIT_SUCCESS`.

Constants

`EXIT_SUCCESS`	return value for successful program execution (value `0`)
`EXIT_FAILURE`	return value for failed program execution (value `-1`)

fdlsignature()

Function

Calculates a digital signature for Premier Farnell's *Design Link*.

Syntax

```
string fdlsignature(string s, string key);
```

The `fdlsignature` function is used to calculate a digital signature when accessing Premier Farnell's *Design Link* interface.

language()

Function

Returns the language code of the system in use.

Syntax

`string language();`

Returns

`language` returns a string consisting of two lowercase characters that identifies the language used on the current system. If no such language setting can be determined, the default "en" will be returned.

See also country

The `language` function can be used to make a ULP use different message string, depending on which language the current system is using.

In the example below all the strings used in the ULP are listed in the string array `I18N[]`, preceeded by a string containing the various language codes supported by this ULP. Note the `vtab` characters used to separate the individual parts of each string (they are important for the `lookup` function) and the use of the commas to separate the strings. The actual work is done in the function `tr()`, which returns the translated version of the given string. If the original string can't be found in the `I18N` array, or there is no translation for the current language, the original string will be used untranslated.

The first language defined in the `I18N` array must be the one in which the strings used throughout the ULP are written, and should generally be English in order to make the program accessible to the largest number of users.

Example

```
string I18N[] = {
  "en\v"
  "de\v"
  "it\v"
  ,
  "I18N Demo\v"
  "Beispiel für Internationalisierung\v"
  "Esempio per internazionalizzazione\v"
  ,
  "Hello world!\v"
  "Hallo Welt!\v"
  "Ciao mondo!\v"
  ,
  "+Ok\v"
  "+Ok\v"
  "+Approvazione\v"
  ,
  "-Cancel\v"
  "-Abbrechen\v"
  "-Annullamento\v"
  };
int Language = strstr(I18N[0], language()) / 3;
string tr(string s)
{
  string t = lookup(I18N, s, Language, '\v');
  return t ? t : s;
}
```

```
dlgDialog(tr("I18N Demo")) {
  dlgHBoxLayout dlgSpacing(350);
  dlgLabel(tr("Hello world!"));
  dlgHBoxLayout {
    dlgPushButton(tr("+Ok")) dlgAccept();
    dlgPushButton(tr("-Cancel")) dlgReject();
    }
  };
```

lookup()

Function

Looks up data in a string array.

Syntax

```
string lookup(string array[], string key, int field_index[,
char separator]);
string lookup(string array[], string key, string field_name[,
char separator]);
```

Returns

`lookup` returns the value of the field identified by `field_index` or `field_name`.
If the field doesn't exist, or no string matching `key` is found, an empty string is returned.

See also fileread, strsplit

An `array` that can be used with `lookup()` consists of strings of text, each string representing one data record.

Each data record contains an arbitrary number of fields, which are separated by the character `separator` (default is `'\t'`, the tabulator). The first field in a record is used as the `key` and is numbered `0`.

All records must have unique `key` fields and none of the `key` fields may be empty - otherwise it is undefined which record will be found.

If the first string in the `array` contains a "Header" record (i.e. a record where each field describes its contents), using `lookup` with a `field_name` string automatically determines the index of that field. This allows using the `lookup` function without exactly knowing which field index contains the desired data.
It is up to the user to make sure that the first record actually contains header information.

If the `key` parameter in the call to `lookup()` is an empty string, the first string of the `array` will be used. This allows a program to determine whether there is a header record with the required field names.

If a field contains the `separator` character, that field must be enclosed in double quotes (as in `"abc;def"`, assuming the semicolon (`';'`) is used as separator). The same applies if the field contains double quotes (`"`), in which case the double quotes inside the field have to be doubled (as in `"abc;""def"";ghi"`, which would be `abc;"def";ghi`).
It is best to use the default "tab" separator, which doesn't have these problems (no field can contain a tabulator).

Here's an example data file (`';'` has been used as separator for better readability):

```
Name;Manufacturer;Code;Price
7400;Intel;I-01-234-97;$0.10
```

```
68HC12;Motorola;M68HC1201234;$3.50
```

Example

```
string OrderCodes[];
if (fileread(OrderCodes, "ordercodes") > 0) {
   if (lookup(OrderCodes, "", "Code", ';')) {
      schematic(SCH) {
        SCH.parts(P) {
          string OrderCode;
          // both following statements do exactly the same:
          OrderCode = lookup(OrderCodes, P.device.name, "Code", ';');
          OrderCode = lookup(OrderCodes, P.device.name, 2, ';');
          }
        }
      }
   else
      dlgMessageBox("Missing 'Code' field in file 'ordercodes');
   }
```

palette()

Function

Returns color palette information.

Syntax

`int palette(int index[, int type]);`

Returns

The `palette` function returns an integer ARGB value in the form 0xaarrggbb, or the type of the currently used palette (depending on the value of `index`).

The `palette` function returns the ARGB value of the color with the given `index` (which may be in the range 0..PALETTE_ENTRIES-1). If `type` is not given (or is `-1`) the palette assigned to the current editor window will be used. Otherwise `type` specifies which color palette to use (PALETTE_BLACK, PALETTE_WHITE or PALETTE_COLORED).

The special value `-1` for `index` makes the function return the type of the palette that is currently in use by the editor window.

If either `index` or `type` is out of range, an error message will be given and the ULP will be terminated.

Constants

`PALETTE_TYPES`	the number of palette types (3)
`PALETTE_BLACK`	the black background palette (0)
`PALETTE_WHITE`	the white background palette (1)
`PALETTE_COLORED`	the colored background palette (2)
`PALETTE_ENTRIES`	the number of colors per palette (64)

sort()

Function

Sorts an array or a set of arrays.

Syntax

```
void sort(int number, array1[, array2,...]);
```

The `sort` function either directly sorts a given `array1`, or it sorts a set of arrays (starting with `array2`), in which case `array1` is supposed to be an array of **int**, which will be used as a pointer array.

In any case, the `number` argument defines the number of items in the array(s).

Sorting a single array

If the `sort` function is called with one single array, that array will be sorted directly, as in the following example:

```
string A[];
int n = 0;
A[n++] = "World";
A[n++] = "Hello";
A[n++] = "The truth is out there...";
sort(n, A);
for (int i = 0; i < n; ++i)
    printf(A[i]);
```

Sorting a set of arrays

If the `sort` function is called with more than one array, the first array must be an array of **int**, while all of the other arrays may be of any array type and hold the data to be sorted. The following example illustrates how the first array will be used as a pointer:

```
numeric string Nets[], Parts[], Instances[], Pins[];
int n = 0;
int index[];
schematic(S) {
  S.nets(N) N.pinrefs(P) {
    Nets[n] = N.name;
    Parts[n] = P.part.name;
    Instances[n] = P.instance.name;
    Pins[n] = P.pin.name;
    ++n;
    }
  sort(n, index, Nets, Parts, Instances, Pins);
  for (int i = 0; i < n; ++i)
      printf("%-8s %-8s %-8s %-8s\n",
             Nets[index[i]], Parts[index[i]],
             Instances[index[i]], Pins[index[i]]);
  }
```

The idea behind this is that one net can have several pins connected to it, and in a netlist you might want to have the net names sorted, and within one net you also want the part names sorted and so on.

Note the use of the keyword `numeric` in the string arrays. This causes the strings to be sorted in a way that takes into account a numeric part at the end of the strings, which leads to IC1, IC2,... IC9, IC10 instead of the alphabetical order IC1, IC10, IC2,...IC9.

When sorting a set of arrays, the first (index) array must be of type `int` and need not be initialized. Any contents the index array might have before calling the `sort` function will be overwritten by the resulting index values.

status()

Function
Displays a status message in the status bar.

Syntax
```
void status(string message);
```

See also dlgMessageBox()

The `status` function displays the given `message` in the status bar of the editor window in which the ULP is running.

system()

Function
Executes an external program.

Syntax
```
int system(string command);
```

Returns
The `system` function returns the exit status of the command. This is typically `0` if everything was ok, and non-zero in case of an error.

The `system` function executes the external program given by the `command` string, and waits until the program ends.

Input/Output redirection

If the external program shall read its standard input from (or write its standard output to) a particular file, input/output needs to be redirected.

On **Linux** and **Mac OS X** this is done by simply adding a `'<'` or `'>'` to the command line, followed by the desired file name, as in

```
system("program < infile > outfile");
```

which runs `program` and makes it read from `infile` and write to `outfile`.

On **Windows** you have to explicitly run a command processor to do this, as in

```
system("cmd.exe /c program < infile > outfile");
```

(on DOS based Windows systems use `command.com` instead of `cmd.exe`).

Background execution

The `system` function waits until the given program has ended. This is useful for programs that only run for a few seconds, or completely take over the user's attention.

If an external program runs for a longer time, and you want the system call to return immediately, without waiting for the program to end, you can simply add an `'&'` to the command string under **Linux** and **Mac OS X**, as in

```
system("program &");
```

Under Windows you need to explicitly run a command processor to do this, as in

```
system("cmd.exe /c start program");
```

(on DOS based Windows systems use `command.com` instead of `cmd.exe`).

Example

```
int result = system("simulate -f filename");
```

This would call a simulation program, giving it a file which the ULP has just created. Note that `simulate` here is just an example, it is not part of the EAGLE package!

If you want to have control over what system commands are actually executed, you can write a wrapper function that prompts the user for confirmation before executing the command, like

```
int MySystem(string command)
{
  if (dlgMessageBox("!Ok to execute the following command?<p><tt>" + command +
"</tt>", "&Yes", "&No") == 0)
     return system(command);
  return -1;
}
int result = MySystem("simulate -f filename");
```

Unit Conversions

Function

Converts internal units.

Syntax

```
real u2inch(int n);
real u2mic(int n);
real u2mil(int n);
real u2mm(int n);
int inch2u(real n);
int mic2u(real n);
int mil2u(real n);
int mm2u(real n);
```

Returns

`u2inch` returns the value of `n` in *inch*.
`u2mic` returns the value of `n` in *microns* (1/1000mm).
`u2mil` returns the value of `n` in *mil* (1/1000inch).
`u2mm` returns the value of `n` in *millimeters*.
`inch2u` returns the value of `n` (which is in *inch*) as internal units.
`mic2u` returns the value of `n` (which is in *microns*) as internal units.
`mil2u` returns the value of `n` (which is in *mil*) as internal units.
`mm2u` returns the value of `n` (which is in *millimeters*) as internal units.

See also UL_GRID

EAGLE stores all coordinate and size values as `int` values with a resolution of 1/320000mm (0.003125µ). The above unit conversion functions can be used to convert these internal units to the desired measurement units, and vice versa.

Example

```
board(B) {
  B.elements(E) {
    printf("%s at (%f, %f)\n", E.name,
           u2mm(E.x), u2mm(E.y));
    }
  }
```

Network Functions

Network functions are used to access remote sites on the Internet.

The following network functions are available:

- neterror()
- netget()
- netpost()

neterror()

Function
Returns the error message of the most recent network function call.

Syntax
`string neterror(void);`

Returns
`neterror` returns a textual message describing the error that occurred in the most recent call to a network function.
If no error has occurred, the return value is an empty string.

See also netget, netpost

The `neterror` function should be called after any of the other network functions has returned a negative value, indicating that an error has occurred. The return value of `neterror` is a textual string that can be presented to the user.

Example

```
string Result;
if (netget(Result, "http://www.cadsoft.de/cgi-bin/http-test?see=me&hear=them")
>= 0) {
   // process Result
   }
else
   dlgMessageBox(neterror());
```

netget()

Function

Performs a GET request on the network.

Syntax

`int netget(*dest*, string url[, int timeout]);`

Returns

`netget` returns the number of objects read from the network.
The actual meaning of the return value depends on the type of `dest`.
In case of an error, a negative value is returned and neterror() may be called to display an error message to the user.

See also netpost, neterror, fileread

The `netget` function sends the given `url` to the network and stores the result in the `dest` variable.
If no network activity has occurred for `timeout` seconds, the connection will be terminated. The default timeout is 20 seconds.
The `url` must contain the protocol to use (HTTP, HTTPS or FTP) and can contain name=value pairs of parameters, as in

```
http://www.cadsoft.de/cgi-bin/http-test?see=me&hear=them
ftp://ftp.cadsoft.de/eagle/userfiles/README
```

If a user id and password is required to access a remote site, these can be given as

```
https://userid:password@www.secret-site.com/...
```

If `dest` is a character array, the result will be treated as raw binary data and the return value reflects the number of bytes stored in the character array.

