Contents

General Help 13

Installation 14

Quick Introduction 16

Control Panel and Editor Windows 16

Entering Parameters and Values 16

Drawing a Schematic 17

Checking the Schematic 17

Generating a Board from a Schematic 18

Checking the Layout 18

Creating a Library Device 19

Control Panel 20

Context Menus 22

Directories 24

Backup 24

User Interface 25

Keyboard and Mouse 26

Selecting objects in dense areas 27

Editor Windows 28

Library Editor 28

Edit Library Object 28

Board Editor 28

Schematic Editor 29

Text Editor 29

Editor Commands 30

Command Syntax 32

ADD 34

ARC 37

ASSIGN 38
AUTO 40
BOARD 42
BUS 43
CHANGE 44
CIRCLE 45
CLASS 46
CLOSE 47
CONNECT 48
COPY 50
CUT 51
DELETE 52
DESCRIPTION 54
DISPLAY 55
DRC 57
EDIT 58
ERC 59
ERRORS 60
EXPORT 61
GATESWAP 63
GRID 64
GROUP 66
HELP 67
HOLE 68
INFO 69
INVOKE 70
JUNCTION 71
LABEL 72

LAYER 73

MARK 76

MENU 77

MIRROR 79

MOVE 80

NAME 81

NET 82

OPEN 83

OPTIMIZE 84

PACKAGE 85

PAD 86

PASTE 88

PIN 89

PINSWAP 92

POLYGON 93

PREFIX 95

PRINT 96

QUIT 97

RATSNEST 98

RECT 99

REDO 100

REMOVE 101

RENAME 102

REPLACE 103

RIPUP 104

ROTATE 105

ROUTE 106

RUN 107

SCRIPT **108**

SET **109**

SHOW **112**

SIGNAL **113**

SMASH **114**

SMD **115**

SPLIT **116**

TECHNOLOGY **117**

TEXT **118**

UNDO **120**

UPDATE **121**

USE **122**

VALUE **123**

VIA **124**

WINDOW **125**

WIRE **127**

WRITE **129**

Generating Output **130**

Printing **130**

Printing a Drawing **131**

Printing a Text **132**

Printer Page Setup **133**

CAM Processor **134**

Main CAM Menu **135**

CAM Processor Job **136**

Output Device **137**

Device Parameters **138**

Aperture Wheel File **139**

Aperture Emulation **139**

Aperture Tolerances **139**

Drill Rack File **139**

Drill Tolerances **139**

Offset **141**

Printable Area **141**

Pen Data **141**

Defining Your Own Device Driver **141**

Output File **142**

Flag Options **143**

Layers and Colors **144**

Outlines data **145**

Autorouter **146**

Design Checks **147**

Design Rules **148**

User Language **150**

Writing a ULP **151**

Executing a ULP **151**

Syntax **152**

Whitespace **153**

Comments **153**

Directives **154**

#INCLUDE 154

#usage **154**

Keywords **155**

Identifiers **156**

Constants **157**

Character Constants **157**

Integer Constants **157**

Real Constants **158**

String Constants **158**

Escape Sequences **159**

Punctuators **160**

Brackets **160**

Parentheses **160**

Braces **160**

Comma **162**

Semicolon **162**

Colon **162**

Equal Sign **162**

Data Types **163**

char **163**

int **163**

real **163**

string **164**

Type Conversions **165**

Typecast **165**

Object Types **166**

UL_ARC **169**

UL_AREA **170**

UL_BOARD **171**

UL_BUS **172**

UL_CIRCLE **173**

UL_CLASS **174**

UL_CONTACT **175**

UL_CONTACTREF **176**

UL_DEVICE	**177**
UL_DEVICESET	**178**
UL_ELEMENT	**179**
UL_GATE	**180**
UL_GRID	**181**
UL_HOLE	**182**
UL_INSTANCE	**183**
UL_JUNCTION	**184**
UL_LAYER	**185**
UL_LIBRARY	**187**
UL_NET	**188**
UL_PACKAGE	**189**
UL_PAD	**190**
UL_PART	**191**
UL_PIN	**192**
UL_PINREF	**194**
UL_POLYGON	**195**
UL_RECTANGLE	**197**
UL_SCHEMATIC	**198**
UL_SEGMENT	**199**
UL_SHEET	**200**
UL_SIGNAL	**201**
UL_SMD	**202**
UL_SYMBOL	**203**
UL_TEXT	**204**
UL_VIA	**205**
UL_WIRE	**206**
Definitions	**207**

Constant Definitions	**208**
Variable Definitions	**209**
Function Definitions	**210**
Operators	**211**
Bitwise Operators	**212**
Logical Operators	**212**
Comparison Operators	**212**
Evaluation Operators	**213**
Arithmetic Operators	**213**
String Operators	**214**
Expressions	**215**
Arithmetic Expression	**215**
Assignment Expression	**215**
String Expression	**215**
Comma Expression	**216**
Conditional Expression	**216**
Function Call	**217**
Statements	**218**
Compound Statement	**218**
Expression Statement	**218**
Control Statements	**218**
break	**220**
continue	**220**
do...while	**221**
for	**221**
if...else	**222**
return	**222**
switch	**223**

while **224**

Builtins **225**

Builtin Constants **225**

Builtin Variables **225**

Builtin Functions **226**

Character Functions **228**

is...() **229**

to...() **230**

File Handling Functions **231**

fileerror() **232**

fileglob() **233**

Filename Functions **234**

Filedata Functions **235**

File Input Functions **236**

fileread() **237**

Mathematical Functions **238**

Absolute, Maximum and Minimum Functions **239**

Rounding Functions **240**

Trigonometric Functions **241**

Exponential Functions **242**

Miscellaneous Functions **243**

exit() **244**

lookup() **245**

sort() **247**

Unit Conversions **248**

Printing Functions **249**

printf() **250**

sprintf() **253**

String Functions **254**
strchr() **255**
strjoin() **255**
strlen() **256**
strlwr() **256**
strrchr() **257**
strrstr() **257**
strsplit() **258**
strstr() **258**
strsub() **259**
strtod() **259**
strtol() **260**
strupr() **260**
Time Functions **261**
time() **261**
Time Conversions **262**
Builtin Statements **263**
board() **264**
deviceset() **265**
library() **266**
output() **267**
package() **268**
schematic() **269**
sheet() **270**
symbol() **271**
Dialogs **272**
Predefined Dialogs **272**
dlgDirectory() **273**

dlgFileOpen(), dlgFileSave() **273**

dlgMessageBox() **274**

Dialog Objects **275**

dlgCell **276**

dlgCheckBox **277**

dlgComboBox **278**

dlgDialog **279**

dlgGridLayout **280**

dlgGroup **281**

dlgHBoxLayout **282**

dlgIntEdit **283**

dlgLabel **284**

dlgListBox **285**

dlgListView **286**

dlgPushButton **287**

dlgRadioButton **288**

dlgRealEdit **289**

dlgSpacing **290**

dlgSpinBox **291**

dlgStretch **292**

dlgStringEdit **293**

dlgTabPage **294**

dlgTabWidget **295**

dlgTextEdit **296**

dlgTextView **297**

dlgVBoxLayout **298**

Layout Information **299**

Dialog Functions **300**

dlgAccept() **301**

dlgRedisplay() **302**

dlgReset() **303**

dlgReject() **304**

Escape Character **305**

A Complete Example **306**

Rich Text **307**

Automatic Backup **309**

Forward&Back Annotation **310**

Consistency Check **311**

Limitations **313**

Technical Support **314**

License **315**

Product Registration **316**

EAGLE Editions **316**

General Help

Explanation of the EAGLE Help Function

While inside a board »Page 28, schematic »Page 29, or library »Page 28 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 »Page 16
- Installation »Page 14
- Control Panel »Page 20

Installation

Global EAGLE parameters can be adjusted in the Control Panel »Page 20.

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 »Page 108 file.

User Interface

Command menu	MENU »Page 77 command..;
Assign keys	ASSIGN »Page 38 function_key command..;
Snap function	SET »Page 109 SNAP_LENGTH number;
	SET »Page 109 SNAP_BENDED ON | OFF;
	SET »Page 109 SELECT_FACTOR value;
Content of menus	SET »Page 109 USED_LAYERS name | number;
	SET »Page 109 WIDTH_MENU value..;
	SET »Page 109 DIAMETER_MENU value..;
	SET »Page 109 DRILL_MENU value..;
	SET »Page 109 SMD_MENU value..;
	SET »Page 109 SIZE_MENU value..;
Wire bend	SET »Page 109 WIRE_BEND bend_nr;
Beep on/off	SET »Page 109 BEEP ON | OFF;

Screen Display

Color for grid lines	SET »Page 109 COLOR_GRID color;
Color for layer	SET »Page 109 COLOR_LAYER layer color;
Fill style for layer	SET »Page 109 FILL_LAYER layer fill;
Grid parameter	SET »Page 109 GRID_REDRAW ON | OFF;
	SET »Page 109 MIN_GRID_SIZE pixels;
Min. text size displayed	SET »Page 109 MIN_TEXT_SIZE size;
Display of net lines	SET »Page 109 NET_WIRE_WIDTH width;
Display of pads	SET »Page 109 DISPLAY_MODE REAL | NODRILL;
	SET »Page 109 PAD_NAMES ON | OFF;
Display of bus lines	SET »Page 109 BUS_WIRE_WIDTH width;
DRC fill style	SET »Page 109 DRC_FILL fill_name;
Polygon processing	SET »Page 109 POLYGON_RATSNEST ON | OFF;
Vector font	SET »Page 109 VECTOR_FONT ON | OFF;

Mode Parameters

Package check	SET »Page 109 CHECK_CONNECTS ON | OFF;
Grid parameters	GRID »Page 64 options;
Replace mode	SET »Page 109 REPLACE_SAME NAMES | COORDS;
UNDO Buffer	SET »Page 109 UNDO_LOG ON | OFF;
Wire Optimizing	SET »Page 109 OPTIMIZING ON | OFF;
Net wire termination	SET »Page 109 AUTO_END_NET ON | OFF;
Automatic junctions	SET »Page 109 AUTO_JUNCTION ON | OFF;

Presettings

Pad shape	CHANGE »Page 44 SHAPE shape;
Wire width	CHANGE »Page 44 WIDTH value;
Pad/via diameter	CHANGE »Page 44 DIAMETER diameter;
Pad/via/hole drill diam.	CHANGE »Page 44 DRILL value;
Smd size	CHANGE »Page 44 SMD width height;
Text height	CHANGE »Page 44 SIZE value;
Text line width	CHANGE »Page 44 RATIO ratio;
Text font	CHANGE »Page 44 FONT font;
Polygon parameter	CHANGE »Page 44 THERMALS ON | OFF;
Polygon parameter	CHANGE »Page 44 ORPHANS ON | OFF;
Polygon parameter	CHANGE »Page 44 ISOLATE distance;

Polygon parameter	CHANGE »Page 44 POUR SOLID | HATCH;
Polygon parameter	CHANGE »Page 44 RANK value;
Polygon parameter	CHANGE »Page 44 SPACING distance;

Quick Introduction

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

- Control Panel and Editor Windows »Page 16
- Using Editor Commands »Page 32
- Entering Parameters and Values »Page 16
- Drawing a Schematic »Page 17
- Checking the Schematic »Page 17
- Generating a Board from a Schematic »Page 18
- Checking the Layout »Page 18
- Creating a Library Device »Page 19
- Using the Autorouter »Page 146
- Using the System Printer »Page 130
- Using the CAM Processor »Page 134

In case of problems please contact our free Technical Support »Page 314.

Control Panel and Editor Windows

From the Control Panel »Page 20 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.

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 »Page 122 and place a frame of your choice with ADD »Page 34.

Place Symbols

Load appropriate libraries with USE »Page 122 and place symbols (see ADD »Page 34, MOVE »Page 80, DELETE »Page 52, ROTATE »Page 105, NAME »Page 81, VALUE »Page 123). Where a particular component is not available, define a new one with the library editor.

Draw Bus Connections

Using the BUS »Page 43 command, draw bus connections. You can NAME »Page 81 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 »Page 82 command, connect up the pins of the various elements on the drawing. Intersecting nets may be made into connections with the JUNCTION »Page 71 command.