If `dest` is a string array, the result will be treated as text data (one line per array member) and the return value will be the number of lines stored in the string array. Newline characters will be stripped.

If `dest` is a string, the result will be stored in that string and the return value will be the length of the string. Note that in case of binary data the result is truncated at the first occurrence of a byte with the value 0x00.

If you need to use a proxy to access the Internet with HTTP or HTTPS, you can set that up in the "Configure" dialog under "Help/Check for Update" in the Control Panel.

Example

```
string Result;
if (netget(Result, "http://www.cadsoft.de/cgi-bin/http-test?see=me&hear=them")
>= 0) {
   // process Result
   }
else
   dlgMessageBox(neterror());
```

netpost()

Function

Performs a POST request on the network.

Syntax

`int netpost(`*`dest`*`, string url, string data[, int timeout[, string content_type] ]);`

Returns

`netpost` returns the number of objects read from the network.
The actual meaning of the return value depends on the type of `dest`.
In case of an error, a negative value is returned and neterror() may be called to display an error message to the user.

See also netget, neterror, fileread

The `netpost` function sends the given `data` to the given `url` on the network and stores the result in the `dest` variable.
If no network activity has occurred for `timeout` seconds, the connection will be terminated. The default timeout is 20 seconds.
If `content_type` is given, it overwrites the default content type of `"text/html; charset=utf-8"`.
The `url` must contain the protocol to use (HTTP or HTTPS).

If a user id and password is required to access a remote site, these can be given as

```
https://userid:password@www.secret-site.com/...
```

If `dest` is a character array, the result will be treated as raw binary data and the return value reflects the number of bytes stored in the character array.

If `dest` is a string array, the result will be treated as text data (one line per array member) and the return value will be the number of lines stored in the string array. Newline characters will be stripped.

If `dest` is a string, the result will be stored in that string and the return value will be the length of the string. Note that in case of binary data the result is truncated at the first occurrence of a byte with the value 0x00.

If you need to use a proxy to access the Internet with HTTP or HTTPS, you can set that up in the "Configure" dialog under "Help/Check for Update" in the Control Panel.

Example

```
string Data = "see=me\nhear=them";
string Result;
if (netpost(Result, "http://www.cadsoft.de/cgi-bin/http-test", Data) >= 0) {
   // process Result
   }
else
   dlgMessageBox(neterror());
```

Printing Functions

Printing functions are used to print formatted strings.

The following printing functions are available:

- printf()
- sprintf()

printf()

Function

Writes formatted output to a file.

Syntax

```
int printf(string format[, argument, ...]);
```

Returns

The `printf` function returns the number of characters written to the file that has been opened by the most recent output statement.

In case of an error, `printf` returns `-1`.

See also sprintf, output, fileerror

Format string

The format string controls how the arguments will be converted, formatted and printed. There must be exactly as many arguments as necessary for the format. The number and type of arguments will be checked against the format, and any mismatch will lead to an error message.

The format string contains two types of objects - *plain characters* and *format specifiers*:

- Plain characters are simply copied verbatim to the output
- Format specifiers fetch arguments from the argument list and apply formatting to them

Format specifiers

A format specifier has the following form:

```
% [flags] [width] [.prec] type
```

Each format specification begins with the percent character (`%`). After the `%` comes the following, in this order:

- an optional sequence of flag characters, `[flags]`
- an optional width specifier, `[width]`
- an optional precision specifier, `[.prec]`
- the conversion type character, `type`

Conversion type characters

`d`	**signed** decimal **int**
`o`	**unsigned** octal **int**
`u`	**unsigned** decimal **int**
`x`	**unsigned** hexadecimal **int** (with **a**, **b**,...)

`X`	**unsigned** hexadecimal **int** (with **A**, **B**,...)
`f`	**signed real** value of the form `[-]dddd.dddd`
`e`	**signed real** value of the form `[-]d.dddde[±]ddd`
`E`	same as `e`, but with **E** for exponent
`g`	**signed real** value in either `e` or `f` form, based on given value and precision
`G`	same as `g`, but with **E** for exponent if `e` format used
`c`	single character
`s`	character string
`%`	the `%` character is printed

Flag characters

The following flag characters can appear in any order and combination.

`"-"`	the formatted item is left-justified within the field; normally, items are right-justified
`"+"`	a signed, positive item will always start with a plus character (`+`); normally, only negative items begin with a sign
`" "`	a signed, positive item will always start with a space character; if both `"+"` and `" "` are specified, `"+"` overrides `" "`

Width specifiers

The width specifier sets the minimum field width for an output value.

Width is specified either directly, through a decimal digit string, or indirectly, through an asterisk (`*`). If you use an asterisk for the width specifier, the preceding argument (which must be an `int`) to the one being formatted (with this format specifier) determines the minimum output field width.

In no case does a nonexistent or small field width cause truncation of a field. If the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result.

`n`	At least *n* characters are printed. If the output value has less than *n* characters, the output is padded with blanks (right-padded if `"-"` flag given, left-padded otherwise).
`0`*`n`*	At least *n* characters are printed. If the output value has less than *n* characters, it is filled on the left with zeros.
`*`	The argument list supplies the width specifier, which must precede the actual argument being formatted.

Precision specifiers

A precision specifier always begins with a period (`.`) to separate it from any preceding width specifier. Then, like width, precision is specified either directly through a decimal digit string, or indirectly, through an asterisk (`*`). If you use an asterisk for the precision specifier, the preceding argument (which must be an `int`) to the one being formatted (with this format specifier) determines the precision.

none	Precision set to default.
`.0`	For `int` types, precision is set to default; for `real` types, no decimal point is printed.
`.`*`n`*	*n* characters or *n* decimal places are printed. If the output value has more than *n* characters the output might be truncated or rounded (depending on the type character).
`*`	The argument list supplies the precision specifier, which must precede the actual argument being formatted.

Default precision values

`douxX`	1
`eEf`	6
`gG`	all significant digits
`c`	no effect
`s`	print entire string

How precision specification (`.n`) affects conversion

`douxX`	.*n* specifies that at least *n* characters are printed. If the input argument has less than *n* digits, the output value is left-padded with zeros. If the input argument has more than *n* digits, the output value is **not** truncated.
`eEf`	.*n* specifies that *n* characters are printed after the decimal point, and the last digit printed is rounded.
`gG`	.*n* specifies that at most *n* significant digits are printed.
`c`	.*n* has no effect on the output.
`s`	.*n* specifies that no more than *n* characters are printed.

Binary zero characters

Unlike sprintf, the `printf` function can print binary zero characters (0x00).

```
char c = 0x00;
printf("%c", c);
```

Example

```
int i = 42;
real r = 3.14;
char c = 'A';
string s = "Hello";
printf("Integer: %8d\n", i);
printf("Hex:     %8X\n", i);
printf("Real:    %8f\n", r);
printf("Char:    %-8c\n", c);
printf("String:  %-8s\n", s);
```

sprintf()

Function

Writes formatted output into a string.

Syntax

```
int sprintf(string result, string format[, argument, ...]);
```

Returns

The `sprintf` function returns the number of characters written into the `result` string.

In case of an error, `sprintf` returns `-1`.

See also printf

Format string

See printf.

Binary zero characters

Note that `sprintf` can not return strings with embedded binary zero characters (0x00). If the resulting string contains a binary zero character, any characters following that zero character will be dropped. Use printf if you need to output binary data.

Example

```
string result;
int number = 42;
sprintf(result, "The number is %d", number);
```

String Functions

String functions are used to manipulate character strings.

The following string functions are available:

- strchr()
- strjoin()
- strlen()
- strlwr()
- strrchr()
- strrstr()
- strsplit()
- strstr()
- strsub()
- strtod()
- strtol()
- strupr()
- strxstr()

strchr()

Function

Scans a string for the first occurrence of a given character.

Syntax

```
int strchr(string s, char c[, int index]);
```

Returns

The `strchr` function returns the integer offset of the character in the string, or `-1` if the character does not occur in the string.

See also strrchr, strstr

If `index` is given, the search starts at that position. Negative values are counted from the end of the string.

Example

```
string s = "This is a string";
char c = 'a';
int pos = strchr(s, c);
if (pos >= 0)
   printf("The character %c is at position %d\n", c, pos);
else
   printf("The character was not found\n");
```

strjoin()

Function

Joins a string array to form a single string.

Syntax

`string strjoin(string array[], char separator);`

Returns

The `strjoin` function returns the combined entries of `array`.

See also strsplit, lookup, fileread

`strjoin` joins all entries in `array`, delimited by the given `separator` and returns the resulting string.

If `separator` is the newline character (`'\n'`) the resulting string will be terminated with a newline character. This is done to have a text file that consists of N lines (each of which is terminated with a newline) and is read in with the fileread() function and split into an array of N strings to be joined to the original string as read from the file.

Example

```
string a[] = { "Field 1", "Field 2", "Field 3" };
string s = strjoin(a, ':');
```

strlen()

Function

Calculates the length of a string.

Syntax

`int strlen(string s);`

Returns

The `strlen` function returns the number of characters in the string.

Example

```
string s = "This is a string";
int l = strlen(s);
printf("The string is %d characters long\n", l);
```

strlwr()

Function

Converts uppercase letters in a string to lowercase.

Syntax

`string strlwr(string s);`

Returns

The `strlwr` function returns the modified string. The original string (given as parameter) is not changed.

See also strupr, tolower

Example

```
string s = "This Is A String";
string r = strlwr(s);
printf("Prior to strlwr: %s - after strlwr: %s\n", s, r);
```

strrchr()

Function

Scans a string for the last occurrence of a given character.

Syntax

`int strrchr(string s, char c[, int index]);`

Returns

The `strrchr` function returns the integer offset of the character in the string, or `-1` if the character does not occur in the string.

See also strchr, strrstr

If `index` is given, the search starts at that position. Negative values are counted from the end of the string.

Example

```
string s = "This is a string";
char c = 'a';
int pos = strrchr(s, c);
if (pos >= 0)
   printf("The character %c is at position %d\n", c, pos);
else
   printf("The character was not found\n");
```

strrstr()

Function

Scans a string for the last occurrence of a given substring.

Syntax

`int strrstr(string s1, string s2[, int index]);`

Returns

The `strrstr` function returns the integer offset of the first character of s2 in s1, or `-1` if the substring does not occur in the string.

See also strstr, strrchr

If `index` is given, the search starts at that position. Negative values are counted from the end of the string.

Example

```
string s1 = "This is a string", s2 = "is a";
int pos = strrstr(s1, s2);
if (pos >= 0)
   printf("The substring starts at %d\n", pos);
else
   printf("The substring was not found\n");
```

strsplit()

Function
: Splits a string into separate fields.

Syntax
: `int strsplit(string &array[], string s, char separator);`

Returns
: The `strsplit` function returns the number of entries copied into `array`.

See also strjoin, lookup, fileread

`strsplit` splits the string `s` at the given `separator` and stores the resulting fields in the `array`.

If `separator` is the newline character (`'\n'`) the last field will be silently dropped if it is empty. This is done to have a text file that consists of N lines (each of which is terminated with a newline) and is read in with the fileread() function to be split into an array of N strings. With any other `separator` an empty field at the end of the string will count, so `"a:b:c:"` will result in 4 fields, the last of which is empty.

Example

```
string a[];
int n = strsplit(a, "Field 1:Field 2:Field 3", ':');
```

strstr()

Function
: Scans a string for the first occurrence of a given substring.

Syntax
: `int strstr(string s1, string s2[, int index]);`

Returns
: The `strstr` function returns the integer offset of the first character of s2 in s1, or `-1` if the substring does not occur in the string.

See also strrstr, strchr, strxstr

If `index` is given, the search starts at that position. Negative values are counted from the end of the string.

Example

```
string s1 = "This is a string", s2 = "is a";
int pos = strstr(s1, s2);
if (pos >= 0)
   printf("The substring starts at %d\n", pos);
else
   printf("The substring was not found\n");
```

strsub()

Function

Extracts a substring from a string.

Syntax

`string strsub(string s, int start[, int length]);`

Returns

The `strsub` function returns the substring indicated by the `start` and `length` value.