Checking the Schematic

Carry out an electrical rule check (ERC »Page 59) to look for open pins, etc., and use the messages generated to correct any errors. Use the SHOW »Page 112 command to follow complete nets across the screen. Use the EXPORT »Page 61 command to generate a netlist, pinlist, or partlist if necessary.

Generating a Board from a Schematic

By using the BOARD »Page 42 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. Supply 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 »Page 310. 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 »Page 80 and SPLIT »Page 116 commands as appropriate before moving each package on the board. Once all packages have been placed, the RATSNEST »Page 98 command is used to optimize airwires.

Define Restricted Areas

If required, restricted areas for the Autorouter can be defined as RECT »Page 99angles, POLYGON »Page 93s, or CIRCLE »Page 45s 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 »Page 106 command. This function can also be performed automatically by the Autorouter »Page 40, when available.

Checking the Layout

Check the layout (DRC »Page 57) and correct the errors (ERRORS »Page 60). Generate net, part, or pin list if necessary(EXPORT »Page 61).

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 »Page 64.

NAME »Page 81 and place PAD »Page 86s properly.

Add texts >NAME and >VALUE with the TEXT »Page 118 command (show actual name and value in the board) and draw package outlines (WIRE »Page 127 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 »Page 89 and NAME »Page 81 and provide pin parameters (CHANGE »Page 44).

Add texts >NAME and >VALUE with the TEXT »Page 118 command (show actual name and value in the schematic) and draw symbol outlines (WIRE »Page 127 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 »Page 85 command.

Add the gate(s) with ADD »Page 34, you can have as many gates as needed.

Use CONNECT »Page 48 to specify which of the packages pads are connected to the pins of each gate.

Save the library and you can USE »Page 122 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 »Page 24. 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 »Page 22 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 or board 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 library is dropped onto a board or schematic editor window, a library update »Page 121 will be perfomed. 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 *Description*. 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
Save all	save all modified editor files
Refresh Tree	refresh the contents of the tree view
Close project	close the current project
Exit	exit from the program

Options

Directories...	opens the directories dialog »Page 24
Backup...	opens the backup dialog »Page 24
User interface...	opens the user interface dialog »Page 25

Window

Control Panel	Alt+0	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
Control panel	opens the help page you are currently looking at
Product registration	opens the product registration »Page 316 dialog
Product information	opens the product information window, which contains details on your EAGLE license »Page 315

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 »Page 20 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 Rich Text Editor.

Rename

Puts the tree item's text into edit mode, so that it can be renamed. You can also do this by clicking onto the text of the selected tree item.

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 board and schematic. You can also use *Drag&Drop* to do this.

Add to schematic

Starts the ADD »Page 34 command in the schematic window with this device. You can also use *Drag&Drop* to do this.

Add to board

Starts the ADD »Page 34 command in the board window with this package. 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.

Print...

Prints the file to the system printer. See the chapter on printing to the system printer »Page 130 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 »Page 96 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 »Page 98 and then PRINT »Page 96.

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 `';'`) in which to look for the various types of files. When entering an OPEN »Page 83, USE »Page 122, SCRIPT »Page 108 or RUN »Page 107 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. Under 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 »Page 309 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 will be automatically saved when you exit from the program, provided you have created a project file with either the File/Open/Project... or the Options/Save as... commands from the Control Panel »Page 20.

User Interface

The *User interface* dialog allows you to customize the appearance of the layout, schematic and library editor windows »Page 28.

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

Layout

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

Schematic

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

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)

Keyboard and Mouse

Keyboard

Pressing the ESC key when a command is active will cancel the current activity of that command without canceling the entire 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.

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

See also ASSIGN »Page 38 command.

Mouse Buttons

Use the left mouse button for all actions not shown in the following tables.

Usage of the Center Mouse Button

ARC »Page 37	Change active layer
CIRCLE »Page 45	Change active layer
LABEL »Page 72	Change active layer
POLYGON »Page 93	Change active layer
RECT »Page 99	Change active layer
ROUTE »Page 106	Change active layer
SMD »Page 115	Change active layer
TEXT »Page 118	Change active layer
WIRE »Page 127	Change active layer

If the Layer menu does not open by using the center mouse button with the commands mentioned above, use the LAYER »Page 73 command.

Usage of the Right Mouse Button

GROUP »Page 66	Close polygon
ADD »Page 34	Rotate element
INVOKE »Page 70	Rotate gate
LABEL »Page 72	Rotate text
MOVE »Page 80	Rotate element
PAD »Page 86	Rotate pad
PIN »Page 89	Rotate pin
PASTE »Page 88	Rotate paste buffer
ROTATE »Page 105	Rotate group
SMD »Page 115	Rotate smd pad
TEXT »Page 118	Rotate text
ARC »Page 37	Change direction of arc
MIRROR »Page 79	Mirror group
POLYGON »Page 93	Change wire bend
ROUTE »Page 106	Change wire bend
SPLIT »Page 116	Change wire bend
WIRE »Page 127	Change wire bend

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=no)

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 »Page 109 Select_Factor select_radius;
```

defines 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 »Page 20 or by selecting a file from the **File/Open** menu, an editor window suitable for that file will be opened.

- Library Editor »Page 28
- Schematic Editor »Page 29
- Board Editor »Page 28
- Text Editor »Page 29

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 »Page 58 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 »Page 315 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 »Page 29 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 »Page 310 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 »Page 28 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 »Page 310 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 »Page 58 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 »Page 150 and Script files »Page 108, or viewing the results of an Electrical Rule Check »Page 59.

Editor Commands

EAGLE Commands and their Meanings

<u>Change Mode/File Commands</u>

EDIT »Page 58	Load/create library element
WRITE »Page 129	Save drawing/library
OPEN »Page 83	Open library for editing
CLOSE »Page 47	Close library after editing
QUIT »Page 97	Quit EAGLE
EXPORT »Page 61	Generate ASCII list (e.g. netlist)
SCRIPT »Page 108	Execute command file
USE »Page 122	Load library for placing elements
REMOVE »Page 101	Delete files/library elements

<u>Create/Edit Drawings or Libraries</u>

ARC »Page 37	Draw arc
CIRCLE »Page 45	Draw circle
POLYGON »Page 93	Draw polygon
RECT »Page 99	Draw rectangle
WIRE »Page 127	Draw line or routed track
TEXT »Page 118	Add text to a drawing
ADD »Page 34	Add element to drawing/symbol to device
COPY »Page 50	Copy objects/elements
GROUP »Page 66	Define group for upcoming operation
CUT »Page 51	Cut prev. defined group
PASTE »Page 88	Paste prev. cut group to a drawing
DELETE »Page 52	Delete objects
MIRROR »Page 79	Mirror objects
MOVE »Page 80	Move or rotate objects
ROTATE »Page 105	Rotate objects
NAME »Page 81	Name object
VALUE »Page 123	Enter/change value for component
SMASH »Page 114	Prepare NAME/VALUE text for moving
SPLIT »Page 116	Bend wires/lines (tracks, nets, etc.)
LAYER »Page 73	Create/change layer

<u>Special Commands for Boards</u>

SIGNAL »Page 113	Define signal (air line)
ROUTE »Page 106	Route signal
RIPUP »Page 104	Ripup routed track (a whole signal)
DELETE »Page 52	Ripup routed track (one segment)
VIA »Page 124	Place via-hole
HOLE »Page 68	Place hole (without conducting material)
RATSNEST »Page 98	Show shortest air lines
REPLACE »Page 103	Replace component
DRC »Page 57	Perform design rule check
ERRORS »Page 60	Show DRC errors

<u>Special Commands for Schematics</u>

NET »Page 82	Define net
BUS »Page 43	Draw bus line
JUNCTION »Page 71	Place connection point
INVOKE »Page 70	Add certain 'gate' from a placed device
LABEL »Page 72	Provide label to bus or net
GATESWAP »Page 63	Swap equivalent 'gates'
PINSWAP »Page 92	Swap equivalent pins
ERC »Page 59	Perform electrical rule check

BOARD »Page 42	Create a board from a schematic

Special Commands for Libraries

RENAME »Page 102	Rename symbol/package/device
CONNECT »Page 48	Define pin/pad assignment
PACKAGE »Page 85	Define package for device
PREFIX »Page 95	Define default prefix for device
VALUE »Page 123	Define if value text can be changed
PAD »Page 86	Add pad to a package
SMD »Page 115	Add smd pad to a package
PIN »Page 89	Add pin to a symbol
HOLE »Page 68	Define non-conducting hole
REMOVE »Page 101	Delete library elements

Change Screen Display and User Interface

WINDOW »Page 125	Choose screen window
DISPLAY »Page 55	Display/hide layers
ASSIGN »Page 38	Assign keys
CHANGE »Page 44	Change parameters
GRID »Page 64	Define grid/unit
MENU »Page 77	Configure command menu
SET »Page 109	Set program parameters

Miscellaneous Commands

AUTO »Page 40	Start Autorouter
HELP »Page 67	Show help page
INFO »Page 69	Show information about object
MARK »Page 76	Set/remove mark (for measuring)
OPTIMIZE »Page 84	Optimize (join) wire segments
RUN »Page 107	Run User Language Program
SHOW »Page 112	Highlight object
UNDO »Page 120	Undo commands
REDO »Page 100	Redo commands
PRINT »Page 96	Print to the system printer
UPDATE »Page 121	Update library objects

Command Syntax

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 »Page 38 command)
- with command files (see SCRIPT »Page 108 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 ON | OFF;
Input:   SET BEEP ON;
         or
         SET BEEP OFF;
```

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.

Example for entering coordinates as text. 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;
```

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

Copy elements into a drawing.
Copy a symbol into a device.

Syntax

```
ADD package_name[@library_name] 'name' orientation •..
ADD device_name[@library_name]  'name' orientation •..
ADD symbol_name                 'name' options     •..
```

Mouse

Right button rotates the elements.

See also UPDATE »Page 121, USE »Page 122

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.

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 »Page 121 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 »Page 57 (DRC) and an Electrical Rule Check »Page 59 (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").

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. It must be enclosed in apostrophe characters. 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 »Page 95, 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 »Page 70.

Orientation

This parameter gives the orientation of the library element in the drawing. The element is normally rotated using the right mouse button. In Script »Page 108 files textual descriptions of this parameter are used:

Permissible orientations are:

R0	no rotation
R90	rotated once (90 degrees anticlockwise)
R180	rotated twice (180 degrees anticlockwise)
R270	rotated three times
MR0	reflected about the y-axis
MR90	rotated once and reflected about the y-axis
MR180	rotated twice and reflected about the y-axis
MR270	rotated three times and reflected about the y-axis

Default: R0

Example:

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

places a 16-pin DIL package, rotated 90 degrees to the left, 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 »Page 42 command). This can be prevented with the "SET »Page 109 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 »Page 44 command. The value entered with the ADD command is also retained as a default value.

Swaplevel

The swaplevel is a number in the range 0..255, to which the following rules apply:

0:	The symbol (gate) can not be swapped with any other in the schematic.
1..255	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 »Page 70.
`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 • • •
ARC width • • •
ARC CW width • • •
ARC CCW width • • •
```

Mouse

Right button changes orientation.
Center button changes active layer.

See also CHANGE »Page 44, WIRE »Page 127, CIRCLE »Page 45

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 textually. This is useful for script files.

CW: Defines curve sense to be clockwise
CCW: Defines curve sense to be counter clockwise

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.

Arcs 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.

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) and A (Alt)
key          = F1..F12, A-Z, 0-9, BS (Backspace)
```

See also SCRIPT »Page 108, Keyboard and Mouse »Page 26

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 Shift, Ctrl and/or Alt).

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` or `C`.

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 keys 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
F7	MOVE	MOVE command
F8	SPLIT	SPLIT command
F9	UNDO	UNDO command
F10	REDO	REDO command
Alt+BS	UNDO	UNDO command
Shift+Alt+BS	REDO	REDO command

AUTO

Function

Starts the Autorouter

Syntax

```
AUTO;
AUTO signal_name..;
AUTO ! signal_name..;
AUTO •..;
```

See also SIGNAL »Page 113, ROUTE »Page 106, WIRE »Page 127, RATSNEST »Page 98, SET »Page 109

The AUTO command activates the integrated Autorouter »Page 146. 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.

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).

The menu of the Autorouter appears after AUTO (without semicolon) has been entered. You can then adjust the parameters and start the routing run. A job file (name.JOB) and a control file (name.CTL) are generated automatically. These files are necessary if you want to continue an interrupted routing run later on. If you don't want to start the routing run you can create these files with "Create Job".

On the left of the menu you can enter the preferred directions for any layer (first entry) and their basic costs (second entry). If you want the Autorouter not to use a layer, enter "0" into the preferred direction field.

All parameters are "global" except the groups "Costs" and "Maximum" which can be different for any routing run.

The individual passes are selected or deselected by clicking the check boxes on their right. The "Route" pass cannot be deselected. The actual grid unit applies to all values.

Polygons

When the Autorouter is started all Polygons »Page 93 are calculated, except the ones present in the outline mode whose signals are not to be routed ("AUTO ! signal_name..;").

Protocol File

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

Board Size

The Autorouter puts a rectangle in the Dimension layer round all wires and takes the size of this rectangle as the routing area. Wires (tracks) in the Dimension layer appear to be border lines for the Autorouter. This means you can delimit the route area with closed polygons drawn 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.

Please note that the autorouter cannot place vias on to the filled areas of calculated polygons. Polygons forming ground areas, for instance, should therefore be placed only after all signals have been routed (except the ground signal).

<u>Restricted Areas</u>

Rectangles, polygons, and circles 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, enter "0" into the preferred direction field.

BOARD

Function

Converts a schematic into a board.

Syntax

```
BOARD
```

See also EDIT »Page 58

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.

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 »Page 101 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 »Page 316: 100x80mm), where the outlines are placed in such a way that the board is centered between the 50mil grid. All packages and connections are shown on the left side of the board. Supply pins are already connected (see PIN »Page 89 command).

If you need board outlines different to the ones that are generated by default, simply delete the respective lines and use the WIRE »Page 127 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 no connects"). Exception: device without pins (e.g. frames)

BUS

Function

Draws buses in a schematic.

Syntax

```
BUS •..
BUS bus_name •..
```

See also NET »Page 82, NAME »Page 81, SET »Page 109

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".

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).

CHANGE

Function

Changes parameters.

Syntax

```
CHANGE option • •..
```

Mouse

Right button changes group.

The CHANGE command is used to change or preset attributes 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 attributes 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 »Page 66 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 height	`CHANGE SIZE value`
Text line width	`CHANGE RATIO ratio`
Text font	`CHANGE FONT VECTOR | PROPORTIONAL | FIXED`
Wire width	`CHANGE WIDTH value`
Wire style	`CHANGE STYLE value`
Pad shape	`CHANGE SHAPE SQUARE | ROUND | OCTAGON | XLONGOCT | YLONGOCT`
Pad/via diameter	`CHANGE DIAMETER diameter`
Pad/via/hole drill	`CHANGE DRILL value`
Smd dimensions	`CHANGE SMD width height`
Pin parameters	`CHANGE DIRECTION NC | IN | OUT | I/O | OC | HIZ | SUP | PAS | PWR | SUP`
	`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 ON | OFF`
	`CHANGE ORPHANS ON | OFF`
	`CHANGE ISOLATE distance`
	`CHANGE POUR SOLID | HATCH`
	`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 name [variant] | 'variant' [name]`
Technology	`CHANGE TECHNOLOGY name [variant] | 'variant' [name]`

CIRCLE

Function

Adds circles to a drawing.

Syntax

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

Mouse

Center button changes the active layer.

See also CHANGE »Page 44, WIRE »Page 127

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 ] ] ]
```

See also Design Rules »Page 148, NET »Page 82, SIGNAL »Page 113, CHANGE »Page 44

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 Design Rule Check and Autorouter need to do their 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. Any changes made by the CLASS command will not be stored in the UNDO/REDO buffer.

CLOSE

Function

Closes an editor window.

Syntax

```
CLOSE
```

See also OPEN »Page 83, EDIT »Page 58, WRITE »Page 129, SCRIPT »Page 108

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 package pads to symbol pins.

Syntax

```
CONNECT
CONNECT symbol_name.pin_name pad_name..
CONNECT pin_name pad_name..
```

See also PREFIX »Page 95, OPEN »Page 83, CLOSE »Page 47, SCRIPT »Page 108

This command is used in the device editing mode in order to define the relationship between the pins of a symbol 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 Symbol

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

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

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

Device with Several Symbols

If several symbols are present in a device, parameters must be entered with symbol_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 symbols - 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.

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 and elements.

Syntax

```
COPY • •..
```

See also GROUP »Page 66, CUT »Page 51, PASTE »Page 88, POLYGON »Page 93

The COPY command is used to select objects or elements and to copy them within the same drawing. 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 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.

Copy Elements

Using the COPY command, elements may be placed in a drawing without opening a library if the elements are already in use.

CUT

Function

Loads a group into the paste buffer.

Syntax

```
CUT;
CUT •
```

See also PASTE »Page 88, GROUP »Page 66

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 to put the selected objects and elements into the buffer. Now you can change to another board or package library (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 paste buffer.

DELETE

Function

Deletes objects and elements.

Syntax

```
DELETE •..
DELETE SIGNALS
```

Mouse

Right button deletes group.

See also RIPUP »Page 104, DRC »Page 57, GROUP »Page 66

The DELETE command is used to delete the object selected by the mouse.

Clicking the right mouse button deletes a previously defined GROUP »Page 66.

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 »Page 98 command.

With active Forward&Back Annotation »Page 310, 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 »Page 104 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.

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 »Page 310) no signals are connected to the component (see also REPLACE »Page 103). 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 a new or changed netlist (see EXPORT). Only those signals are deleted which are connected to pads.

Don't forget: Deleting can be reversed by the UNDO »Page 120 command!

DESCRIPTION

Function

Defines the description of a device, package or library.

Syntax

```
DESCRIPTION
DESCRIPTION description_string;
```

See also CONNECT »Page 48, PACKAGE »Page 85, VALUE »Page 123

This command is used in the library editor to define or edit the description of a device, package or library.

The `description_string` may contain Rich Text »Page 307 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 Rich Text »Page 307 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 rich text tags.

The description of a library can be defined or modified via the command line only if the library is newly opened, and no device, symbol or package has been edited yet. It can always be defined via the pulldown menu "Library/Description...".
The description of a device set or package can always be edited via the command line, or via the pulldown menu "Edit/Description...".

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.

DISPLAY

Function

Selects the visible layers.

Syntax

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

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

See also LAYER »Page 73, PRINT »Page 96

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 the Bottom layer is displayed.

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 "-".

If tPlace or bPlace is selected, t/bNames, t/bValues, t/bOrigins and t/bDocu are automatically selected, too.

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

If t/bPlace is selected or deselected in the DISPLAY menu, the layers t/bNames, t/bValues, and t/bOrigins are selected or deselected, too. If Symbols is selected/deselected, the layers Names and Values are selected/deselected, too.

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 black or white), 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 »Page 96.

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!

DRC

Function

Checks design rules.

Syntax

```
DRC
DRC • • ;
```

See also Design Rules »Page 148, CLASS »Page 46, SET »Page 109, ERC »Page 59, ERRORS »Page 60

The command DRC checks a board against the current set of Design Rules »Page 148.

Only those signal layers that are currently active will be checked, so in order to make sure everything is ok all used signal layers should be active when running the DCR (at least for the final check before manufacturing).

The errors found are displayed as error polygons in the respective layers, and can be browsed through with the ERRORS »Page 60 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.

To delete all error polygons use the command

```
ERRORS CLEAR
```

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
```

See also OPEN »Page 83, CLOSE »Page 47, BOARD »Page 42

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 name.s1`	loads sheet 1 of a schematic.
`EDIT name.s99`	loads sheet 99 of a schematic

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 »Page 28 (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.

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 »Page 24.

ERC

Function
Electrical Rule Check.

Syntax
`ERC`

See also DRC »Page 57, Consistency Check »Page 311

This command is used to test schematics for electrical errors. The summary is written into a text file with the extension `.ERC`.

The following warning messages may be generated:

- SUPPLY Pin Pin_Name overwritten with Net_Name
- NC Pin Elem._Name Pin_Name connected to Net_Name
- POWER Pin El._Name Pin_N. connected to Net_Name
- only one Pin on net Net_Name
- no Pins on net Net_Name
- SHEET Sheet_Nr.: unconnected Pin: Element_N. Pin_N.
- SHEET Sheet_Nr.: Junction at (x, y) appears to connect nets Net_Name and Net_Name
- SHEET Sheet_Nr., NET Net_Name: missing Junction at (x, y)
- SHEET Sheet_Nr., NET Net_Name: close but unconnected wires at (x, y)
- SHEET Sheet_Nr.: NETS Net_Name and Net_Name too close at (x, y)
- SHEET Sheet_Nr., NET Net_Name: wire overlaps pin at (x, y)
- SHEET Sheet_Nr.: pins overlap at (x, y)

The following error messages may be generated:

- no SUPPLY for POWER Pin Element_Name Pin_Name
- no SUPPLY for implicit POWER Pin El._Name Pin_Name
- unconnected INPUT Pin: Element_Name Pin_Name
- only INPUT Pins on net Net_Name
- OUTPUT and OC Pins mixed on net Net_Name
- n OUTPUT Pins on net Net_Name
- OUTPUT and SUPPLY Pins mixed on net OUTNET

Additional messages may be produced by the automatic Consistency Check »Page 311.

Consistency Check

The ERC command also performs a Consistency Check »Page 311 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 »Page 310 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 »Page 310 will still be performed, but the supply pin configuration should be checked!

ERRORS

Function
Shows the errors found by the DRC command.

Syntax
```
ERRORS
ERRORS CLEAR
```

Mouse
Left button shows the error.
Double click shows the error in the center of the window.

See also DRC »Page 57

The command ERRORS is used to show the errors found by the Design Rule Check (DRC). If selected, a window is opened in which all errors found by the DRC are listed. Selecting an entry with the mouse causes the error to be marked 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.

The "Del" button in the ERRORS dialog can be used to delete a particular error polygon. To delete all error polygons you can use 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 resolution;
```

See also SCRIPT »Page 108, RUN »Page 107

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 as defined in the directories dialog »Page 24.

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.

When exporting a library, the actual grid unit is used.

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 a previously routed track have been deleted with `DELETE SIGNALS`.

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
`.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.

GATESWAP

Function

Swaps equivalent gates on a schematic.

Syntax

```
GATESWAP • •..;
GATESWAP gate_name gate_name..;
```

See also ADD »Page 34

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 »Page 108

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 0.1 micron
`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

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. It is equivalent to: `GRID OFF DOTS INCH 0.05 1;`

GROUP

Function

Defines a group.

Syntax

```
GROUP •..
GROUP;
```

Mouse

Right button closes the polygon.
Left-click&drag defines a rectangular group.

See also CHANGE »Page 44, CUT »Page 51, PASTE »Page 88, MIRROR »Page 79, DELETE »Page 52

The GROUP command is used to define a group of objects and elements for a succeeding command. Also a whole drawing or an element can be defined as a group. Objects are selected - after activating the GROUP command - 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 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.

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 »Page 124, PAD »Page 86, CHANGE »Page 44

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 succeeding 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.

Holes generate Annulus symbols in supply layers.

In the layers tStop and bStop, holes generate the solder stop mask, whose diameter is calculated by the hole diameter plus the value frame defined with the option -B.

INFO

Function

Displays object attributes.

Syntax

```
INFO •..
```

See also CHANGE »Page 44, SHOW »Page 112

The INFO command displays further details about an object's attributes on screen, e.g. wire width, layer number, text size etc.

INVOKE

Function

Call a specific symbol from a device.

Syntax

```
INVOKE • orientation •
INVOKE part_name gate_name orientation •
```

Mouse

Right button rotates the gate 90 degrees.

See also COPY »Page 50

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 menu which 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.

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 gate has to be specified in the INVOKE command. In this case the right column of the popup menu shows the sheet number where the already used gates are placed. A gate placed on the current sheet is indicated by an asterisk.

JUNCTION

Function

Places a dot at intersecting nets.