The value for `length` must be positive, otherwise an empty string will be returned. If `length` is ommitted, the rest of the string (beginning at `start`) is returned.

If `start` points to a position outside the string, an empty string is returned.

Example

```
string s = "This is a string";
string t = strsub(s, 4, 7);
printf("The extracted substring is: %s\n", t);
```

strtod()

Function

Converts a string to a real value.

Syntax

`real strtod(string s);`

Returns

The `strtod` function returns the numerical representation of the given string as a `real` value. Conversion ends at the first character that does not fit into the format of a real constant. If an error occurs during conversion of the string `0.0` will be returned.

See also strtol

Example

```
string s = "3.1415";
real r = strtod(s);
printf("The value is %f\n", r);
```

strtol()

Function

Converts a string to an integer value.

Syntax

`int strtol(string s);`

Returns

The `strtol` function returns the numerical representation of the given string as an `int` value. Conversion ends at the first character that does not fit into the format of an integer constant. If an error occurs during conversion of the string `0` will be returned.

See also strtod

Example

```
string s = "1234";
int i = strtol(s);
printf("The value is %d\n", i);
```

strupr()

Function

Converts lowercase letters in a string to uppercase.

Syntax

`string strupr(string s);`

Returns

The `strupr` function returns the modified string. The original string (given as parameter) is not changed.

See also strlwr, toupper

Example

```
string s = "This Is A String";
string r = strupr(s);
printf("Prior to strupr: %s - after strupr: %s\n", s, r);
```

strxstr()

Function

Scans a string for the first occurrence of a given regular expression.

Syntax

`int strxstr(string s1, string s2[, int index[, int &length]]);`

Returns

The `strxstr` function returns the integer offset of the substring in s1 that matches the regular expression in s2, or `-1` if the regular expression does not match in the string.

See also strstr, strchr, strrstr

If `index` is given, the search starts at that position. Negative values are counted from the end of the string.

If `length` is given, the actual length of the matching substring is returned in that variable.

Regular expressions allow you to find a pattern within a text string. For instance, the regular expression "i.*a" would find a sequence of characters that starts with an 'i', followed by any character ('.') any number of times ('*'), and ends with an 'a'. It would match on "is a" as well as "is this a" or "ia".
Details on regular expressions can be found, for instance, in the book *Mastering Regular Expressions* by Jeffrey E. F. Friedl.

Example

```
string s1 = "This is a string", s2 = "i.*a";
int len = 0;
int pos = strxstr(s1, s2, 0, len);
if (pos >= 0)
   printf("The substring starts at %d and is %d charcaters long\n", pos, len);
else
   printf("The substring was not found\n");
```

Time Functions

Time functions are used to get and process time and date information.

The following time functions are available:

- t2day()
- t2dayofweek()
- t2hour()
- t2minute()
- t2month()
- t2second()
- t2string()
- t2year()
- time()
- timems()

time()

Function
Gets the current system time.

Syntax
`int time(void);`

Returns
The `time` function returns the current system time as the number of seconds elapsed since a system dependent reference date.

See also Time Conversions, filetime, timems()

Example

```
int CurrentTime = time();
```

timems()

Function

Gets the number of milliseconds since the start of the ULP.

Syntax

```
int timems(void);
```

Returns

The `timems` function returns the number of milliseconds since the start of the ULP.

After 86400000 milliseconds (i.e. every 24 hours), the value starts at 0 again.

See also time

Example

```
int elapsed = timems();
```

Time Conversions

Function

Convert a time value to day, month, year etc.

Syntax

```
int t2day(int t);
int t2dayofweek(int t);
int t2hour(int t);
int t2minute(int t);
int t2month(int t);
int t2second(int t);
int t2year(int t);

string t2string(int t[, string format]);
```

Returns

`t2day` returns the day of the month (`1`..`31`)
`t2dayofweek` returns the day of the week (`0`=sunday..`6`)
`t2hour` returns the hour (`0`..`23`)
`t2minute` returns the minute (`0`..`59`)
`t2month` returns the month (`0`..`11`)
`t2second` returns the second (`0`..`59`)
`t2year` returns the year (including century!)
`t2string` returns a formatted string containing date and time

See also time

The `t2string` function without the optional `format` parameter converts the given time `t` into a country specific string in local time.

If `t2string` is called with a `format` string, that format is used to determine what the result should look like.

The following expressions can be used in a `format` string:

d	the day as a number without a leading zero (1 to 31)

dd	the day as a number with a leading zero (01 to 31)
ddd	the abbreviated localized day name (e.g. "Mon" to "Sun")
dddd	the long localized day name (e.g. "Monday" to "Sunday")
M	the month as a number without a leading zero (1-12)
MM	the month as a number with a leading zero (01-12)
MMM	the abbreviated localized month name (e.g. "Jan" to "Dec")
MMMM	the long localized month name (e.g. "January" to "December")
yy	the year as a two digit number (00-99)
yyyy	the year as a four digit number
h	the hour without a leading zero (0 to 23 or 1 to 12 if AM/PM display)
hh	the hour with a leading zero (00 to 23 or 01 to 12 if AM/PM display)
m	the minute without a leading zero (0 to 59)
mm	the minute with a leading zero (00 to 59)
s	the second without a leading zero (0 to 59)
ss	the second with a leading zero (00 to 59)
z	the milliseconds without leading zeros (always 0, since the given time only has a one second resolution)
zzz	the milliseconds with leading zeros (always 000, since the given time only has a one second resolution)
AP	use AM/PM display (*AP* will be replaced by either "AM" or "PM")
ap	use am/pm display (*ap* will be replaced by either "am" or "pm")
U	display the given time as UTC (must be the first character; default is local time)

All other characters will be copied "as is". Any sequence of characters that are enclosed in singlequotes will be treated as text and not be used as an expression. Two consecutive single quotes ('') are replaced by a single quote in the output.

Example

```
int t = time();
printf("It is now %02d:%02d:%02d\n",
       t2hour(t), t2minute(t), t2second(t));
printf("ISO time is %s\n", t2string(t, "Uyyyy-MM-dd hh:mm:ss"));
```

Object Functions

Object functions are used to access common information about objects.

The following object functions are available:

- clrgroup()
- ingroup()
- setgroup()
- setvariant()
- variant()

clrgroup()

Function

Clears the group flags of an object.

Syntax

```
void clrgroup(object);
```

See also ingroup(), setgroup(), GROUP command

The `clrgroup()` function clears the group flags of the given object, so that it is no longer part of the previously defined group.

When applied to an object that contains other objects (like a UL_BOARD or UL_NET) the group flags of all contained objects are cleared recursively, but with analogous limitations like for setgroup().

Example

```
board(B) {
  B.elements(E)
    clrgroup(E);
  }
```

ingroup()

Function

Checks whether an object is in the group.

Syntax

`int ingroup(object);`

Returns

The `ingroup` function returns a non-zero value if the given object is in the group.

See also clrgroup(), setgroup(), GROUP command

If a group has been defined in the editor, the `ingroup()` function can be used to check whether a particular object is part of the group.

Objects with a single coordinate that are individually selectable in the current drawing (like UL_TEXT, UL_VIA, UL_CIRCLE etc.) return a non-zero value in a call to `ingroup()` if that coordinate is within the defined group.

A UL_WIRE returns 0, 1, 2 or 3, depending on whether none, the first, the second or both of its end points are in the group.

A UL_RECTANGLE and UL_FRAME returns a non-zero value if one or more of its corners are in the group. The value has bit 0 set for the upper right corner, bit 1 for the upper left, bit 2 for the bottom left, and bit 3 for the bottom right corner.

Higher ranking objects that have no coordinates (UL_NET, UL_SEGMENT, UL_SIGNAL, UL_POLYGON) or that are actually not available as drawing objects (UL_SHEET, UL_DEVICESET, UL_SYMBOL, UL_PACKAGE), return a non-zero value if one or more of the objects within them are in the group. For details on the object hierarchies see Object Types.
UL_CONTACTREF and UL_PINREF, though not having coordinates of their own, return a non-zero value if the referenced UL_CONTACT or UL_PIN, respectively, is within the group.
For other not selectable objects like UL_GRID, UL_VARIANT or wires of a UL_TEXT or UL_FRAME object, the behaviour of `ingroup()` is undefined and therefore should not be used.

Identifying the context menu object

If the ULP is started from a context menu the selected object can be accessed by the group mechansim (see RUN): A one element group is made from the selected object. So it can be identified with `ingroup()`.

Example

```
output("group.txt") {
  board(B) {
    B.elements(E) {
      if (ingroup(E))
         printf("Element %s is in the group\n", E.name);
      }
    }
  }
```

setgroup()

Function

Sets the group flags of an object.

Syntax

```
void setgroup(object[, int flags]);
```

See also clrgroup(), ingroup(), GROUP command

The `setgroup()` function sets the group flags of the given object, so that it becomes part of the group.

If no `flags` are given, the object is added to the group as a whole (i.e. all of its selection points, in case it has more than one).

If `flags` has a non-zero value, only the group flags of the given points of the object are set. For a UL_WIRE this means that `'1'` sets the group flag of the first point, `'2'` that of the second point, and `'3'` sets both. Any previously set group flags remain unchanged by a call to `setgroup()`.

When applied to an object that contains other objects (like a UL_BOARD or UL_NET) the group flags of all contained objects are set recursively with following limitations:
It's not the case for UL_LIBRARY and UL_SCHEMATIC. Subordinate objects that are not selectable or not inidividually selectable are not flagged (e.g. UL_GRID or UL_VARIANT objects or wires of UL_TEXT or UL_FRAME objects).
For details on the object hierarchies see Object Types.

Example

```
board(B) {
  B.elements(E)
    setgroup(E);
  }
```

setvariant()

Function

Sets the current assembly variant.

Syntax

```
int setvariant(string name);
```

See also variant(), UL_VARIANTDEF, VARIANT command

The `setvariant()` function sets the current assembly variant to the one given by `name`. This can be used to loop through all of the parts and "see" their data exactly as defined in the given variant.

`name` must reference a valid assembly variant that is contained in the current drawing.

This function returns a non-zero value if the given assembly variant exists, zero otherwise.

The assembly variant that has been set by a call to `setvariant()` is only active until the User Language Program returns. After that, the variant in the drawing will be the same as before the start of the ULP.

Example

```
if (setvariant("My variant")) {
   // do something ...
else
   // error: unknown variant
```

variant()

Function

Query the current assembly variant.

Syntax

```
string variant(void);
```

See also setvariant(), UL_VARIANTDEF, VARIANT command

The `variant()` function returns the name of the current assembly variant. If no variant is currently selected, the empty string (`''`) is returned.

Example

```
string CurrentVariant = variant();
```

XML Functions

XML functions are used to process XML (*Extensible Markup Language*) data.

The following XML functions are available:

- xmlattribute()
- xmlattributes()
- xmlelement()

- xmlelements()
- xmltags()
- xmltext()

xmlattribute(), xmlattributes()

Function
    Extract the attributes of an XML tag.
Syntax

```
    string xmlattribute(string xml, string tag, string attribute);
    int xmlattributes(string &array[], string xml, string tag);
```

See also xmlelement(), xmltags(), xmltext()

The `xmlattribute` function returns the value of the given `attribute` from the given `tag` within the given `xml` code. If an attribute appears more than once in the same tag, the value of its last occurrence is taken.

The `xmlattributes` function stores the names of all attributes from the given `tag` within the given `xml` code in the `array` and returns the number of attributes found. If an attribute appears more than once in the same tag, its name appears only once in the `array`.

The `tag` is given in the form of a *path*.

If the given `xml` code contains an error, the result of any XML function is empty, and a warning dialog is presented to the user, giving information about where in the ULP and XML code the error occurred. Note that the line and column number within the XML code refers to the actual string given to this function as the `xml` parameter.

Example

```
// XML contains the following data:
<root>
  <body abc="def" xyz="123">
    ...
  </body>
</root>
//
string s[];
int n = xmlattributes(s, XML, "root/body");
Result: { "abc", "xyz" }
string s = xmlattribute(XML, "root/body", "xyz");
Result: "123"
```

xmlelement(), xmlelements()

Function
    Extract elements from an XML code.
Syntax

```
    string xmlelement(string xml, string tag);
    int xmlelements(string &array[], string xml, string tag);
```

See also xmltags(), xmlattribute(), xmltext()

The `xmlelement` function returns the complete XML element of the given `tag` within the given `xml` code. The result still contains the element's outer XML tag, and can thus be used for further processing with the other XML functions. Any whitespace within plain text parts of the element is retained. The overall formatting of the XML tags within the element may be different than the original `xml` code, though.
If there is more than one occurrence of `tag` within `xml`, the first one will be returned. Use `xmlelements` if you want to get all occurrences.

The `xmlelements` function works just like `xmlelement`, but returns all occurrences of elements with the given `tag`. The return value is the number of elements stored in the `array`.

The `tag` is given in the form of a *path*.

If the given `xml` code contains an error, the result of any XML function is empty, and a warning dialog is presented to the user, giving information about where in the ULP and XML code the error occurred. Note that the line and column number within the XML code refers to the actual string given to this function as the `xml` parameter.

Example

```
// XML contains the following data:
<root>
  <body>
    <contents>
      <string>Some text 1</string>
      <any>anything 1</any>
    </contents>
    <contents>
      <string>Some text 2</string>
      <any>anything 2</any>
    </contents>
    <appendix>
      <string>Some text 3</string>
    </appendix>
  </body>
</root>
//
string s = xmlelement(XML, "root/body/appendix");
Result: " <appendix>\n  <string>Some text 3</string>\n </appendix>\n"
string s[];
int n = xmlelements(s, XML, "root/body/contents");
Result: { " <contents>\n  <string>Some text 1</string>\n  <any>anything
1</any>\n </contents>\n",
          " <contents>\n  <string>Some text 2</string>\n  <any>anything
2</any>\n </contents>\n"
        }
```

xmltags()

Function

Extract the list of tag names within an XML code.

Syntax

```
int xmltags(string &array[], string xml, string tag);
```

See also xmlelement(), xmlattribute(), xmltext()

The `xmltags` function returns the names of all the tags on the top level of the given `tag` within the given `xml` code. The return value is the number of tag names stored in the `array`.

Each tag name is returned only once, even if it appears several times in the XML code.

The `tag` is given in the form of a *path*.

If the given `xml` code contains an error, the result of any XML function is empty, and a warning dialog is presented to the user, giving information about where in the ULP and XML code the error occurred. Note that the line and column number within the XML code refers to the actual string given to this function as the `xml` parameter.

Example

```
// XML contains the following data:
<root>
  <body>
    <contents>
      <string>Some text 1</string>
      <any>anything 1</any>
    </contents>
    <contents>
      <string>Some text 2</string>
      <any>anything 2</any>
    </contents>
    <appendix>
      <string>Some text 3</string>
    </appendix>
  </body>
</root>
//
string s[];
int n = xmltags(s, XML, "root/body");
Result: { "contents", "appendix" }
int n = xmltags(s, XML, "");
Result: "root"
```

xmltext()

Function

Extract the textual data of an XML element.

Syntax

```
string xmltext(string xml, string tag);
```

See also xmlelement(), xmlattribute(), xmltags()

The `xmltext` function returns the textual data from the given `tag` within the given `xml` code.

Any tags within the text are stripped, whitespace (including newline characters) is retained.

The `tag` is given in the form of a *path*.

If the given `xml` code contains an error, the result of any XML function is empty, and a warning dialog is presented to the user, giving information about where in the ULP and

XML code the error occurred. Note that the line and column number within the XML code refers to the actual string given to this function as the `xml` parameter.

Example

```
// XML contains the following data:
<root>
  <body>
    Some <b>text</b>.
  </body>
</root>
//
string s = xmltext(XML, "root/body");
Result: "\n    Some text.\n  "
```

Builtin Statements

Builtin statements are generally used to open a certain context in which data structures or files can be accessed.

The general syntax of a builtin statement is

```
name(parameters) statement
```

where `name` is the name of the builtin statement, `parameters` stands for one or more parameters, and `statement` is the code that will be executed inside the context opened by the builtin statement.

Note that `statement` can be a compound statement, as in

```
board(B) {
  B.elements(E) printf("Element: %s\n", E.name);
  B.Signals(S)  printf("Signal: %s\n", S.name);
  }
```

The following builtin statements are available:

- board()
- deviceset()
- library()
- output()
- package()
- schematic()
- sheet()
- symbol()

board()

Function

Opens a board context.

Syntax

`board(identifier) statement`

See also schematic, library

The `board` statement opens a board context if the current editor window contains a board drawing. A variable of type UL_BOARD is created and is given the name indicated by `identifier`.

Once the board context is successfully opened and a board variable has been created, the `statement` is executed. Within the scope of the `statement` the board variable can be accessed to retrieve further data from the board.

If the current editor window does not contain a board drawing, an error message is given and the ULP is terminated.

Check if there is a board

By using the `board` statement without an argument you can check if the current editor window contains a board drawing. In that case, `board` behaves like an integer constant, returning `1` if there is a board drawing in the current editor window, and `0` otherwise.

Accessing board from a schematic

If the current editor window contains a schematic drawing, you can still access that schematic's board by preceding the `board` statement with the prefix `project`, as in

```
project.board(B) { ... }
```

This will open a board context regardless whether the current editor window contains a board or a schematic drawing. However, there must be an editor window containing that board somewhere on the desktop!

Example

```
if (board)
   board(B) {
     B.elements(E)
       printf("Element: %s\n", E.name);
     }
```

deviceset()

Function
: Opens a device set context.

Syntax
: `deviceset(identifier) statement`

See also package, symbol, library

The `deviceset` statement opens a device set context if the current editor window contains a device drawing. A variable of type UL_DEVICESET is created and is given the name indicated by `identifier`.

Once the device set context is successfully opened and a device set variable has been created, the `statement` is executed. Within the scope of the `statement` the device set variable can be accessed to retrieve further data from the device set.

If the current editor window does not contain a device drawing, an error message is given and the ULP is terminated.

Check if there is a device set

By using the `deviceset` statement without an argument you can check if the current editor window contains a device drawing. In that case, `deviceset` behaves like an integer constant, returning `1` if there is a device drawing in the current editor window, and `0` otherwise.

Example

```
if (deviceset)
   deviceset(D) {
     D.gates(G)
       printf("Gate: %s\n", G.name);
     }
```

library()

Function

Opens a library context.

Syntax

```
library(identifier) statement
```

See also board, schematic, deviceset, package, symbol

The `library` statement opens a library context if the current editor window contains a library drawing. A variable of type UL_LIBRARY is created and is given the name indicated by `identifier`.

Once the library context is successfully opened and a library variable has been created, the `statement` is executed. Within the scope of the `statement` the library variable can be accessed to retrieve further data from the library.

If the current editor window does not contain a library drawing, an error message is given and the ULP is terminated.

Check if there is a library

By using the `library` statement without an argument you can check if the current editor window contains a library drawing. In that case, `library` behaves like an integer constant, returning `1` if there is a library drawing in the current editor window, and `0` otherwise.

Example

```
if (library)
   library(L) {
     L.devices(D)
       printf("Device: %s\n", D.name);
     }
```

output()

Function

Opens an output file for subsequent printf() calls.

Syntax

```
output(string filename[, string mode]) statement
```

See also printf, fileerror

The `output` statement opens a file with the given `filename` and `mode` for output through subsequent printf() calls. If the file has been successfully opened, the `statement` is executed, and after that the file is closed.

If the file cannot be opened, an error message is given and execution of the ULP is terminated.

By default the output file is written into the **Project** directory.

File Modes

The `mode` parameter defines how the output file is to be opened. If no `mode` parameter is given, the default is `"wt"`.

`a`	append to an existing file, or create a new file if it does not exist
`w`	create a new file (overwriting an existing file)
`t`	open file in text mode
`b`	open file in binary mode
`D`	delete this file when ending the EAGLE session (only works together with `w`)
`F`	force using this file name (normally *.brd, *.sch and *.lbr are rejected)

Mode characters may appear in any order and combination. However, only the last one of `a` and `w` or `t` and `b`, respectively, is significant. For example a mode of `"abtw"` would open a file for textual write, which would be the same as `"wt"`.

Nested Output statements

`output` statements can be nested, as long as there are enough file handles available, and provided that no two active `output` statements access the **same** file.

Example

```
void PrintText(string s)
{
  printf("This also goes into the file: %s\n", s);
}
output("file.txt", "wt") {
  printf("Directly printed\n");
  PrintText("via function call");
  }
```

package()

Function

Opens a package context.

Syntax

`package(identifier) statement`

See also library, deviceset, symbol

The `package` statement opens a package context if the current editor window contains a package drawing. A variable of type UL_PACKAGE is created and is given the name indicated by `identifier`.

Once the package context is successfully opened and a package variable has been created, the `statement` is executed. Within the scope of the `statement` the package variable can be accessed to retrieve further data from the package.

If the current editor window does not contain a package drawing, an error message is given and the ULP is terminated.

Check if there is a package

By using the `package` statement without an argument you can check if the current editor window contains a package drawing. In that case, `package` behaves like an integer constant, returning `1` if there is a package drawing in the current editor window, and `0` otherwise.

Example

```
if (package)
   package(P) {
     P.contacts(C)
       printf("Contact: %s\n", C.name);
     }
```

schematic()

Function

Opens a schematic context.

Syntax

`schematic(identifier) statement`

See also board, library, sheet

The `schematic` statement opens a schematic context if the current editor window contains a schematic drawing. A variable of type UL_SCHEMATIC is created and is given the name indicated by `identifier`.

Once the schematic context is successfully opened and a schematic variable has been created, the `statement` is executed. Within the scope of the `statement` the schematic variable can be accessed to retrieve further data from the schematic.

If the current editor window does not contain a schematic drawing, an error message is given and the ULP is terminated.

Check if there is a schematic

By using the `schematic` statement without an argument you can check if the current editor window contains a schematic drawing. In that case, `schematic` behaves like an integer constant, returning `1` if there is a schematic drawing in the current editor window, and `0` otherwise.

Accessing schematic from a board

If the current editor window contains a board drawing, you can still access that board's schematic by preceding the `schematic` statement with the prefix `project`, as in

```
project.schematic(S) { ... }
```

This will open a schematic context regardless whether the current editor window contains a schematic or a board drawing. However, there must be an editor window containing that schematic somewhere on the desktop!

Access the current Sheet

Use the `sheet` statement to directly access the currently loaded sheet.

Example

```
if (schematic)
   schematic(S) {
     S.parts(P)
       printf("Part: %s\n", P.name);
     }
```

sheet()

Function

Opens a sheet context.

Syntax

```
sheet(identifier) statement
```

See also schematic

The `sheet` statement opens a sheet context if the current editor window contains a sheet drawing. A variable of type UL_SHEET is created and is given the name indicated by `identifier`.

Once the sheet context is successfully opened and a sheet variable has been created, the `statement` is executed. Within the scope of the `statement` the sheet variable can be accessed to retrieve further data from the sheet.

If the current editor window does not contain a sheet drawing, an error message is given and the ULP is terminated.

Check if there is a sheet

By using the `sheet` statement without an argument you can check if the current editor window contains a sheet drawing. In that case, `sheet` behaves like an integer constant, returning `1` if there is a sheet drawing in the current editor window, and `0` otherwise.

Example

```
if (sheet)
   sheet(S) {
     S.instances(I)
       printf("Instance: %s\n", I.name);
     }
```

symbol()

Function

Opens a symbol context.

Syntax

```
symbol(identifier) statement
```

See also library, deviceset, package

The `symbol` statement opens a symbol context if the current editor window contains a symbol drawing. A variable of type UL_SYMBOL is created and is given the name indicated by `identifier`.

Once the symbol context is successfully opened and a symbol variable has been created, the `statement` is executed. Within the scope of the `statement` the symbol variable can be accessed to retrieve further data from the symbol.

If the current editor window does not contain a symbol drawing, an error message is given and the ULP is terminated.

Check if there is a symbol

By using the `symbol` statement without an argument you can check if the current editor window contains a symbol drawing. In that case, `symbol` behaves like an integer constant, returning `1` if there is a symbol drawing in the current editor window, and `0` otherwise.

Example

```
if (symbol)
   symbol(S) {
     S.pins(P)
       printf("Pin: %s\n", P.name);
     }
```

Dialogs

User Language Dialogs allow you to define your own frontend to a User Language Program.

The following sections describe User Language Dialogs in detail:

Predefined Dialogs	describes the ready to use standard dialogs
Dialog Objects	defines the objects that can be used in a dialog
Layout Information	explains how to define the location of objects within a dialog
Dialog Functions	describes special functions for use with dialogs
A Complete Example	shows a complete ULP with a data entry dialog

Predefined Dialogs

Predefined Dialogs implement the typical standard dialogs that are frequently used for selecting file names or issuing error messages.