Syntax

```
JUNCTION •..
```

See also NET »Page 82

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 • •..
```

Mouse

Right button rotates the label.
Center button selects layer for the label.

See also NAME »Page 81, BUS »Page 43

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 new location for the text.

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.

LAYER

Function

Changes and defines layers.

Syntax

```
LAYER layer_number
LAYER layer_name
LAYER layer_number layer_name
LAYER -layer_number
```

See also DISPLAY »Page 55

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.

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.

The predefined standard layers cannot be deleted.

Supply Layers

Layers 2...15 are treated by the Autorouter as supply layers, provided their name starts with the $ character and there is a signal with an identical name but without the leading $.

Predefined EAGLE Layers

Layout

1 Top	Tracks, top side
2 Route2	Inner layer (signal or supply)
3 Route3	Inner layer (signal or supply)
4 Route4	Inner layer (signal or supply)
5 Route5	Inner layer (signal or supply)
6 Route6	Inner layer (signal or supply)
7 Route7	Inner layer (signal or supply)
8 Route8	Inner layer (signal or supply)
9 Route9	Inner layer (signal or supply)
10 Route10	Inner layer (signal or supply)
11 Route11	Inner layer (signal or supply)
12 Route12	Inner layer (signal or supply)
13 Route13	Inner layer (signal or supply)
14 Route14	Inner layer (signal or supply)
15 Route15	Inner layer (signal or supply)
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

MARK

Function

Defines a mark on the drawing area.

Syntax

```
MARK •
MARK;
```

See also GRID »Page 64

The MARK command allows you to define a point on the drawing area and display the distance of the mouse cursor relative to that point at the upper left corner of the screen (with a leading @ 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.

MENU

Function
Customizes the textual command menu.

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

See also ASSIGN »Page 38, SCRIPT »Page 108

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

The complete syntax specification for the `option` parameters is

```
option    := command | menu | delimiter
command   := text [ ':' text ]
menu      := text '{' option [ '|' option ] '}'
delimiter := '---'
```

A menu option can either be a simple command, as in

```
MENU Display Grid;
```

which would set the menu to the commands `Display` and `Grid`; 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 when the respective button is clicked; or a submenu button as in

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

which would define a button labelled `Grid` that, when clicked opens a submenu with the two options `Fine` and `Coarse`.

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

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;\
         }';
```

Example

```
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.

MIRROR

Function

Mirrors objects and groups.

Syntax

```
MIRROR •..
```

Mouse

Right button selects group.

See also ROTATE »Page 105, TEXT »Page 118

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.

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/tracks are mirrored, too (beware of short circuits!).

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, rectangles, polygons and labels 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.

Text on a schematic cannot be mirrored (this is also the case when using the CAM Processor).

MOVE

Function

Moves objects and elements.

Syntax

```
MOVE • •..
MOVE package_name •..
```

Mouse

Right button rotates the element or selects a group.
Left-click&drag immediately moves the object.

Keyboard

`F7: MOVE` activates the MOVE command.

See also GROUP »Page 66, RATSNEST »Page 98

The MOVE command is used to move objects. Packages can also be selected with the package name. This is especially useful if the package is not in the currently shown screen window.

Components 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"). In this mode, however, it is not possible to rotate or mirror the object while moving it.

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.

NAME

Function

Displays and changes names.

Syntax

```
NAME •..
NAME new_name •
NAME old_name new_name
```

See also SHOW »Page 112, SMASH »Page 114, VALUE »Page 123

Drawing

When in drawing edit mode, the NAME command is used to display or edit the name of the selected element, signal, net, or bus.

The *old_name new_name* syntax can be used only inside a board to rename an element.

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.

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).

NET

Function

Draws nets on a schematic.

Syntax

```
NET • •..
NET net_name • •..
```

See also BUS »Page 43, NAME »Page 81, CLASS »Page 46, SET »Page 109

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".

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.

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).

OPEN

Function

Opens a library for editing.

Syntax

```
OPEN library_name
```

See also CLOSE »Page 47, USE »Page 122, EDIT »Page 58, SCRIPT »Page 108

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 together.

Syntax

```
OPTIMIZE;
OPTIMIZE signal_name ..
OPTIMIZE •..
```

Mouse

Right button executes the command for previously defined groups.

See also SET »Page 109, SPLIT »Page 116, MOVE »Page 80, ROUTE »Page 106

The OPTIMIZE command joins wire segments in a signal layer 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.

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 name
PACKAGE -old_name new_name
PACKAGE -name
```

See also CONNECT »Page 48, TECHNOLOGY »Page 117, PREFIX »Page 95

This command is used in the device edit mode to define, delete or rename a package variant.

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 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 »Page 42 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.

PAD

Function

Adds pads to a package.

Syntax

```
PAD •..
PAD pad_diameter pad_shape 'name' •..
```

Mouse

Right button rotates elongated pads.

See also SMD »Page 115, CHANGE »Page 44, DISPLAY »Page 55, SET »Page 109, NAME »Page 81, VIA »Page 124, Design Rules »Page 148

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 the pad at the current position and attaches a new pad symbol to the cursor. 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).

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 succeeding operations.

Pad Shapes

A pad can have one of the following shapes (pad_shape):

Square	
Round	
Octagon	octagonal
XLongOct	elongated in x direction
YLongOct	elongated in y direction

The two sides of elongated pads have a fixed ratio of 2:1. The smaller one has to be entered as the parameter when such a pad is defined.

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 »Page 148 of the board the part is used in.

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 ON | OFF;
```

This change will be visible after the next screen refresh.

<u>Single Pads</u>

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.

<u>Alter Package</u>

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 paste buffer to a drawing.

Syntax

```
PASTE •
```

Mouse

Right button rotates the copy.

See also CUT »Page 51, GROUP »Page 66

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.
- Elements, 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. If no label is present, a new name is generated if the previous name is already in use.

If there are modified versions of devices or packages in the paste buffer, an automatic library update »Page 121 will be started to replace the objects in the schematic or board with the ones from the paste buffer. **Note: You should always run a Design Rule Check »Page 57 (DRC) and an Electrical Rule Check »Page 59 (ERC) after a library update has been performed!**

PIN

Function

Defines connection points for symbols.

Syntax

```
PIN 'name' options •..
```

Mouse

Right button rotates the Pin.

See also NAME »Page 81, SHOW »Page 112, CHANGE »Page 44

<u>Options</u>

There are six possible options:

Direction
Function
Length
Orientation
Visible
Swaplevel

<u>Direction</u>

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)
I/O	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: I/O

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.

<u>Function</u>

The graphic representation of the pin:

None	no special function
Dot	inverter symbol
Clk	clock symbol
DotClk	inverted clock symbol

Default: None

<u>Length</u>

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

A number between 0 and 255. 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.

PINSWAP

Function

Swap pins or pads.

Syntax

```
PINSWAP • •..
```

See also PIN »Page 89

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 »Page 310 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 wire_width • • •..
```

Mouse

Double click with left button closes polygon.
Center button selects the layer.
Right button changes the wire bend (see SET Wire_Bend »Page 109).

See also CHANGE »Page 44, DELETE »Page 52, RATSNEST »Page 98, RIPUP »Page 104

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.

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 »Page 134.
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.

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.

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 RATSNEST »Page 98 command starts the calculation of the polygons (this can be turned off with `SET »Page 109 POLYGON_RATSNEST OFF;`).

The RIPUP »Page 104 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] or Hatch).

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` for signal polygons and `0` or `7` for polygons in packages. Polygons with the same rank are checked against each other by the Design Rule Check »Page 57. The rank parameter only has a meaning for polygons in signal layers (`1..16`) and will be ignored for polygons in other layers. The default is `1` for signal polygons and `7` for package polygons.

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 (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.

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 none of the polygon parts has any such connection, 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.

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 »Page 145 (for example for milling prototype boards). This name should not be used otherwise.

PREFIX

Function

Defines the prefix for a symbol name.

Syntax

```
PREFIX prefix_string;
```

See also CONNECT »Page 48, PACKAGE »Page 85, VALUE »Page 123

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 »Page 134, printing to the system printer »Page 130

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.

If you want to print pads and vias "filled" (without the drill holes being visible), use the command

```
SET »Page 109 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 »Page 98 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.

If the PRINT command is not terminated with a `';'`, a print dialog »Page 131 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 »Page 131 (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

If an `option` 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`.

Examples

`PRINT`	opens the print dialog »Page 131 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

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
```

See also SIGNAL »Page 113, MOVE »Page 80, POLYGON »Page 93, RIPUP »Page 104

The RATSNEST command assesses all of 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 »Page 108 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 with polygons. All of the calculated areas are then being displayed in the "real mode". Normally, this decreases the screen redraw speed. 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 »Page 109 POLYGON_RATSNEST OFF;`.

RATSNEST ignores airwires representing signals which have their own layer in a multilayer board (e.g. layer $GND for signal GND), apart from signals connecting smd pads to a supply layer with a via-hole.

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 a thick dot 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 »Page 106 command just like ordinary airwires. They may also be handled by placing a VIA »Page 124 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.

RECT

Function
Adds rectangles to a drawing.

Syntax
```
RECT • •..
```

Mouse
Center button changes the active layer.

See also CIRCLE »Page 45

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.

Rectangles are filled with the layer's color, so deleting a rectangle may erase the drawing area it has covered. In such a case the "WINDOW;" command (F2) should be entered to redraw the screen.

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 »Page 120, Forward&Back Annotation »Page 310

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, AUTO 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 »Page 83, RENAME »Page 102

Files

The REMOVE command is used to delete the file "name" if in board 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.

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 no. 3 from the presently loaded schematic.

If you delete the currently loaded sheet, sheet no. 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.

UNDO does not work with this command. If you have deleted a sheet accidentally it will be present in the "old" schematic file as long as the "new" file has not been saved.

REMOVE clears the UNDO buffer.

RENAME

Function

Renames symbols, devices or packages.

Syntax

```
RENAME old_name new_name;
```

See also OPEN »Page 83

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 package on a board.

Syntax

```
REPLACE package_name •..
```

See also SET »Page 109, UPDATE »Page 121

The REPLACE command has two different modes that are chosen by the SET command. In both modes it is possible to substitute a component with another component.

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.

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.

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.

The REPLACE command can't be used while Forward&Back Annotation »Page 310 is active. Use the CHANGE »Page 44 PACKAGE or UPDATE »Page 121 command in that case.

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 »Page 121 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 »Page 57 (DRC) and an Electrical Rule Check »Page 59 (ERC) after a library update has been performed!**

RIPUP

Function

Changes routed wires and vias into airwires.
Changes the display of polygons to "outlines".

Syntax

```
RIPUP;
RIPUP •..
RIPUP name..
RIPUP ! name..
```

The RIPUP command can be applied to a previously selected group (see GROUP command).

See also DELETE »Page 52, GROUP »Page 66, POLYGON »Page 93, RATSNEST »Page 98

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).

In the latter case routed wires and vias are converted to airwires. Selecting an airwire with RIPUP converts all adjacent routed wires and vias into airwires, up to the next pad, smd or airwire.

```
RIPUP name..
```

removes the complete signal "name" (several signals may be listed, e.g. `RIPUP D0 D1 D2;`).

```
RIPUP •..
```

removes 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).

Polygons

If the RIPUP command is applied to a signal which contains a polygon the polygon will be displayed with its outlines (faster screen redraw!). After the RATSNEST command polygons are displayed in the "real mode".

The RIPUP command can also be used with groups.

ROTATE

Function

Rotates objects and elements.

Syntax

```
ROTATE •..
```

Mouse

Right button rotates previously selected group.

See also ADD »Page 34, MIRROR »Page 79, MOVE »Page 80, GROUP »Page 66

The ROTATE command is used to change the direction of objects in steps of 90 degrees.

Packages

When rotating a package, wires (tracks) connected to the element are moved at the connection points (beware of short circuits!).

Packages can only be rotated if the appropriate tOrigins/bOrigins layer is visible.

Objects

Wires, circles, pads, rectangles, polygons, and labels cannot be rotated individually unless defined as a group.

Text

Text is always displayed so that it can be read from in front 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!

ROUTE

Function

Converts unrouted connections into routed wires (tracks).

Syntax

```
ROUTE • •..
ROUTE wire_width • •..
ROUTE • wire_width •..
```

Mouse

Right button changes the wire bend (see SET Wire_Bend »Page 109).
Center button changes layer.

See also AUTO »Page 40, UNDO »Page 120, WIRE »Page 127, SIGNAL »Page 113, SET »Page 109, RATSNEST »Page 98

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 »Page 127). The right mouse button will change the wire bend and the center mouse button will change the layer. 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.

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.

While the ROUTE command is active the wire width can be entered from the keyboard.

Routing to and from Smds

If you start routing at an Smd, that Smd's layer will automatically be the current routing layer.

If you end routing at an Smd's coordinates, the route will only be considered finished if the last wire is placed on the Smd's layer.

Snap Function

If an airwire is routed very close to the end point the last wire is placed automatically. 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. If "number" is 0 the snap function is disabled.

Wire Angle

If you route airwires connected to pads which are not positioned at a grid point, this might not be possible with 45-degree angles. In this case the command

```
SET SNAP_BENDED OFF;
```

can be executed so that this pad can be connected, even if the parameter Wire_Bend is set to 1 or 3.

RUN

Function

Executes a User Language »Page 150 Program.

Syntax

```
RUN file_name [argument ...]
```

See also SCRIPT »Page 108

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 »Page 225 `argc` and `argv`.

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() »Page 244` 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() »Page 244` 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 »Page 109, MENU »Page 77, ASSIGN »Page 38, EXPORT »Page 61, RUN »Page 107

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 "file_name" can contain any extension. If no extension is given 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

Continued Lines

SCRIPT files contain one or more commands in every line according to the syntax rules. Only the MENU command can make it necessary to use more than one line. In this case please note that every opening apostrophe must have a counterpart (closing apostrophe) in the same line. 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 »Page 24 - is executed each time a new drawing is loaded into an editor window (or when the drawing type is changed in a library).

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 »Page 106 command (using the current unit).
Default: 20 mil
If tracks are being laid with the ROUTE »Page 106 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 SNAP_BENDED ON | OFF;`
If set *on* then the bend in the routed track will be on the grid in *wire_bends* 1 and 3.
`SET SELECT_FACTOR value;`
This setting controls the distance from the cursor within which nearby objects will be suggested for selection »Page 27. The value is entered relative to the height of the presently visible part of the diagram.
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 Top, Bottom, Pads, Vias, Unrouted, Dimension, Drills and Holes will in any case remain in the menu, as will the schematic layer. Signal layers in which tracks are present also remain in the menu.`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..;
```

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).
Example:

```
Grid Inch;
Set Width_Menu 0.1 0.2 0.3;
```

Bend angle for wires

`SET WIRE_BEND bend_nr;`
style_nr can be varied from 0 to 4.

```
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
```

Beep on/off

`SET BEEP ON | OFF;`

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 GRID_REDRAW ON	OFF;` Refresh display or not when grid has been altered.

	`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 ON | OFF;` Pad names are displayed/not displayed.
Bus line display	`SET BUS_WIRE_WIDTH width;`
DRC »Page 57-Parameter	`SET DRC_FILL fill_name;`
Polygon calculation	`SET POLYGON_RATSNEST ON | OFF;` See POLYGON »Page 93 command.
Vector font	`SET VECTOR_FONT ON | OFF;` See TEXT »Page 118 command.

Mode parameters

Package check	`SET CHECK_CONNECTS ON | OFF;` The ADD »Page 34 command checks whether a pin has been connected to every pad (with CONNECT »Page 48). 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 »Page 103 mode	`SET REPLACE_SAME NAMES | COORDS;`
UNDO »Page 120 buffer on/off	`SET UNDO_LOG ON | OFF;`
Wire optim. on/off	`SET OPTIMIZING ON | OFF;` If set *on*, wires which lie in one line after a MOVE, ROUTE or SPLIT are subsumed into a single wire. See also OPTIMIZE »Page 84.

`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

SHOW

Function

Displays information about the selected object.

Syntax

```
SHOW •..
SHOW name..
```

See also INFO »Page 69

The SHOW command is used to show names and other details of elements and objects. Parameters are listed in the top left corner of the screen. Complete signals and nets can be highlighted with the SHOW command.

Cross Probing

With active Forward&Back Annotation »Page 310 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.

Examples

```
SHOW IC1 IC2;
```

First IC1 is highlighted, then it gets dark and IC2 is highlighted.

```
SHOW IC1;
```

IC1 is highlighted and remains highlighted.

```
SHOW IC2;
```

IC2 is also highlighted.

SIGNAL

Function

Defines signals.

Syntax

```
SIGNAL • •..
SIGNAL signal_name • •..
SIGNAL signal_name package_name pad_name..;
```

See also AUTO »Page 40, ROUTE »Page 106, NAME »Page 81, CLASS »Page 46, WIRE »Page 127, RATSNEST »Page 98, EXPORT »Page 61

The SIGNAL command is used to define signals (connections between the various packages). The user must define a minimum of two package_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 »Page 145. This name should not be used otherwise.

SMASH

Function

Separates >NAME and >VALUE text from elements.

Syntax

```
SMASH •..
```

See also NAME »Page 81, VALUE »Page 123

The SMASH command is used with elements in order to separate the text parameters indicating name and value from the element. The text may then be placed in a new and more convenient location with the MOVE command.

Use of the SMASH command allows the text to be treated as any other text, e.g. CHANGE SIZE, ROTATE, etc., but the actual text may not be changed.

If a text separated with SMASH is deleted, then it appears at its initial position with its initial size.

SMD

Function

Adds smd pads to a package.

Syntax

```
SMD [ x_width y_width ] [ -roundness ] [ 'name' ] •..
```

Mouse

Right button rotates the smd pad.
Center button changes the layer.

See also PAD »Page 86, CHANGE »Page 44, NAME »Page 81, ROUTE »Page 106, Design Rules »Page 148

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 the smd pad at the current position and attaches a new smd symbol to the cursor. 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 succeeding SMD commands and can be changed with the CHANGE command. Only even values should be used. Pressing the center mouse button changes the layer onto which the smd pad will be drawn, and the right mouse button will rotate the smd pad. The SMD command is active until ';' (or another command) is entered.

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.

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.

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 • •..
```

Mouse

Right button changes the wire bend (see SET Wire_Bend »Page 109).

Keyboard

`F8: SPLIT` activates the SPLIT command.

See also MOVE »Page 80, OPTIMIZE »Page 84, SET »Page 109

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.

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 »Page 85

This command is used in the device editor mode to define the possible *technology* parts of a device name. 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 »Page 85 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 quotes. Using `'-*'` removes all technologies.

Only ASCII characters in the range 33..126 may be used in technologies, and the maximum number of technologies per device is 254.

The special "empty" technology can be entered as two quotes (`''`, an empty string).

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

Right button rotates the text.
Center button changes the layer.

See also CHANGE »Page 44, MOVE »Page 80, MIRROR »Page 79, PIN »Page 89, ROTATE »Page 105

The TEXT command is used to add text to a library element or drawing. When entering several text labels 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 PIN 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.

Special Characters

If the text contains several succeeding 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 of characters and the line width can be changed with the CHANGE commands:

```
CHANGE SIZE text_size •..
CHANGE RATIO ratio •..
```

Maximum text height: 2 inches
Maximum line width: 0.51602 inch (13.1 mm)
Ratio: 0...31 (% 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 »Page 25, all texts will always be displayed and printed using the builtin vector font. 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 »Page 109 VECTOR_FONT ON|OFF` command to change the setting in an existing board or schematic drawing.

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`.

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

`>NAME`	Component name (ev.+gate name) 1)
`>VALUE`	Comp. value/type 1)
`>PART`	Component name 2)
`>GATE`	Gate name 2)
`>DRAWING_NAME`	Drawing name
`>LAST_DATE_TIME`	Time of the last modification
`>PLOT_DATE_TIME`	Time of the plot creation
`>SHEET`	Sheet number of schematic 3)

1) Only for package or symbol
2) Only for symbol
3) Only for symbol or schematic

UNDO

Function
Cancels previous commands.

Syntax
```
UNDO;
```

Keyboard
`F9:` `UNDO` execute the UNDO command. `Alt+BS: UNDO`

See also REDO »Page 100, SET »Page 109, Forward&Back Annotation »Page 310

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, AUTO, or REMOVE command is reached. It is not possible to "undo" window operations.

The UNDO command uses up disk space. If you are short of this you can switch off this function with the SET command

```
SET UNDO_LOG OFF;
```

After executing the UNDO command, objects on the screen may appear to be partly destroyed. Refresh the screen (function key F2) in such a case.

UNDO/REDO is completely integrated within Forward&Back Annotation.

UPDATE

Function

Updates library objects.

Syntax

```
UPDATE
UPDATE;
UPDATE library_name..;
```

See also ADD »Page 34, REPLACE »Page 103

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 you activate the UPDATE command without a parameter, a file dialog will be presented to select the library from which to update.

If the command is terminated with a `';'`, but has no parameters, all parts will be checked.

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").

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 »Page 24, 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 »Page 310 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.

Note: You should always run a Design Rule Check »Page 57 (DRC) and an Electrical Rule Check »Page 59 (ERC) after a library update has been performed!

USE

Function

Marks a library for use.

Syntax

```
USE
USE -*;
USE library_name..;
```

See also ADD »Page 34, REPLACE »Page 103

The USE command marks a library for later use with the ADD »Page 34 or REPLACE »Page 103 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 »Page 121 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 »Page 20 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 -* C:\eagle\lbr;`	first drops all previously marked libraries and then marks all libraries from the directory C:\eagle\lbr

VALUE

Function

Displays and changes values.

Syntax

```
VALUE •..
VALUE value •..
VALUE name value ..
VALUE ON;
VALUE OFF;
```

See also NAME »Page 81, SMASH »Page 114

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.

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.

VIA

Function

Adds vias to a board.

Syntax

```
VIA •..
VIA 'signal_name' diameter shape •..
```

See also SMD »Page 115, CHANGE »Page 44, DISPLAY »Page 55, SET »Page 109, PAD »Page 86, Design Rules »Page 148

When the VIA command is active, a via symbol is attached to the cursor. Pressing the left mouse button places the via at the current position and attaches a new via symbol to the cursor. When vias are added to a signal, a short circuit test is performed. 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 »Page 57 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

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 »Page 148 of the board the via is used in.

WINDOW

Function

Zooms in and out of a drawing.

Syntax

```
WINDOW;
WINDOW •;
WINDOW • •;
WINDOW • • •
WINDOW scale_factor
WINDOW FIT
```

Mouse

Left-click&drag works as 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.

WIRE

Function

Adds wires (tracks) to a drawing.

Syntax

```
WIRE • •..
WIRE 'signal_name' wire_width • •..
WIRE • wire_width •..
```

Mouse

Right button changes the wire bend (see SET Wire_Bend »Page 109).
Center button selects the layer.

See also SIGNAL »Page 113, ROUTE »Page 106, CHANGE »Page 44, NET »Page 82, BUS »Page 43, DELETE »Page 52, RIPUP »Page 104

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. Pressing the center mouse button brings a popup menu from which you may select the layer into which the wire will be drawn.

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 »Page 57 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 »Page 44 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 »Page 82, BUS »Page 43 and SIGNAL »Page 113.

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 »Page 24.

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 »Page 310 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 »Page 130
- CAM Processor »Page 134
- Outlines data »Page 145

Printing

The parameters for printing to the system printer can be modified through the following three dialogs:

- Printing a Drawing »Page 131
- Printing a Text »Page 132
- Printer Page Setup »Page 133

See also PRINT »Page 96

Printing a Drawing

If you enter the PRINT »Page 96 command without a terminating `';'`, or select **Print** from the context menu »Page 22 of a drawing's icon in the Control Panel »Page 20, you will be presented a dialog with the following options:

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 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 »Page 22 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 »Page 96 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 (like paper size and orientation).

Page...

Invokes the Printer Page Setup »Page 133.

Printing a Text

If you select **Print** from the context menu »Page 22 of a text file's icon in the Control Panel »Page 20, or from the **File** menu of the Text Editor »Page 29, you will be presented a dialog with the following options:

Wrap

Enables wrapping a line that is 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 (like paper size and orientation).

Page...

Invokes the Printer Page Setup »Page 133.

Printer Page Setup

The Printer Page Setup dialog is 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.

Vertical

Defines the vertical alignment of the drawing to be either **Top**, **Center** or **Bottom**.

Horizontal

Defines the horizontal alignment of the drawing to be either **Left**, **Center** or **Right**.

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`).

Printer...

Invokes the system printer dialog, which enables you to choose which printer to use and to set printer specific parameters (like paper size and orientation).