The following predefined dialogs are available:

- dlgDirectory()
- dlgFileOpen()
- dlgFileSave()
- dlgMessageBox()

See Dialog Objects for information on how to define your own complex user dialogs.

dlgDirectory()

Function

Displays a directory dialog.

Syntax

`string dlgDirectory(string Title[, string Start])`

Returns

The `dlgDirectory` function returns the full pathname of the selected directory.
If the user has canceled the dialog, the result will be an empty string.

See also dlgFileOpen

The `dlgDirectory` function displays a directory dialog from which the user can select a directory.

`Title` will be used as the dialog's title.

If `Start` is not empty, it will be used as the starting point for the `dlgDirectory`.

Example

```
string dirName;
dirName = dlgDirectory("Select a directory", "");
```

dlgFileOpen(), dlgFileSave()

Function

Displays a file dialog.

Syntax

`string dlgFileOpen(string Title[, string Start[, string`

```
    Filter]])
    string dlgFileSave(string Title[, string Start[, string
    Filter]])
```
Returns

The `dlgFileOpen` and `dlgFileSave` functions return the full pathname of the selected file.
If the user has canceled the dialog, the result will be an empty string.

See also dlgDirectory

The `dlgFileOpen` and `dlgFileSave` functions display a file dialog from which the user can select a file.

`Title` will be used as the dialog's title.

If `Start` is not empty, it will be used as the starting point for the file dialog. Otherwise the current directory will be used.

Only files matching `Filter` will be displayed. If `Filter` is empty, all files will be displayed.

`Filter` can be either a simple wildcard (as in `"*.brd"`), a list of wildcards (as in `"*.bmp *.jpg"`) or may even contain descriptive text, as in `"Bitmap files (*.bmp)"`. If the "File type" combo box of the file dialog shall contain several entries, they have to be separated by double semicolons, as in `"Bitmap files (*.bmp);;Other images (*.jpg *.png)"`.

Example

```
string fileName;
fileName = dlgFileOpen("Select a file", "", "*.brd");
```

dlgMessageBox()

Function

Displays a message box.

Syntax

`int dlgMessageBox(string Message[, *button_list*])`

Returns

The `dlgMessageBox` function returns the index of the button the user has selected.
The first button in `button_list` has index `0`.

See also status()

The `dlgMessageBox` function displays the given `Message` in a modal dialog and waits until the user selects one of the buttons defined in `button_list`.

If `Message` contains any HTML tags, the characters '<', '>' and '&' must be given as "&lt;", "&gt;" and "&amp;", respectively, if they shall be displayed as such.

`button_list` is an optional list of comma separated strings, which defines the set of buttons that will be displayed at the bottom of the message box.
A maximum of three buttons can be defined. If no `button_list` is given, it defaults to `"OK"`.

The first button in `button_list` will become the default button (which will be selected if the user hits ENTER), and the last button in the list will become the "cancel button", which is selected if the user hits ESCape or closes the message box. You can make a different button the default button by starting its name with a `'+'`, and you can make a different button the cancel button by starting its name with a `'-'`. To start a button text with an actual `'+'` or `'-'` it has to be escaped.

If a button text contains an `'&'`, the character following the ampersand will become a hotkey, and when the user hits the corresponding key, that button will be selected. To have an actual `'&'` character in the text it has to be escaped.

The message box can be given an icon by setting the first character of `Message` to
`';'` - for an *Information*
`'!'` - for a *Warning*
`':'` - for an *Error*
If, however, the `Message` shall begin with one of these characters, it has to be escaped.

On **Mac OS X** only the character `':'` will actually result in showing an icon. All others are ignored.

Example

```
if (dlgMessageBox("!Are you sure?", "&Yes", "&No") == 0) {
   // let's do it!
   }
```

Dialog Objects

A User Language Dialog is built from the following *Dialog Objects*:

dlgCell	a grid cell context
dlgCheckBox	a checkbox
dlgComboBox	a combo box selection field
dlgDialog	the basic container of any dialog
dlgGridLayout	a grid based layout context
dlgGroup	a group field
dlgHBoxLayout	a horizontal box layout context
dlgIntEdit	an integer entry field
dlgLabel	a text label
dlgListBox	a list box
dlgListView	a list view
dlgPushButton	a push button
dlgRadioButton	a radio button
dlgRealEdit	a real entry field
dlgSpacing	a layout spacing object
dlgSpinBox	a spin box selection field
dlgStretch	a layout stretch object
dlgStringEdit	a string entry field
dlgTabPage	a tab page
dlgTabWidget	a tab page container
dlgTextEdit	a text entry field
dlgTextView	a text viewer field

dlgVBoxLayout   a vertical box layout context

dlgCell

Function

Defines a cell location within a grid layout context.

Syntax

```
dlgCell(int row, int column[, int row2, int column2])
statement
```

See also dlgGridLayout, dlgHBoxLayout, dlgVBoxLayout, Layout Information, A Complete Example

The `dlgCell` statement defines the location of a cell within a grid layout context.

The row and column indexes start at 0, so the upper left cell has the index (0, 0).

With two parameters the dialog object defined by `statement` will be placed in the single cell addresses by `row` and `column`. With four parameters the dialog object will span over all cells from `row`/`column` to `row2`/`column2`.

By default a `dlgCell` contains a dlgHBoxLayout, so if the cell contains more than one dialog object, they will be placed next to each other horizontally.

Example

```
string Text;
dlgGridLayout {
  dlgCell(0, 0) dlgLabel("Cell 0,0");
  dlgCell(1, 2, 4, 7) dlgTextEdit(Text);
  }
```

dlgCheckBox

Function

Defines a checkbox.

Syntax

```
dlgCheckBox(string Text, int &Checked) [ statement ]
```

See also dlgRadioButton, dlgGroup, Layout Information, A Complete Example

The `dlgCheckBox` statement defines a check box with the given `Text`.

If `Text` contains an `'&'`, the character following the ampersand will become a hotkey, and when the user hits `Alt+hotkey`, the checkbox will be toggled. To have an actual `'&'` character in the text it has to be escaped.

`dlgCheckBox` is mainly used within a dlgGroup, but can also be used otherwise.
All check boxes within the same dialog must have **different** `Checked` variables!

If the user checks a `dlgCheckBox`, the associated `Checked` variable is set to `1`, otherwise it is set to `0`. The initial value of `Checked` defines whether a checkbox is initially checked. If `Checked` is not equal to `0`, the checkbox is initially checked.

The optional `statement` is executed every time the `dlgCheckBox` is toggled.

Example

```
int mirror = 0;
int rotate = 1;
int flip   = 0;
dlgGroup("Orientation") {
  dlgCheckBox("&Mirror", mirror);
  dlgCheckBox("&Rotate", rotate);
  dlgCheckBox("&Flip", flip);
  }
```

dlgComboBox

Function

Defines a combo box selection field.

Syntax

`dlgComboBox(string array[], int &Selected) [ statement ]`

See also dlgListBox, dlgLabel, Layout Information, A Complete Example

The `dlgComboBox` statement defines a combo box selection field with the contents of the given `array`.

`Selected` reflects the index of the selected combo box entry. The first entry has index `0`.

Each element of `array` defines the contents of one entry in the combo box. None of the strings in `array` may be empty (if there is an empty string, all strings after and including that one will be dropped).

The optional `statement` is executed whenever the selection in the `dlgComboBox` changes.
Before the `statement` is executed, all variables that have been used with dialog objects are updated to their current values, and any changes made to these variables inside the `statement` will be reflected in the dialog when the statement returns.

If the initial value of `Selected` is outside the range of the `array` indexes, it is set to `0`.

Example

```
string Colors[] = { "red", "green", "blue", "yellow" };
int Selected = 2; // initially selects "blue"
dlgComboBox(Colors, Selected) dlgMessageBox("You have selected " +
Colors[Selected]);
```

dlgDialog

Function

Executes a User Language Dialog.

Syntax

`int dlgDialog(string Title) *block* ;`

Returns

The `dlgDialog` function returns an integer value that can be given a user defined meaning through a call to the `dlgAccept()` function.
If the dialog is simply closed, the return value will be `-1`.

See also dlgGridLayout, dlgHBoxLayout, dlgVBoxLayout, dlgAccept, dlgReset, dlgReject, A Complete Example

The `dlgDialog` function executes the dialog defined by `block`. This is the only dialog object that actually is a User Language builtin function. Therefore it can be used anywhere where a function call is allowed.

The `block` normally contains only other dialog objects, but it is also possible to use other User Language statements, for example to conditionally add objects to the dialog (see the second example below).

By default a `dlgDialog` contains a dlgVBoxLayout, so a simple dialog doesn't have to worry about the layout.

A `dlgDialog` should at some point contain a call to the `dlgAccept()` function in order to allow the user to close the dialog and accept its contents.

If all you need is a simple message box or file dialog you might want to use one of the Predefined Dialogs instead.

Examples

```
int Result = dlgDialog("Hello") {
  dlgLabel("Hello world");
  dlgPushButton("+OK") dlgAccept();
  };
int haveButton = 1;
dlgDialog("Test") {
  dlgLabel("Start");
  if (haveButton)
     dlgPushButton("Here") dlgAccept();
  };
```

dlgGridLayout

Function

Opens a grid layout context.

Syntax

`dlgGridLayout` *`statement`*

See also dlgCell, dlgHBoxLayout, dlgVBoxLayout, Layout Information, A Complete Example

The `dlgGridLayout` statement opens a grid layout context.

The only dialog object that can be used directly in `statement` is dlgCell, which defines the location of a particular dialog object within the grid layout.

The row and column indexes start at 0, so the upper left cell has the index (0, 0).
The number of rows and columns is automatically extended according to the location of dialog objects that are defined within the grid layout context, so you don't have to explicitly define the number of rows and columns.

Example

```
dlgGridLayout {
  dlgCell(0, 0) dlgLabel("Row 0/Col 0");
  dlgCell(1, 0) dlgLabel("Row 1/Col 0");
  dlgCell(0, 1) dlgLabel("Row 0/Col 1");
  dlgCell(1, 1) dlgLabel("Row 1/Col 1");
  }
```

dlgGroup

Function

Defines a group field.

Syntax

`dlgGroup(string Title)` *`statement`*

See also dlgCheckBox, dlgRadioButton, Layout Information, A Complete Example

The `dlgGroup` statement defines a group with the given `Title`.

By default a `dlgGroup` contains a dlgVBoxLayout, so a simple group doesn't have to worry about the layout.

`dlgGroup` is mainly used to contain a set of radio buttons or check boxes, but may as well contain any other objects in its `statement`.
Radio buttons within a `dlgGroup` are numbered starting with `0`.

Example

```
int align = 1;
dlgGroup("Alignment") {
  dlgRadioButton("&Top", align);
  dlgRadioButton("&Center", align);
  dlgRadioButton("&Bottom", align);
  }
```

dlgHBoxLayout

Function

Opens a horizontal box layout context.

Syntax

`dlgHBoxLayout` *`statement`*

See also dlgGridLayout, dlgVBoxLayout, Layout Information, A Complete Example

The `dlgHBoxLayout` statement opens a horizontal box layout context for the given `statement`.

Example

```
dlgHBoxLayout {
  dlgLabel("Box 1");
  dlgLabel("Box 2");
  dlgLabel("Box 3");
  }
```

dlgIntEdit

Function

Defines an integer entry field.

Syntax

```
dlgIntEdit(int &Value, int Min, int Max)
```

See also dlgRealEdit, dlgStringEdit, dlgLabel, Layout Information, A Complete Example

The `dlgIntEdit` statement defines an integer entry field with the given `Value`.

If `Value` is initially outside the range defined by `Min` and `Max` it will be limited to these values.

Example

```
int Value = 42;
dlgHBoxLayout {
  dlgLabel("Enter a &Number between 0 and 99");
  dlgIntEdit(Value, 0, 99);
  }
```

dlgLabel

Function

Defines a text label.

Syntax

```
dlgLabel(string Text [, int Update])
```

See also Layout Information, A Complete Example, dlgRedisplay()

The `dlgLabel` statement defines a label with the given `Text`.

`Text` can be either a string literal, as in `"Hello"`, or a string variable.

If `Text` contains any HTML tags, the characters '<', '>' and '&' must be given as "&lt;", "&gt;" and "&amp;", respectively, if they shall be displayed as such.

External hyperlinks in the `Text` will be opened with the appropriate application program.

If the `Update` parameter is not `0` and `Text` is a string variable, its contents can be modified in the `statement` of, e.g., a dlgPushButton, and the label will be automatically updated. This, of course, is only useful if `Text` is a dedicated string variable (not, e.g., the loop variable of a `for` statement).

If `Text` contains an `'&'`, and the object following the label can have the keyboard focus, the character following the ampersand will become a hotkey, and when the user hits `Alt+hotkey`, the focus will go to the object that was defined immediately following the `dlgLabel`. To have an actual `'&'` character in the text it has to be escaped.

Example

```
string OS = "Windows";
dlgHBoxLayout {
  dlgLabel(OS, 1);
  dlgPushButton("&Change OS") { OS = "Linux"; }
  }
```

dlgListBox

Function

Defines a list box selection field.

Syntax

`dlgListBox(string array[], int &Selected) [ statement ]`

See also dlgComboBox, dlgListView, dlgSelectionChanged, dlgLabel, Layout Information, A Complete Example

The `dlgListBox` statement defines a list box selection field with the contents of the given `array`.

`Selected` reflects the index of the selected list box entry. The first entry has index `0`.

Each element of `array` defines the contents of one line in the list box. None of the strings in `array` may be empty (if there is an empty string, all strings after and including that one will be dropped).

The optional `statement` is executed whenever the user double clicks on an entry of the `dlgListBox` (see dlgSelectionChanged for information on how to have the `statement` called when only the selection in the list changes).
Before the `statement` is executed, all variables that have been used with dialog objects are updated to their current values, and any changes made to these variables inside the `statement` will be reflected in the dialog when the statement returns.

If the initial value of `Selected` is outside the range of the `array` indexes, no entry will be selected.

Example

```
string Colors[] = { "red", "green", "blue", "yellow" };
int Selected = 2; // initially selects "blue"
dlgListBox(Colors, Selected) dlgMessageBox("You have selected " +
Colors[Selected]);
```

dlgListView

Function

Defines a multi column list view selection field.

Syntax

```
dlgListView(string Headers, string array[], int &Selected[,
int &Sort]) [ statement ]
```

See also dlgListBox, dlgSelectionChanged, dlgLabel, Layout Information, A Complete

Example

The `dlgListView` statement defines a multi column list view selection field with the contents of the given `array`.

`Headers` is the tab separated list of column headers.

`Selected` reflects the index of the selected list view entry in the `array` (the sequence in which the entries are actually displayed may be different, because the contents of a `dlgListView` can be sorted by the various columns). The first entry has index `0`.
If no particular entry shall be initially selected, `Selected` should be initialized to `-1`. If it is set to `-2`, the first item according to the current sort column is made current.

`Sort` defines which column should be used to sort the list view. The leftmost column is numbered `1`. The sign of this parameter defines the direction in which to sort (positive values sort in ascending order). If `Sort` is `0` or outside the valid number of columns, no sorting will be done. The returned value of `Sort` reflects the column and sort mode selected by the user by clicking on the list column headers. By default `dlgListView` sorts by the first column, in ascending order.

Each element of `array` defines the contents of one line in the list view, and must contain tab separated values. If there are fewer values in an element of `array` than there are entries in the `Headers` string the remaining fields will be empty. If there are more values in an element of `array` than there are entries in the `Headers` string the superfluous elements will be silently dropped. None of the strings in `array` may be empty (if there is an empty string, all strings after and including that one will be dropped).

A list entry that contains line feeds (`'\n'`) will be displayed in several lines accordingly.

The optional `statement` is executed whenever the user double clicks on an entry of the `dlgListView` (see dlgSelectionChanged for information on how to have the `statement` called when only the selection in the list changes).
Before the `statement` is executed, all variables that have been used with dialog objects are updated to their current values, and any changes made to these variables inside the `statement` will be reflected in the dialog when the statement returns.

If the initial value of `Selected` is outside the range of the `array` indexes, no entry will be selected.

If `Headers` is an empty string, the first element of the `array` is used as the header string. Consequently the index of the first entry is then `1`.

The contents of a `dlgListView` can be sorted by any column by clicking on that column's header. Columns can also be swapped by "click&dragging" a column header. Note that none of these changes will have any effect on the contents of the `array`. If the contents shall be sorted alphanumerically a `numeric string[]` array can be used.

Example

```
string Colors[] = { "red\tThe color RED", "green\tThe color GREEN", "blue\tThe
color BLUE" };
int Selected = 0; // initially selects "red"
dlgListView("Name\tDescription", Colors, Selected) dlgMessageBox("You have
selected " + Colors[Selected]);
```

dlgPushButton

Function

Defines a push button.

Syntax

```
dlgPushButton(string Text) statement
```

See also Layout Information, Dialog Functions, A Complete Example

The `dlgPushButton` statement defines a push button with the given `Text`.

If `Text` contains an `'&'`, the character following the ampersand will become a hotkey, and when the user hits `Alt+hotkey`, the button will be selected. To have an actual `'&'` character in the text it has to be escaped.

If `Text` starts with a `'+'` character, this button will become the default button, which will be selected if the user hits ENTER.
If `Text` starts with a `'-'` character, this button will become the cancel button, which will be selected if the user closes the dialog.
CAUTION: Make sure that the `statement` of such a marked cancel button contains a call to dlgReject()! Otherwise the user may be unable to close the dialog at all!
To have an actual `'+'` or `'-'` character as the first character of the text it has to be escaped.

If the user selects a `dlgPushButton`, the given `statement` is executed.
Before the `statement` is executed, all variables that have been used with dialog objects are updated to their current values, and any changes made to these variables inside the `statement` will be reflected in the dialog when the statement returns.

Example

```
int defaultWidth = 10;
int defaultHeight = 20;
int width = 5;
int height = 7;
dlgPushButton("&Reset defaults") {
  width = defaultWidth;
  height = defaultHeight;
  }
dlgPushButton("+&Accept") dlgAccept();
dlgPushButton("-Cancel") { if (dlgMessageBox("Are you sure?", "Yes", "No") == 0)
dlgReject(); }
```

dlgRadioButton

Function

Defines a radio button.

Syntax

```
dlgRadioButton(string Text, int &Selected) [ statement ]
```

See also dlgCheckBox, dlgGroup, Layout Information, A Complete Example

The `dlgRadioButton` statement defines a radio button with the given `Text`.

If `Text` contains an `'&'`, the character following the ampersand will become a hotkey, and when the user hits `Alt+hotkey`, the button will be selected. To have an actual `'&'` character in the text it has to be escaped.

`dlgRadioButton` can only be used within a dlgGroup.
All radio buttons within the same group must use the **same** `Selected` variable!

If the user selects a `dlgRadioButton`, the index of that button within the `dlgGroup` is stored in the `Selected` variable.
The initial value of `Selected` defines which radio button is initially selected. If `Selected` is outside the valid range for this group, no radio button will be selected. In order to get the correct radio button selection, `Selected` must be set **before** the first `dlgRadioButton` is defined, and must not be modified between adding subsequent radio buttons. Otherwise it is undefined which (if any) radio button will be selected.

The optional `statement` is executed every time the `dlgRadioButton` is selected.

Example

```
int align = 1;
dlgGroup("Alignment") {
  dlgRadioButton("&Top", align);
  dlgRadioButton("&Center", align);
  dlgRadioButton("&Bottom", align);
  }
```

dlgRealEdit

Function
 Defines a real entry field.
Syntax
```
    dlgRealEdit(real &Value, real Min, real Max)
```

See also dlgIntEdit, dlgStringEdit, dlgLabel, Layout Information, A Complete Example

The `dlgRealEdit` statement defines a real entry field with the given `Value`.

If `Value` is initially outside the range defined by `Min` and `Max` it will be limited to these values.

Example

```
real Value = 1.4142;
dlgHBoxLayout {
  dlgLabel("Enter a &Number between 0 and 99");
  dlgRealEdit(Value, 0.0, 99.0);
  }
```

dlgSpacing

Function
 Defines additional space in a box layout context.
Syntax
```
    dlgSpacing(int Size)
```

See also dlgHBoxLayout, dlgVBoxLayout, dlgStretch, Layout Information, A Complete Example

The `dlgSpacing` statement defines additional space in a vertical or horizontal box layout context.

`Size` defines the number of pixels of the additional space.

Example

```
dlgVBoxLayout {
  dlgLabel("Label 1");
  dlgSpacing(40);
  dlgLabel("Label 2");
  }
```

dlgSpinBox

Function
: Defines a spin box selection field.

Syntax
: `dlgSpinBox(int &Value, int Min, int Max)`

See also dlgIntEdit, dlgLabel, Layout Information, A Complete Example

The `dlgSpinBox` statement defines a spin box entry field with the given `Value`.

If `Value` is initially outside the range defined by `Min` and `Max` it will be limited to these values.

Example

```
int Value = 42;
dlgHBoxLayout {
  dlgLabel("&Select value");
  dlgSpinBox(Value, 0, 99);
  }
```

dlgStretch

Function
: Defines an empty stretchable space in a box layout context.

Syntax
: `dlgStretch(int Factor)`

See also dlgHBoxLayout, dlgVBoxLayout, dlgSpacing, Layout Information, A Complete Example

The `dlgStretch` statement defines an empty stretchable space in a vertical or horizontal box layout context.

`Factor` defines the stretch factor of the space.

Example

```
dlgHBoxLayout {
  dlgStretch(1);
  dlgPushButton("+OK")    { dlgAccept(); };
  dlgPushButton("Cancel") { dlgReject(); };
  }
```

dlgStringEdit

Function

Defines a string entry field.

Syntax

`dlgStringEdit(string &Text[, string &History[][, int Size]])`

See also dlgRealEdit, dlgIntEdit, dlgTextEdit, dlgLabel, Layout Information, A Complete Example

The `dlgStringEdit` statement defines a one line text entry field with the given `Text`.

If `History` is given, the strings the user has entered over time are stored in that string array. The entry field then has a button that allows the user to select from previously entered strings. If a `Size` greater than zero is given, only at most that number of strings are stored in the array. If `History` contains data when the dialog is newly opened, that data will be used to initialize the history. The most recently entered user input is stored at index 0. None of the strings in `History` may be empty (if there is an empty string, all strings after and including that one will be dropped).

Example

```
string Name = "Linus";
dlgHBoxLayout {
  dlgLabel("Enter &Name");
  dlgStringEdit(Name);
  }
```

dlgTabPage

Function

Defines a tab page.

Syntax

`dlgTabPage(string Title)` *`statement`*

See also dlgTabWidget, Layout Information, A Complete Example

The `dlgTabPage` statement defines a tab page with the given `Title` containing the given `statement`.

If `Title` contains an `'&'`, the character following the ampersand will become a hotkey, and when the user hits `Alt+hotkey`, this tab page will be opened. To have an actual `'&'` character in the text it has to be escaped.

Tab pages can only be used within a dlgTabWidget.

By default a `dlgTabPage` contains a dlgVBoxLayout, so a simple tab page doesn't have to worry about the layout.

Example

```
dlgTabWidget {
  dlgTabPage("Tab &1") {
    dlgLabel("This is page 1");
    }
  dlgTabPage("Tab &2") {
    dlgLabel("This is page 2");
    }
  }
```

dlgTabWidget

Function

Defines a container for tab pages.

Syntax

`dlgTabWidget` *`statement`*

See also dlgTabPage, Layout Information, A Complete Example

The `dlgTabWidget` statement defines a container for a set of tab pages.

`statement` must be a sequence of one or more dlgTabPage objects. There must be no other dialog objects in this sequence.

Example

```
dlgTabWidget {
  dlgTabPage("Tab &1") {
    dlgLabel("This is page 1");
    }
  dlgTabPage("Tab &2") {
    dlgLabel("This is page 2");
    }
  }
```

dlgTextEdit

Function

Defines a multiline text entry field.

Syntax

`dlgTextEdit(string &Text)`

See also dlgStringEdit, dlgTextView, dlgLabel, Layout Information, A Complete Example

The `dlgTextEdit` statement defines a multiline text entry field with the given `Text`.