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 »Page 135
- Select the output device type »Page 137
- Select the output file »Page 142
- Select the plot layers »Page 144
- Adjust the device parameters »Page 138
- Adjust the flag options »Page 143

The CAM Processor allows you to combine several sets of parameter settings to form a CAM Processor Job »Page 136, 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 »Page 130

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

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 »Page 142 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 »Page 139 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 »Page 139 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.

Device Parameters

Depending on the type of output device »Page 137 you have selected, there are several device specific parameters that allow you to adjust the output to your needs:

- Aperture Wheel File »Page 139
- Aperture Emulation »Page 139
- Aperture Tolerances »Page 139
- Drill Rack File »Page 139
- Drill Tolerances »Page 139
- Offset »Page 141
- Page Size »Page 141
- Pen Data »Page 141

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    annulus   0.004 x 0.000
D010    round     0.004
D040    square    0.004
D054    thermal   0.090 x 0.060
D100    rectangle 0.060 x 0.075
D104    oval      0.030 x 0.090
D110    draw      0.004
```

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.

"Annulus" and/or "Thermal" is to be selected if these aperture types are to be emulated (only effective if "Apertures" is selected, too).

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

A drill station 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 »Page 107 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 »Page 29 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
*.ly4	$User1	Inner supply 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.

Alternative Output File Names

If you enter ".EXT" into the Output field the output file "boardname.EXT" will be generated.

In order to avoid the Gerber info file in a Job being overwritten by the following Section, you can enter ".*#" into the Output field (where "*" stands for any number of valid filename characters). The output file name will then be "boardname.*x", and the info file name will be "boardname.*i". If, for instance, a board named "myboard.brd" is loaded and you have entered ".cp#" into the Output field, an output file "myboard.cpx" and a Gerber info file "myboard.cpi" will be generated.

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 »Page 137 that supports colors, you can enter the color number in the *Color* field of each layer.

The following layers and output file names »Page 142 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
*.ly4	$User1	Inner supply 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 »Page 93 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 *Isolate* value must be `'0'`
- 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 »Page 98 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 »Page 150. 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 »Page 195 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 »Page 57 (DRC) with all *Clearance* values for different signals set to at least the diameter of your milling tool!

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 »Page 148!

Autorouter

The integrated Autorouter can be started from a board window with the AUTO »Page 40 command.

Please check your license »Page 315 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 »Page 59)
- Design Rule Check (DRC »Page 57)

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 »Page 57 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 »Page 57 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.

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 of 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.

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 »Page 145 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.

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 usually 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 three 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.

Supply

The *Supply* tab defines the dimensions of Thermal and Annulus symbols used in supply layers.
Please note that the actual shape of supply symbols may be different when generating output for photoplotters that use specific thermal/annulus apertures!

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, and to limit the maximum number of errors that will be reported.

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) »Page 151, and then execute »Page 151 it.

The following sections describe the EAGLE User Language in detail:

Syntax »Page 152	lists the rules a ULP file has to follow
Data Types »Page 163	defines the basic data types
Object Types »Page 166	defines the EAGLE objects
Definitions »Page 207	shows how to write a definition
Operators »Page 211	lists the valid operators
Expressions »Page 215	shows how to write expressions
Statements »Page 218	defines the valid statements
Builtins »Page 225	lists the builtin constants, functions etc.
Dialogs »Page 272	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 »Page 152. 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 »Page 29.

A User Language Program consists of two major items, definitions »Page 207 and statements »Page 218.

Definitions »Page 207 are used to define constants, variables and functions to be used by statements »Page 218.

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 »Page 154` directive is present, its value will be used in the Control Panel »Page 20 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() »Page 244` function can be used to send that command to the editor window.

Executing a ULP

User Language Programs are executed by the RUN »Page 107 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() »Page 244` 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 »Page 107 command is executed as part of a script file »Page 108, the script is terminated if the ULP has exited with a return value other than `0`.

A special variant of the `exit() »Page 244` 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 »Page 153
- Comments »Page 153
- Directives »Page 154
- Keywords »Page 155
- Identifiers »Page 156
- Constants »Page 157
- Punctuators »Page 160

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 »Page 153 are considered *whitespace* and are discarded.

The only place where ASCII characters representing *whitespace* are not discarded is within literal strings »Page 157, 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 »Page 154
#usage »Page 154
```

#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.

#usage

Every User Language Program should contain information about its function, how to use it and maybe who wrote it. The directive

`#usage `*`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 »Page 158) will be used in the Control Panel »Page 20 to display a description of the program.

In case the ULP needs to use this information in, for example, a dlgMessageBox() »Page 274, the `text` is available to the program through the builtin constant »Page 225 `usage`.

Only the `#usage` directive of the main program file (that is the one started with the RUN »Page 107 command) will take effect. Therefore pure include »Page 154 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.

Example

```
#usage "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"
```

Keywords

The following *keywords* are reserved for special purposes and must not be used as normal identifier names:

```
break »Page 220
case »Page 223
char »Page 163
continue »Page 220
default »Page 223
do »Page 221
else »Page 222
enum »Page 208
for »Page 221
if »Page 222
int »Page 163
numeric »Page 209
real »Page 163
return »Page 222
string »Page 164
switch »Page 223
void »Page 163
while »Page 224
```

In addition, the names of builtins »Page 225 and object types »Page 166 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 »Page 208, variable »Page 209 or function »Page 210.

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 »Page 163, there are also different types of constants.

- Character constants »Page 157
- Integer constants »Page 157
- Real constants »Page 158
- String constants »Page 158

Character Constants

A *character constant* consists of a single character or an escape sequence »Page 159 enclosed in single quotes, like

```
'a'
'='
'\n'
```

The type of a character constant is `char »Page 163`.

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 »Page 163`.

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 »Page 163`.

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 »Page 159 enclosed in double quotes, like

```
"Hello world\n"
```

The type of a string constant is `string »Page 164`.

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 »Page 157 and string constants »Page 158.

Examples

```
'\n'
"A tab\tinside a text\n"
"Ring the bell\a\n"
```

Punctuators

The *punctuators* used in a User Language Program are

`[]`	Brackets »Page 160
`()`	Parentheses »Page 160
`{}`	Braces »Page 160
`,`	Comma »Page 162
`;`	Semicolon »Page 162
`:`	Colon »Page 162
`=`	Equal sign »Page 162

Other special characters are used as operators »Page 211 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 »Page 215 (possibly altering normal operator »Page 211 precedence), isolate conditional expressions, and indicate function calls »Page 217 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 »Page 218, as in

```
i = a + b;
```

and it also delimits the init, test and increment expressions of a for »Page 221 statement:

```
for (int n = 0; n < 3; ++n) {
    func(n);
    }
```

Colon

The *colon* indicates the end of a label in a switch »Page 223 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 »Page 213.

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` »Page 163	for single characters
`int` »Page 163	for integral values
`real` »Page 163	for floating point values
`string` »Page 164	for textual information

Besides these basic data types there are also high level Object Types »Page 166, 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 »Page 210, 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 »Page 211, Character Constants »Page 157

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 »Page 157

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 »Page 158

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 `char »Page 163` 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`!

See also Operators »Page 211, Builtin Functions »Page 226, String Constants »Page 158

Implementation details

The data type `string` is actually implemented through native C-type zero terminated strings (i.e. `char[]`). 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() »Page 256` 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"`.

Type Conversions

The result type of an arithmetic expression »Page 215, 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 »Page 163`, `int »Page 163` and `real »Page 163` (in that order). So if, e.g. `a` is of type `int »Page 163` and `b` is of type `real »Page 163`, the result of the expression `a + b` would be `real »Page 163`.

See also Typecast »Page 165

Typecast

The result type of an arithmetic expression »Page 215 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 »Page 163`, `int »Page 163` or `real »Page 163`, and `expression` is any arithmetic expression »Page 215.

When typecasting a `real »Page 163` expression to `int »Page 163`, the fractional part of the value is truncated!

See also Type Conversions »Page 165

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 »Page 266(L) { ... }
schematic »Page 269(S) { ... }
board »Page 264(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) {
  B.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 »Page 187
  GRID »Page 181
  LAYER »Page 185
  DEVICESET »Page 178
    DEVICE »Page 177
    GATE »Page 180
  PACKAGE »Page 189
    PAD »Page 190
    SMD »Page 202
    ARC »Page 169
    CIRCLE »Page 173
    HOLE »Page 182
    RECTANGLE »Page 197
    TEXT »Page 204
    WIRE »Page 206
    POLYGON »Page 195
      WIRE »Page 206
  SYMBOL »Page 203
    PIN »Page 192
    ARC »Page 169
    CIRCLE »Page 173
    RECTANGLE »Page 197
    TEXT »Page 204
    WIRE »Page 206
    POLYGON »Page 195
      WIRE »Page 206
```

Object hierarchy of a Schematic:

```
SCHEMATIC »Page 198
  GRID »Page 181
  LAYER »Page 185
  LIBRARY »Page 187
  SHEET »Page 200
    ARC »Page 169
    CIRCLE »Page 173
    RECTANGLE »Page 197
    TEXT »Page 204
    WIRE »Page 206
    POLYGON »Page 195
      WIRE »Page 206
    PART »Page 191
      INSTANCE »Page 183
    BUS »Page 172
      SEGMENT »Page 199
        TEXT »Page 204
        WIRE »Page 206
    NET »Page 188
      SEGMENT »Page 199
        JUNCTION »Page 184
        PINREF »Page 194
        TEXT »Page 204
        WIRE »Page 206
```

Object hierarchy of a Board:

```
BOARD »Page 171
  GRID »Page 181
  LAYER »Page 185
  LIBRARY »Page 187
  ARC »Page 169
  CIRCLE »Page 173
  HOLE »Page 182
  RECTANGLE »Page 197
  TEXT »Page 204
  WIRE »Page 206
  POLYGON »Page 195
    WIRE »Page 206
  ELEMENT »Page 179
  SIGNAL »Page 201
    CONTACTREF »Page 176
    POLYGON »Page 195
      WIRE »Page 206
    VIA »Page 205
    WIRE »Page 206
```

UL_ARC

Data members

`angle1`	real »Page 163 (start angle, `0.0`...`359.9`°)
`angle2`	real »Page 163 (end angle, `0.0`...`719.9`°)
`layer`	int »Page 163
`radius`	int »Page 163
`width`	int »Page 163
`x1, y1`	int »Page 163 (starting point)
`x2, y2`	int »Page 163 (end point)
`xc, yc`	int »Page 163 (center point)

See also UL_BOARD »Page 171, UL_PACKAGE »Page 189, UL_SHEET »Page 200, UL_SYMBOL »Page 203

Note

Start and end angles are defined mathematically positive (i.e. counterclockwise), with `angle1` < `angle2`.

Example

```
board(B) {
  B.arcs(A) {
    printf("Arc: (%d %d), (%d %d), (%d %d)\n",
           A.x1, A.y1, A.x2, A.y2, A.xc, A.yc);
    }
  }
```

UL_AREA

Data members

`x1, y1`	int »Page 163 (lower left corner)
`x2, y2`	int »Page 163 (upper right corner)

See also UL_BOARD »Page 171, UL_DEVICE »Page 177, UL_PACKAGE »Page 189, UL_SHEET »Page 200, UL_SYMBOL »Page 203

A UL_AREA is an abstract object which gives information about the area covered by an object. For a UL_DEVICE, UL_PACKAGE and UL_SYMBOL the area is defined as the surrounding rectangle of the object definition in the library, so even if e.g. a UL_PACKAGE is derived from a UL_ELEMENT, the package's area will not reflect the elements offset within the board.

Example

```
board(B) {
  printf("Area: (%d %d), (%d %d)\n",
          B.area.x1, B.area.y1, B.area.x2, B.area.y2);
  }
```

UL_BOARD

Data members

`area`	UL_AREA »Page 170
`grid`	UL_GRID »Page 181
`name`	string »Page 164

Loop members

`arcs()`	UL_ARC »Page 169
`circles()`	UL_CIRCLE »Page 173
`classes()`	UL_CLASS »Page 174
`elements()`	UL_ELEMENT »Page 179
`holes()`	UL_HOLE »Page 182
`layers()`	UL_LAYER »Page 185
`libraries()`	UL_LIBRARY »Page 187
`polygons()`	UL_POLYGON »Page 195
`rectangles()`	UL_RECTANGLE »Page 197
`signals()`	UL_SIGNAL »Page 201
`texts()`	UL_TEXT »Page 204
`wires()`	UL_WIRE »Page 206

See also UL_LIBRARY »Page 187, UL_SCHEMATIC »Page 198

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 »Page 164 (`BUS_NAME_LENGTH`)

Loop members

`segments()`	UL_SEGMENT »Page 199

Constants

`BUS_NAME_LENGTH`	max. length of a bus name (obsolete - as from version 4 bus names can have any length)

See also UL_SHEET »Page 200

Example

```
schematic(SCH) {
  SCH.sheets(SH) {
    SH.busses(B) printf("Bus: %s\n", B.name);
    }
  }
```

UL_CIRCLE

Data members

`layer`	int »Page 163
`radius`	int »Page 163
`width`	int »Page 163
`x, y`	int »Page 163 (center point)

See also UL_BOARD »Page 171, UL_PACKAGE »Page 189, UL_SHEET »Page 200, UL_SYMBOL »Page 203

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`	int »Page 163
`drill`	int »Page 163
`name`	string »Page 164 (see Note)
`number`	int »Page 163
`width`	int »Page 163

See also Design Rules »Page 148, UL_NET »Page 188, UL_SIGNAL »Page 201, UL_SCHEMATIC »Page 198, UL_BOARD »Page 171

Note

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 »Page 164 (`CONTACT_NAME_LENGTH`)
`pad`	UL_PAD »Page 190
`signal`	string »Page 164
`smd`	UL_SMD »Page 202
`x, y`	int »Page 163 (center point, see Note)

Constants

`CONTACT_NAME_LENGTH`	max. recommended length of a contact name (used in formatted output only)

See also UL_PACKAGE »Page 189, UL_PAD »Page 190, UL_SMD »Page 202, UL_CONTACTREF »Page 176, UL_PINREF »Page 194

Note

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 »Page 175
`element`	UL_ELEMENT »Page 179

See also UL_SIGNAL »Page 201, UL_PINREF »Page 194

Example

```
board(B) {
  B.signals(S) {
    printf("Signal '%s'\n");
    S.contactrefs(C) {
      printf("\t%s, %s\n", C.element.name, C.contact.name);
      }
    }
  }
```

UL_DEVICE

Data members

`area`	UL_AREA »Page 170
`description`	string »Page 164
`headline`	string »Page 164
`name`	string »Page 164 (`DEVICE_NAME_LENGTH`)
`package`	UL_PACKAGE »Page 189
`prefix`	string »Page 164 (`DEVICE_PREFIX_LENGTH`)
`technologies`	string »Page 164 (see Note)
`value`	string »Page 164 ("On" or "Off")

Loop members

`gates()`	UL_GATE »Page 180

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)

See also UL_DEVICESET »Page 178, UL_LIBRARY »Page 187, UL_PART »Page 191

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. When using the `description` text keep in mind that it may contain newline characters (`'\n'`).

Note

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.

Example

```
library(L) {
  L.devicesets(S) {
    S.devices(D) {
      printf("Device: %s, Package: %s\n", D.name, D.package.name);
      D.gates(G) {
        printf("\t%s\n", G.name);
        }
      }
    }
  }
```

UL_DEVICESET

Data members

`area`	UL_AREA »Page 170
`description`	string »Page 164
`headline`	string »Page 164 (see Note)
`name`	string »Page 164 (`DEVICE_NAME_LENGTH`)
`prefix`	string »Page 164 (`DEVICE_PREFIX_LENGTH`)
`value`	string »Page 164 ("On" or "Off")

Loop members

`devices()`	UL_DEVICE »Page 177
`gates()`	UL_GATE »Page 180

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)

See also UL_DEVICE »Page 177, UL_LIBRARY »Page 187, UL_PART »Page 191

Note

The `description` member returns the complete descriptive text as defined with the DESCRIPTION »Page 54 command, while the `headline` member returns only the first line of the description, without any Rich Text »Page 307 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);
      }
    }
  }
```

UL_ELEMENT

Data members

`angle`	real »Page 163 (`0.0`...`359.9`°)
`mirror`	int »Page 163
`name`	string »Page 164 (`ELEMENT_NAME_LENGTH`)
`package`	UL_PACKAGE »Page 189
`value`	string »Page 164 (`ELEMENT_VALUE_LENGTH`)
`x, y`	int »Page 163 (origin point)

Loop members

`texts()`	UL_TEXT »Page 204

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)

See also UL_BOARD »Page 171, UL_CONTACTREF »Page 176

Note

When processing an element's texts, you have to loop through the element's own `texts()` member as well as the `texts()` member of the element's package »Page 189.

Example

```
board(B) {
  B.elements(E) {
    printf("Element: %s, (%d %d), Package=%s\n",
           E.name, E.x, E.y, E.package.name);
    }
  }
```

UL_GATE

Data members

`addlevel`	int »Page 163 (`GATE_ADDLEVEL_...`)
`name`	string »Page 164 (`GATE_NAME_LENGTH`)
`swaplevel`	int »Page 163
`symbol`	UL_SYMBOL »Page 203
`x, y`	int »Page 163 (origin point)

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)

See also UL_DEVICE »Page 177

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 »Page 163
`dots`	int »Page 163 (0=lines, 1=dots)
`multiple`	int »Page 163
`on`	int »Page 163 (0=off, 1=on)
`unit`	int »Page 163 (`GRID_UNIT_...`)

Constants

`GRID_UNIT_MIC`	microns
`GRID_UNIT_MM`	millimeter
`GRID_UNIT_MIL`	mil
`GRID_UNIT_INCH`	inch

See also UL_BOARD »Page 171, UL_LIBRARY »Page 187, UL_SCHEMATIC »Page 198, Unit Conversions »Page 248

Example

```
board(B) {
  printf("Gridsize=%f\n", B.grid.distance);
  }
```

UL_HOLE

Data members

`drill`	int »Page 163
`x, y`	int »Page 163 (center point)

See also UL_BOARD »Page 171, UL_PACKAGE »Page 189

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 »Page 163 (`0.0`...`359.9`°)
`gate`	UL_GATE »Page 180
`mirror`	int »Page 163
`name`	string »Page 164 (`INSTANCE_NAME_LENGTH`)
`sheet`	int »Page 163 (0=unused, 1..99=sheet number)
`value`	string »Page 164 (`PART_VALUE_LENGTH`)
`x, y`	int »Page 163 (origin point)

Loop members

`texts()`	UL_TEXT »Page 204

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!)

See also UL_PART »Page 191, UL_PINREF »Page 194

Note

When processing an instance's texts, you have to loop through the instance's own `texts()` member as well as the `texts()` member of the instance's gate's symbol »Page 203.

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 »Page 163
`x, y`	int »Page 163 (center point)

See also UL_SEGMENT »Page 199

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_LAYER

Data members

`color`	int »Page 163
`fill`	int »Page 163
`name`	string »Page 164 (`LAYER_NAME_LENGTH`)
`number`	int »Page 163
`visible`	int »Page 163 (0=off, 1=on)

See also UL_BOARD »Page 171, UL_LIBRARY »Page 187, UL_SCHEMATIC »Page 198

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_USER`	

Example

```
board(B) {
  B.layers(L) printf("Layer %3d %s\n", L.number, L.name);
  }
```

UL_LIBRARY

Data members

`description`	string »Page 164 (see Note)
`grid`	UL_GRID »Page 181
`headline`	string »Page 164
`name`	string »Page 164 (`LIBRARY_NAME_LENGTH`, see Note)

Loop members

`devices()`	UL_DEVICE »Page 177
`devicesets()`	UL_DEVICESET »Page 178
`layers()`	UL_LAYER »Page 185
`packages()`	UL_PACKAGE »Page 189
`symbols()`	UL_SYMBOL »Page 203

Constants

`LIBRARY_NAME_LENGTH`	max. recommended length of a library name (used in formatted output only)

See also UL_BOARD »Page 171, UL_SCHEMATIC »Page 198

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 »Page 54 command, while the `headline` member returns only the first line of the description, without any Rich Text »Page 307 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 »Page 174
`name`	string »Page 164 (`NET_NAME_LENGTH`)

Loop members

`pinrefs()`	UL_PINREF »Page 194 (see Note)
`segments()`	UL_SEGMENT »Page 199 (see Note)

Constants

`NET_NAME_LENGTH`	max. recommended length of a net name (used in formatted output only)

See also UL_SHEET »Page 200, UL_SCHEMATIC »Page 198

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.

Example

```
schematic(S) {
  S.nets(N) {
    printf("Net: %s\n", N.name);
    // N.segments(SEG) will NOT work here!
    }
  }
schematic(S) {
  SCH.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 »Page 170
`description`	string »Page 164
`headline`	string »Page 164
`library`	string »Page 164
`name`	string »Page 164 (`PACKAGE_NAME_LENGTH`)

Loop members

`arcs()`	UL_ARC »Page 169
`circles()`	UL_CIRCLE »Page 173
`contacts()`	UL_CONTACT »Page 175
`holes()`	UL_HOLE »Page 182
`polygons()`	UL_POLYGON »Page 195
`rectangles()`	UL_RECTANGLE »Page 197
`texts()`	UL_TEXT »Page 204
`wires()`	UL_WIRE »Page 206

Constants

`PACKAGE_NAME_LENGTH`	max. recommended length of a package name (used in formatted output only)

See also UL_DEVICE »Page 177, UL_ELEMENT »Page 179, UL_LIBRARY »Page 187

Note

The `description` member returns the complete descriptive text as defined with the DESCRIPTION »Page 54 command, while the `headline` member returns only the first line of the description, without any Rich Text »Page 307 tags. When using the `description` text keep in mind that it may contain newline characters (`'\n'`).

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

`diameter[layer]`	int »Page 163
`drill`	int »Page 163
`name`	string »Page 164 (`PAD_NAME_LENGTH`)
`shape[layer]`	int »Page 163 (`PAD_SHAPE_...`)
`signal`	string »Page 164
`x, y`	int »Page 163 (center point, see Note)

Constants

`PAD_SHAPE_SQUARE`	square
`PAD_SHAPE_ROUND`	round
`PAD_SHAPE_OCTAGON`	octagon
`PAD_SHAPE_XLONGOCT`	xlongoct
`PAD_SHAPE_YLONGOCT`	ylongoct
`PAD_NAME_LENGTH`	max. recommended length of a pad name (same as `CONTACT_NAME_LENGTH`)

See also UL_PACKAGE »Page 189, UL_CONTACT »Page 175, UL_SMD »Page 202

Note

The coordinates (`x, y`) of the pad depend on the context in which it is called:

- if the pad is derived from a UL_LIBRARY context, the coordinates of the pad 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 »Page 148. If one of the layers »Page 185 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.

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

`device`	UL_DEVICE »Page 177
`name`	string »Page 164 (`PART_NAME_LENGTH`)
`value`	string »Page 164 (`PART_VALUE_LENGTH`)

Loop members

`instances()`	UL_INSTANCE »Page 183 (see Note)

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)

See also UL_BOARD »Page 171

Note

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);
  }
```

UL_PIN

Data members

`angle`	real »Page 163 (`0.0`...`359.9`°)
`contact`	UL_CONTACT »Page 175 (see Note)
`direction`	int »Page 163 (`PIN_DIRECTION_...`)
`function`	int »Page 163 (`PIN_FUNCTION_FLAG_...`)
`length`	int »Page 163 (`PIN_LENGTH_...`)
`name`	string »Page 164 (`PIN_NAME_LENGTH`)
`swaplevel`	int »Page 163
`visible`	int »Page 163 (`PIN_VISIBLE_FLAG_...`)
`x, y`	int »Page 163 (connection point)

Loop members

`circles()`	UL_CIRCLE »Page 173
`texts()`	UL_TEXT »Page 204
`wires()`	UL_WIRE »Page 206

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

See also UL_SYMBOL »Page 203, UL_PINREF »Page 194, UL_CONTACTREF »Page 176

Note

The `contact` data member returns the contact »Page 175 that has been assigned to the pin through a CONNECT »Page 48 command. It can be used as a boolean function to check whether a contact has been assigned to a pin (see example below).