The lines in the `Text` have to be delimited by a newline character (`'\n'`). Any whitespace characters at the end of the lines contained in `Text` will be removed, and upon return there will be no whitespace characters at the end of the lines. Empty lines at the end of the text will be removed entirely.

Example

```
string Text = "This is some text.\nLine 2\nLine 3";
dlgVBoxLayout {
  dlgLabel("&Edit the text");
  dlgTextEdit(Text);
  }
```

dlgTextView

Function

Defines a multiline text viewer field.

Syntax

```
dlgTextView(string Text)
dlgTextView(string Text, string &Link) statement
```

See also dlgTextEdit, dlgLabel, Layout Information, A Complete Example

The `dlgTextView` statement defines a multiline text viewer field with the given `Text`.

The `Text` may contain HTML tags.

External hyperlinks in the `Text` will be opened with the appropriate application program.

If `Link` is given and the `Text` contains hyperlinks, `statement` will be executed every time the user clicks on a hyperlink, with the value of `Link` set to whatever the `<a href=...>` tag defines as the value of *href*. If, after the execution of `statement`, the `Link` variable is not empty, the default handling of hyperlinks will take place. This is also the case if `Link` contains some text before dlgTextView is opened, which allows for an initial scrolling to a given position. If a `Link` is given, external hyperlinks will not be opened.

Example

```
string Text = "This is some text.\nLine 2\nLine 3";
dlgVBoxLayout {
  dlgLabel("&View the text");
  dlgTextView(Text);
  }
```

dlgVBoxLayout

Function

Opens a vertical box layout context.

Syntax

```
dlgVBoxLayout statement
```

See also dlgGridLayout, dlgHBoxLayout, Layout Information, A Complete Example

The `dlgVBoxLayout` statement opens a vertical box layout context for the given `statement`.

By default a dlgDialog contains a `dlgVBoxLayout`, so a simple dialog doesn't have to worry about the layout.

Example

```
dlgVBoxLayout {
  dlgLabel("Box 1");
  dlgLabel("Box 2");
  dlgLabel("Box 3");
  }
```

Layout Information

All objects within a User Language Dialog a placed inside a *layout context*.

Layout contexts can be either grid, horizontal or vertical.

Grid Layout Context

Objects in a grid layout context must specify the grid coordinates of the cell or cells into which they shall be placed. To place a text label at row 5, column 2, you would write

```
dlgGridLayout {
  dlgCell(5, 2) dlgLabel("Text");
  }
```

If the object shall span over more than one cell you need to specify the coordinates of the starting cell and the ending cell. To place a group that extends from row 1, column 2 up to row 3, column 5, you would write

```
dlgGridLayout {
  dlgCell(1, 2, 3, 5) dlgGroup("Title") {
    //...
    }
  }
```

Horizontal Layout Context

Objects in a horizontal layout context are placed left to right.

The special objects dlgStretch and dlgSpacing can be used to further refine the distribution of the available space.

To define two buttons that are pushed all the way to the right edge of the dialog, you would write

```
dlgHBoxLayout {
  dlgStretch(1);
  dlgPushButton("+OK")    dlgAccept();
  dlgPushButton("Cancel") dlgReject();
  }
```

Vertical Layout Context

Objects in a vertical layout context follow the same rules as those in a horizontal layout context, except that they are placed top to bottom.

Mixing Layout Contexts

Vertical, horizontal and grid layout contexts can be mixed to create the desired layout structure of a dialog. See the Complete Example for a demonstration of this.

Dialog Functions

The following functions can be used with User Language Dialogs:

dlgAccept()	closes the dialog and accepts its contents
dlgRedisplay()	immediately redisplays the dialog after changes to any values
dlgReset()	resets all dialog objects to their initial values
dlgReject()	closes the dialog and rejects its contents
dlgSelectionChanged()	tells whether the current selection in a dlgListView or dlgListBox has changed

dlgAccept()

Function

Closes the dialog and accepts its contents.

Syntax

```
void dlgAccept([ int Result ]);
```

See also dlgReject, dlgDialog, A Complete Example

The `dlgAccept` function causes the dlgDialog to be closed and return after the current statement sequence has been completed.

Any changes the user has made to the dialog values will be accepted and are copied into the variables that have been given when the dialog objects were defined.

The optional `Result` is the value that will be returned by the dialog. Typically this should be a positive integer value. If no value is given, it defaults to `1`.

Note that `dlgAccept()` does return to the normal program execution, so in a sequence like

```
dlgPushButton("OK") {
  dlgAccept();
  dlgMessageBox("Accepting!");
  }
```

the statement after `dlgAccept()` will still be executed!

Example

```
int Result = dlgDialog("Test") {
               dlgPushButton("+OK")    dlgAccept(42);
               dlgPushButton("Cancel") dlgReject();
               };
```

dlgRedisplay()

Function

Redisplays the dialog after changing values.

Syntax

```
void dlgRedisplay(void);
```

See also dlgReset, dlgDialog, A Complete Example

The `dlgRedisplay` function can be called to immediately refresh the dlgDialog after changes have been made to the variables used when defining the dialog objects.

You only need to call `dlgRedisplay()` if you want the dialog to be refreshed while still executing program code. In the example below the status is changed to "Running..." and `dlgRedisplay()` has to be called to make this change take effect before the "program action" is performed. After the final status change to "Finished." there is no need to call `dlgRedisplay()`, since all dialog objects are automatically updated after leaving the statement.

Example

```
string Status = "Idle";
int Result = dlgDialog("Test") {
               dlgLabel(Status, 1); // note the '1' to tell the label to be
updated!
               dlgPushButton("+OK")    dlgAccept(42);
               dlgPushButton("Cancel") dlgReject();
               dlgPushButton("Run") {
                 Status = "Running...";
                 dlgRedisplay();
                 // some program action here...
                 Status = "Finished.";
                 }
               };
```

dlgReset()

Function

Resets all dialog objects to their initial values.

Syntax

```
void dlgReset(void);
```

See also dlgReject, dlgDialog, A Complete Example

The `dlgReset` function copies the initial values back into all dialog objects of the current dlgDialog.

Any changes the user has made to the dialog values will be discarded.

Calling `dlgReject()` implies a call to `dlgReset()`.

Example

```
int Number = 1;
int Result = dlgDialog("Test") {
               dlgIntEdit(Number);
               dlgPushButton("+OK")    dlgAccept(42);
               dlgPushButton("Cancel") dlgReject();
               dlgPushButton("Reset")  dlgReset();
               };
```

dlgReject()

Function

Closes the dialog and rejects its contents.

Syntax

```
void dlgReject([ int Result ]);
```

See also dlgAccept, dlgReset, dlgDialog, A Complete Example

The `dlgReject` function causes the dlgDialog to be closed and return after the current statement sequence has been completed.

Any changes the user has made to the dialog values will be discarded. The variables that have been given when the dialog objects were defined will be reset to their original values when the dialog returns.

The optional `Result` is the value that will be returned by the dialog. Typically this should be `0` or a negative integer value. If no value is given, it defaults to `0`.

Note that `dlgReject()` does return to the normal program execution, so in a sequence like

```
dlgPushButton("Cancel") {
  dlgReject();
  dlgMessageBox("Rejecting!");
  }
```

the statement after `dlgReject()` will still be executed!

Calling `dlgReject()` implies a call to `dlgReset()`.

Example

```
int Result = dlgDialog("Test") {
               dlgPushButton("+OK")    dlgAccept(42);
               dlgPushButton("Cancel") dlgReject();
               };
```

dlgSelectionChanged()

Function

Tells whether the current selection in a dlgListView or dlgListBox has changed.

Syntax

```
int dlgSelectionChanged(void);
```

Returns

The `dlgSelectionChanged` function returns a nonzero value if only the selection in the list has changed.

See also dlgListView, dlgListBox

The `dlgSelectionChanged` function can be used in a list context to determine whether the statement of the `dlgListView` or `dlgListBox` was called because the user double clicked on an item, or whether only the current selection in the list has changed.

If the statement of a `dlgListView` or `dlgListBox` doesn't contain any call to `dlgSelectionChanged`, that statement is only executed when the user double clicks on an item in the list. However, if a ULP needs to react on changes to the current selection in the list, it can call `dlgSelectionChanged` within the list's statement. This causes the statement to also be called if the current selection in the list changes.

If a list item is initially selected when the dialog is opened and the list's statement contains a call to `dlgSelectionChanged`, the statement is executed with `dlgSelectionChanged` returning true in order to indicate the initial change from "no selection" to an actual selection. Any later programmatical changes to the strings or the selection of the list will not trigger an automatic execution of the list's statement. This is important to remember in case the current list item controls another dialog object, for instance a `dlgTextView` that shows an extended representation of the currently selected item.

Example

```
string Colors[] = { "red\tThe color RED", "green\tThe color GREEN", "blue\tThe
color BLUE" };
int Selected = 0; // initially selects "red"
string MyColor;
dlgLabel(MyColor, 1);
dlgListView("Name\tDescription", Colors, Selected) {
  if (dlgSelectionChanged())
     MyColor = Colors[Selected];
  else
     dlgMessageBox("You have chosen " + Colors[Selected]);
  }
```

Escape Character

Some characters have special meanings in button or label texts, so they need to be *escaped* if they shall appear literally.

To do this you need to prepend the character with a *backslash*, as in

```
dlgLabel("Miller \\& Co.");
```

This will result in "Miller & Co." displayed in the dialog.

Note that there are actually **two** backslash characters here, since this line will first go through the User Language parser, which will strip the first backslash.

A Complete Example

Here's a complete example of a User Language Dialog.

```
int hor = 1;
int ver = 1;
string fileName;
int Result = dlgDialog("Enter Parameters") {
  dlgHBoxLayout {
    dlgStretch(1);
    dlgLabel("This is a simple dialog");
    dlgStretch(1);
    }
  dlgHBoxLayout {
    dlgGroup("Horizontal") {
      dlgRadioButton("&Top", hor);
      dlgRadioButton("&Center", hor);
      dlgRadioButton("&Bottom", hor);
      }
    dlgGroup("Vertical") {
      dlgRadioButton("&Left", ver);
      dlgRadioButton("C&enter", ver);
      dlgRadioButton("&Right", ver);
      }
    }
  dlgHBoxLayout {
    dlgLabel("File &name:");
    dlgStringEdit(fileName);
    dlgPushButton("Bro&wse") {
      fileName = dlgFileOpen("Select a file", fileName);
      }
    }
  dlgGridLayout {
    dlgCell(0, 0) dlgLabel("Row 0/Col 0");
    dlgCell(1, 0) dlgLabel("Row 1/Col 0");
    dlgCell(0, 1) dlgLabel("Row 0/Col 1");
    dlgCell(1, 1) dlgLabel("Row 1/Col 1");
    }
  dlgSpacing(10);
  dlgHBoxLayout {
    dlgStretch(1);
    dlgPushButton("+OK")    dlgAccept();
    dlgPushButton("Cancel") dlgReject();
    }
  };
```

Supported HTML tags

EAGLE supports a subset of the tags used to format HTML pages. This can be used to format the text of several User Language Dialog objects, in the `#usage` directive or in the description of library objects.

Text is considered to be HTML if the first line contains a tag. If this is not the case, and you want the text to be formatted, you need to enclose the entire text in the `<html>...</html>` tag.

The following table lists all supported HTML tags and their available attributes:

Tag	**Description**
<html>...</html>	An HTML document.
<html>...</html>	The body of an HTML document. It understands the following attribute • `bgcolor` - The background color, for example `bgcolor="yellow"` or `bgcolor="#0000FF"`. This attribute works only within a dlgTextView.
<h1>...</h1>	A top-level heading.
<h2>...</h2>	A sub-level heading.

<h3>...</h3>	A sub-sub-level heading.
<p>...</p>	A left-aligned paragraph. Adjust the alignment with the `align` attribute. Possible values are `left`, `right` and `center`.
<center>...</center>	A centered paragraph.
<blockquote>...</blockquote>	An indented paragraph, useful for quotes.
<ul>...</ul>	An un-ordered list. You can also pass a type argument to define the bullet style. The default is `type=disc`, other types are `circle` and `square`.
<ol>...</ol>	An ordered list. You can also pass a type argument to define the enumeration label style. The default is `type="1"`, other types are `"a"` and `"A"`.
<li>...</li>	A list item. This tag can only be used within the context of `ol` or `ul`.
<pre>...</pre>	For larger chunks of code. Whitespaces in the contents are preserved. For small bits of code, use the inline-style `code`.
<a>...</a>	An anchor or link. It understands the following attributes: • `href` - The reference target as in `<a href="target.html">...</a>`. You can also specify an additional anchor within the specified target document, for example `<a href="target.html#123">...</a>`. If you want to link to a local file that has a blank in its name, you need to prepend the file name with `file:`, as in `<a href="file:/path with blanks/target.html">...</a>`. • `name` - The anchor name, as in `<a name="123">...</a>`.
<em>...</em>	Emphasized (same as `<i>...</i>`).
<strong>...</strong>	Strong (same as `<b>...</b>`).
<i>...</i>	Italic font style.
<b>...</b>	Bold font style.
<u>...</u>	Underlined font style.
<big>...</big>	A larger font size.
<small>...</small>	A smaller font size.
<code>...</code>	Indicates Code. (same as `<tt>...</tt>`. For larger chunks of code, use the block-tag `pre`.
<tt>...</tt>	Typewriter font style.
<font>...</font>	Customizes the font size, family and text color. The tag understands the following attributes: • `color` - The text color, for example `color="red"` or `color="#FF0000"`. • `size` - The logical size of the font. Logical sizes 1 to 7 are supported. The value may either be absolute, for example `size=3,` or relative like `size=-2`. In the latter case, the sizes are simply added. • `face` - The family of the font, for example `face=times`.
<img...>	An image. This tag understands the following attributes: • `src` - The image name, for example `<img src="image.png">`. The URL of the image may be external, as in `<img src="http://www.cadsoft.de/cslogo.gif">`.

	• `width` - The width of the image. If the image does not fit to the specified size, it will be scaled automatically. • `height` - The height of the image. • `align` - Determines where the image is placed. Per default, an image is placed inline, just like a normal character. Specify `left` or `right` to place the image at the respective side.
<hr>	A horizonal line.

	A line break.
<nobr>...</nobr>	No break. Prevents word wrap.
<table>...</table>	A table definition. The default table is frameless. Specify the boolean attribute `border` in order to get a frame. Other attributes are: • `bgcolor` - The background color. • `width` - The table width. This is either absolute in pixels or relative in percent of the column width, for example `width=80%`. • `border` - The width of the table border. The default is 0 (= no border). • `cellspacing` - Additional space around the table cells. The default is 2. • `cellpadding` - Additional space around the contents of table cells. Default is 1.
<tr>...</tr>	A table row. Can only be used within `table`. Understands the attribute • `bgcolor` - The background color.
<td>...</td>	A table data cell. Can only be used within `tr.` Understands the attributes • `bgcolor` - The background color. • `width` - The cell width. This is either absolute in pixels or relative in percent of the entire table width, for example `width=50%`. • `colspan` - Defines how many columns this cell spans. The default is 1. • `rowspan` - Defines how many rows this cell spans. The default is 1. • `align` - Alignment, possible values are `left`, `right` and `center`. The default is left-aligned.
<th>...</th>	A table header cell. Like `td` but defaults to center-alignment and a bold font.
<author>...</author>	Marks the author of this text.
<dl>...</dl>	A definition list.
<dt>...</dt>	A definition tag. Can only be used within `dl`.
<dd>...</dd>	Definition data. Can only be used within `dl`.

Tag	**Meaning**
&lt;	<
&gt;	>
&amp;	&amp;
&nbsp;	non-breaking space
&auml;	ä
&ouml;	ö
&uuml;	ü
&Auml;	Ä

&Ouml;	Ö
&Uuml;	Ü
&szlig;	ß
&copy;	©
&deg;	°
&micro;	µ
&plusmn;	±
&quot;	"

Automatic Backup

Maximum backup level

The WRITE command creates backup copies of the saved files. These backups have the same name as the original file, with a modified extension that follows the pattern

```
.x#n
```

In this pattern `'x'` is replaced by the character

`'b'` for board files
`'s'` for schematic files
`'l'` for library files

`'n'` stands for a single digit number in the range 1..9. Higher numbers indicate older files.

The fixed '#' character makes it easy to delete all backup files from the operating system, using **`*.?#?`** as a wildcard.

Note that backup files with the same number 'n' do not necessarily represent consistent combinations of board and schematic files!

The maximum number of backup copies can be set in the backup dialog.

Auto backup interval

If a drawing has been modified a safety backup copy will be automatically created after at most the given *Auto backup interval*.

This safety backup file will have a name that follows the pattern

```
.x##
```

In this pattern `'x'` is replaced by the character

`'b'` for board files
`'s'` for schematic files
`'l'` for library files

The safety backup file will be deleted after a successful regular save operation. If the drawing has not been saved with the WRITE command (e.g. due to a power failure) this file can be renamed and loaded as a normal board, schematic or library file, repectively.

The auto backup interval can be set in the backup dialog.

Forward&Back Annotation

A schematic and board file are logically interconnected through automatic Forward&Back Annotation. Normally there are no special things to be considered about Forward&Back Annotation. This section, however, lists all of the details about what exactly happens during f/b activities:

- When adding a new part to a schematic, the part's package is added to the board at the lower left corner of the drawing. If the part contains power pins (pins with Direction "Pwr") the related pads will be automatically connected to their power signals.
- When deleting a part from a schematic drawing, the part's package is deleted from the board. Any wires that were connected to that package are left unchanged. This may require additional vias to be set in order to keep signals connected. These vias will **not** be set automatically! The ratsnest will be re-calculated for those signals that were connected to the removed package.
- When deleting a part from a board drawing, all of the gates contained in that part will be deleted from the schematic. Note that this may affect more than one sheet, if the gates were placed on different sheets!
- After an operation that removes a pad from a signal (or adds it to a signal) that has a polygon, the display of the connections to that polygon may be incorrect. In such a case the RATSNEST command will recalculate the polygon to show the correct Thermal/Annulus symbols. The same applies to Undo/Redo operations that involve pads connected to signals with polygons.
- A PinSwap or GateSwap operation in the schematic will make all the necessary changes to the wires of the board. However, after this operation the wires may overlap or violate minimum distance rules. Therefore the user should take a look at these wires and modify them with Move, Split, Change Layer etc.
- To make absolutely sure that a board and schematic belong to each other (and are therefore connected via Forward&Back Annotation) the two files must have the same file name (with extensions .brd and .sch) and must be located in the same directory!
- The Replace command checks whether all pads in the old package which have been assigned to pins will also be present in the new package, regardless whether they are connected to a signal or not.
- When the pins of two parts in the schematic are directly overlapping (and thus connected without a visible net wire), a net wire will be generated when these parts are moved away from each other. This is done to avoid unnecessary ripup of signal wires in the board.

Consistency Check

In order to use Forward&Back Annotation a board and schematic must be consistent, which means they must contain an equivalent set of parts/elements and nets/signals.

Normally a board and schematic will always be consistent as long as they have never been edited separately (in which case the message *"No Forward&Back Annotation will be performed!"* will have warned you).

When loading a pair of board and schematic files the program will check some consistency markers in the data files to see if these two files are still consistent. If these markers indicate an inconsistency, you will be offered to run an Electrical Rule Check (ERC), which will do a detailed cross-check on both files.

If this check turns out positive, the two files are marked as consistent and Forward&Back Annotation will be activated.

If the two files are found to be inconsistent the ERC protocol file will be brought up in a dialog and Forward&Back Annotation will **not** be activated.

Please do not be alarmed if you get a lot of inconsistency messages. In most cases fixing one error (like renaming a part or a net) will considerably reduce the number of error messages you get in the next ERC run.

Making a Board and Schematic consistent

To make an inconsistent pair of board and schematic files consistent, you have to manually fix any inconsistency listed in the ERC protocol. This can be done by applying editor commands like NAME, VALUE, PINSWAP, REPLACE etc. After fixing the inconsistencies you must use the ERC command again to check the files and eventually activate Forward&Back Annotation.

Limitations

The following actions are not allowed in a board when Back Annotation is active (i.e. the schematic is loaded, too):

- adding or copying a part that contains Pads or Smds
- deleting an airwire
- defining connections with the Signal command
- pasting from a board into a board, if the pasted objects contain parts with Pads or Smds, or Signals with connections

If you try to do one of the above things, you will receive a message telling you that this operation cannot be backannotated. In such a case please do the necessary operations in the schematic (they will then be forward annotated to the board). If you absolutely have to do it in the board, you can close the schematic window and then do anything you like inside the board. In that case, however, board and schematic will not be consistent any more!

Technical Support

As a registered EAGLE user you get free technical support from CadSoft. There are several ways to contact us or obtain the latest part libraries, drivers or program versions:

CadSoft Computer
19620 Pines Blvd. Suite 217
Pembroke Pines, FL 33029
USA

Phone	954-237-0932
Fax	954-237-0968
Email	support@cadsoftusa.com

URL www.cadsoftusa.com

License

To legally use EAGLE you need a registered user license. Please check whether the dialog "Help/About EAGLE" contains your name and address under "Registered to:". If you have any doubts about the validity or authenticity of your license, please contact our Technical Support staff for verification.

Under **Mac OS X** you can find this information under "EAGLE/About EAGLE".

There are different types of licenses, varying in the number of users who may use the program and in the areas of application the program may be used in:

Single-User License

Only **one** user may use the program at any given time. However, that user may install the program on any of his computers, as long as he makes sure that the program will only be used on one of these computers at a time.

A typical application of this kind would be a user who has a PC at home and also a notebook or laptop computer which he uses "on the road". As he would only use one of these computers at a time it is ok to have EAGLE installed on both of them.

Multi-user License

A multi-user license may be used by several users (up to the maximum number listed on the license) simultaneously. The program may be installed on any number of different computers at the location of the license holder.

Commercial License

The program may be used for any purpose, be it commercial or private.

Educational License

The program may only be used in an educational environment like a school, university or training workshop, in order to teach how to use ECAD software.

Student License

The program may only be used for private ("non-profit") purposes. Student versions are sold at a very low price, to allow people who could otherwise never afford buying EAGLE the use of the program for their private hobby or education. It is a violation of the license terms if you "earn money" by using a Student Licence of EAGLE.

EAGLE License

Before you can work with EAGLE it is necessary to register the program with your personalized license data.

In the dialog "EAGLE License" enter the name of your EAGLE license file, as well as the corresponding Installation Code you have received together with your license file (this code consists of 10 lowercase characters).

After pressing enter or clicking on the **OK** button, EAGLE will be installed with your personalized license data.

If you have problems installing EAGLE or are in doubt about the validity of your license please contact our Technical Support staff for assistance.

Installing additional modules

If you decided to update your license with the schematic/autorouter module you get a new license file with a new Installation Code. To make the new modules available you have to register your EAGLE again. Start the EAGLE program and choose in the Control Panel in the Help menu the item *EAGLE License*.

EAGLE Editions

EAGLE is available in three different editions to fit various user requirements.

Professional

The *Professional* edition provides full functionality:

- board area up to 1600x1600mm (64x64inch)
- up to 16 routing layers
- up to 999 sheets per schematic

Standard

The *Standard* edition has the following limitations:

- board area limited to 160x100mm (6.3x4inch), which corresponds to a full Eurocard
- only six routing layers (Top, Route2, Route3, Route14, Route15 and Bottom)
- a schematic can consist of up to 99 separate sheets

Light

The *Light* edition has the following limitations:

- board area limited to 100x80mm (4x3.2inch), which corresponds to half of a Eurocard
- only two routing layers (Top and Bottom)
- a schematic can consist of only one single sheet

Freemium

The *Freemium* edition is a *Free Premium* edition, which has capabilities between the *Light* and the *Standard* editions. The Freemium edition is available only after registration on http://www.element-14.com/eagle-freemium and has the following limitations:

- board area limited to 100x80mm (4x3.2inch), which corresponds to half of a Eurocard
- only four routing layers (Top, Route2, Route15 and Bottom)
- a schematic can consist of only four sheets
- a Freemium license is limited to one single user and computer, and requires an active connection to the Internet in order to work; it expires 60 days after installation

If you receive an error message like

The Light edition of EAGLE can't perform the requested action!

this means that you are attempting to do something that would violate the limitations that apply to the EAGLE edition in use, like for example placing an element outside of the allowed area.

All editions of EAGLE can be used to view files created with the *Standard* or *Professional* edition, even if these drawings exceed the editing capabilities of the edition currently in use.

To check which edition your license has enabled, select *Help/About EAGLE* from the Control Panel's menu.