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

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");
      }
    }
  L.devices(D) {
    D.gates(G) {
      G.symbol.pins(P) {
        if (!P.contact)
           printf("Unconnected pin: %s/%s/%s\n", D.name, G.name, P.name);
        }
      }
    }
  }
```

UL_PINREF

Data members

`instance`	UL_INSTANCE »Page 183
`part`	UL_PART »Page 191
`pin`	UL_PIN »Page 192

See also UL_SEGMENT »Page 199, UL_CONTACTREF »Page 176

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 »Page 163
`layer`	int »Page 163
`orphans`	int »Page 163 (0=off, 1=on)
`pour`	int »Page 163 (`POLYGON_POUR_...`)
`rank`	int »Page 163
`spacing`	int »Page 163
`thermals`	int »Page 163 (0=off, 1=on)
`width`	int »Page 163

Loop members

`contours()`	UL_WIRE »Page 206 (see Note)
`fillings()`	UL_WIRE »Page 206
`wires()`	UL_WIRE »Page 206

Constants

`POLYGON_POUR_SOLID`	solid
`POLYGON_POUR_HATCH`	hatch

See also UL_BOARD »Page 171, UL_PACKAGE »Page 189, UL_SHEET »Page 200, UL_SIGNAL »Page 201, UL_SYMBOL »Page 203

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 »Page 98 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.

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;
           }
        }
      }
    }
  }
```

UL_RECTANGLE

Data members

`layer`	int »Page 163
`x1, y1`	int »Page 163 (lower left corner)
`x2, y2`	int »Page 163 (upper right corner)

See also UL_BOARD »Page 171, UL_PACKAGE »Page 189, UL_SHEET »Page 200, UL_SYMBOL »Page 203

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

`grid`	UL_GRID »Page 181
`name`	string »Page 164

Loop members

`classes()`	UL_CLASS »Page 174
`layers()`	UL_LAYER »Page 185
`libraries()`	UL_LIBRARY »Page 187
`nets()`	UL_NET »Page 188
`parts()`	UL_PART »Page 191
`sheets()`	UL_SHEET »Page 200

See also UL_BOARD »Page 171, UL_LIBRARY »Page 187

Example

```
schematic(S) {
  S.parts(P) printf("Part: %s\n", P.name);
  }
```

UL_SEGMENT

Loop members

`junctions()`	UL_JUNCTION »Page 184 (see Note)
`pinrefs()`	UL_PINREF »Page 194 (see Note)
`texts()`	UL_TEXT »Page 204
`wires()`	UL_WIRE »Page 206

See also UL_BUS »Page 172, UL_NET »Page 188

Note

The `junctions()` and `pinrefs()` loop members are only available for net segments.

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 »Page 170
`number`	int »Page 163

Loop members

`arcs()`	UL_ARC »Page 169
`busses()`	UL_BUS »Page 172
`circles()`	UL_CIRCLE »Page 173
`nets()`	UL_NET »Page 188
`parts()`	UL_PART »Page 191
`polygons()`	UL_POLYGON »Page 195
`rectangles()`	UL_RECTANGLE »Page 197
`texts()`	UL_TEXT »Page 204
`wires()`	UL_WIRE »Page 206

See also UL_SCHEMATIC »Page 198

Example

```
schematic(SCH) {
  SCH.sheets(S) {
    printf("Sheet: %d\n", S.number);
    }
  }
```

UL_SIGNAL

Data members

`class`	UL_CLASS »Page 174
`name`	string »Page 164 (`SIGNAL_NAME_LENGTH`)

Loop members

`contactrefs()`	UL_CONTACTREF »Page 176
`polygons()`	UL_POLYGON »Page 195
`vias()`	UL_VIA »Page 205
`wires()`	UL_WIRE »Page 206

Constants

`SIGNAL_NAME_LENGTH`	max. recommended length of a signal name (used in formatted output only)

See also UL_BOARD »Page 171

Example

```
board(B) {
  B.signals(S) printf("Signal: %s\n", S.name);
  }
```

UL_SMD

Data members

`dx[layer], dy[layer]`	int »Page 163 (size)
`layer`	int »Page 163 (see Note)
`name`	string »Page 164 (`SMD_NAME_LENGTH`)
`roundness`	int »Page 163 (see Note)
`signal`	string »Page 164
`x, y`	int »Page 163 (center point, see Note)

Constants

`SMD_NAME_LENGTH`	max. recommended length of an smd name (same as `CONTACT_NAME_LENGTH`)

See also UL_PACKAGE »Page 189, UL_CONTACT »Page 175, UL_PAD »Page 190

Note

The coordinates (`x, y`), `layer` and `roundness` of the smd depend on the context in which it is called:

- if the smd is derived from a UL_LIBRARY context, the coordinates, 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 »Page 148. Valid layers »Page 185 are LAYER_TOP, LAYER_TSTOP and LAYER_TCREAM for a via in the Top layer, and LAYER_BOTTOM, LAYER_BSTOP and LAYER_BCREAM for a via in the Bottom layer, respectively.

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 »Page 170
`name`	string »Page 164 (`SYMBOL_NAME_LENGTH`)

Loop members

`arcs()`	UL_ARC »Page 169
`circles()`	UL_CIRCLE »Page 173
`rectangles()`	UL_RECTANGLE »Page 197
`pins()`	UL_PIN »Page 192
`polygons()`	UL_POLYGON »Page 195
`texts()`	UL_TEXT »Page 204
`wires()`	UL_WIRE »Page 206

Constants

`SYMBOL_NAME_LENGTH`	max. recommended length of a symbol name (used in formatted output only)

See also UL_GATE »Page 180, UL_LIBRARY »Page 187

Example

```
library(L) {
  L.symbols(S) printf("Sym: %s\n", S.name);
  }
```

UL_TEXT

Data members

angle	real »Page 163 (0.0...359.9°)
font	int »Page 163 (FONT_...)
layer	int »Page 163
mirror	int »Page 163
ratio	int »Page 163
size	int »Page 163
value	string »Page 164
x, y	int »Page 163 (origin point)

Loop members

wires()	UL_WIRE »Page 206 (see note)

Constants

FONT_VECTOR	vector font
FONT_PROPORTIONAL	proportional font
FONT_FIXED	fixed font

See also UL_BOARD »Page 171, UL_PACKAGE »Page 189, UL_SHEET »Page 200, UL_SYMBOL »Page 203

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`.

Example

```
board(B) {
  B.texts(T) {
    printf("Text: %s\n", T.value);
    }
  }
```

UL_VIA

Data members

`diameter[layer]`	int »Page 163
`drill`	int »Page 163
`shape[layer]`	int »Page 163 (`VIA_SHAPE_...`)
`x, y`	int »Page 163 (center point)

Constants

`VIA_SHAPE_SQUARE`	square
`VIA_SHAPE_ROUND`	round
`VIA_SHAPE_OCTAGON`	octagon

See also UL_SIGNAL »Page 201

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 »Page 148. If one of the layers »Page 185 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.

Example

```
board(B) {
  B.signals(S) {
    S.vias(V) {
      printf("Via: (%d %d)\n", V.x, V.y);
      }
    }
  }
```

UL_WIRE

Data members

`layer`	int »Page 163
`style`	int »Page 163 (`WIRE_STYLE_...`)
`width`	int »Page 163
`x1, y1`	int »Page 163 (starting point)
`x2, y2`	int »Page 163 (end point)

Loop members

`pieces()`	UL_WIRE »Page 206 (see note)

Constants

`WIRE_STYLE_CONTINUOUS`	continuous
`WIRE_STYLE_LONGDASH`	long dash
`WIRE_STYLE_SHORTDASH`	short dash
`WIRE_STYLE_DASHDOT`	dash dot

See also UL_BOARD »Page 171, UL_PACKAGE »Page 189, UL_SEGMENT »Page 199, UL_SHEET »Page 200, UL_SIGNAL »Page 201, UL_SYMBOL »Page 203

Note

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).

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 »Page 208
- Variable Definitions »Page 209
- Function Definitions »Page 210

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 »Page 218, 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 »Page 163 or object types »Page 166, `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 »Page 160 (`[]`), 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 »Page 164 arrays to have them sorted alphanumerically by the sort() »Page 247 function.

By default (if no `initializer` is present), data variables »Page 163 are set to `0` (or `""`, in case of a string), and object variables »Page 166 are "invalid".

Examples

`int i;`	defines an int »Page 163 variable named `i`
`string s = "Hello";`	defines a string »Page 164 variable named `s` and initializes it to `"Hello"`
`real a, b = 1.0, c;`	defines three real »Page 163 variables named `a`, `b` and `c`, initializing `b` to the value `1.0`
`int n[] = { 1, 2, 3 };`	defines an array of int »Page 163, initializing the first three elements to `1`, `2` and `3`
`numeric string names[];`	defines a string »Page 164 array that can be sorted alphanumerically
`UL_WIRE w;`	defines a UL_WIRE »Page 206 object named `w`

Function Definitions

You can write your own User Language functions and call them just like the Builtin Functions »Page 226.

The general syntax of a *function definition* is

```
type identifier(parameters)
{
  statements
}
```

where `type` is one of the data »Page 163 or object types »Page 166, `identifier` is the name of the function, `parameters` is a list of comma separated parameter definitions, and `statements` is a sequence of statements »Page 218.

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 »Page 217.

Execution of a function can be terminated by the `return »Page 222` statement. Without any `return` statement the function body is executed until it's closing brace (`}`).

A call to the `exit() »Page 244` 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 »Page 151 of the program.

Command line arguments are available to the program through the global Builtin Variables »Page 225 `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	`! »Page 212 ~ »Page 212 + - ++ -- »Page 213`
Multiplicative	`* / % »Page 213`
Additive	`+ - »Page 213`
Shift	`<< >> »Page 212`
Relational	`< <= > >= »Page 212`
Equality	`== != »Page 212`
Bitwise AND	`& »Page 212`
Bitwise XOR	`^ »Page 212`
Bitwise OR	`| »Page 212`
Logical AND	`&& »Page 212`
Logical OR	`|| »Page 212`
Conditional	`?: »Page 213`
Assignment	`= *= /= %= += -= »Page 213 &= ^= |= <<= >>= »Page 212`
Comma	`, »Page 213`

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 »Page 160.

Bitwise Operators

Bitwise operators work only with data types `char »Page 163` and `int »Page 163`.

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 »Page 215 of any data type.

Unary	
`!`	Logical NOT
Binary	
`&&`	Logical AND
`||`	Logical OR

Using a `string »Page 164` expression with a logical operator checks whether the string is empty.

Using an Object Type »Page 166 with a logical operator checks whether that object contains valid data.

Comparison Operators

Comparison operators work with expressions »Page 215 of any data type, except Object Types »Page 166.

`<`	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 »Page 215 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 »Page 163`, `int »Page 163` and `real »Page 163` (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 »Page 214

String Operators

String operators work with data types `char` »Page 163, `int` »Page 163 and `string` »Page 164. The left operand must always be of type `string` »Page 164.

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 »Page 213

Expressions

An *expression* can be one of the following:

- Arithmetic Expression »Page 215
- Assignment Expression »Page 215
- String Expression »Page 215
- Comma Expression »Page 216
- Conditional Expression »Page 216
- Function Call »Page 217

Expressions can be grouped using parentheses »Page 160, 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 »Page 213 or a bitwise operator »Page 212.

Examples

```
a + b
c++
m << 1
```

Assignment Expression

An *assignment expression* consists of a variable on the left side of an assignment operator »Page 213, 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 »Page 164 and char »Page 163 operands and a string operator »Page 214.

Examples

```
s + ".brd"
t + 'x'
```

Comma Expression

A *comma expression* is a sequence of expressions, delimited by the comma operator »Page 213

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 »Page 213 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 »Page 210 or a builtin function »Page 226. The formal parameters defined in the function definition »Page 210 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 »Page 218
- Control Statement »Page 218
- Expression Statement »Page 218
- Builtin Statement »Page 263
- Constant Definition »Page 208
- Variable Definition »Page 209

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 »Page 156 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 »Page 215 followed by a semicolon »Page 162.

An expression statement is executed by evaluating the expression. All side effects of this evaluation are completed before the next statement »Page 218 is executed. Most expression statements are assignments »Page 215 or function calls »Page 217.

A special case is the *empty statement*, consisting of only a semicolon »Page 162. 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 »Page 221
for »Page 221
while »Page 224
```

Selection statements are

```
if...else »Page 222
switch »Page 223
```

Jump statements are

```
break »Page 220
continue »Page 220
return »Page 222
```

break

The *break* statement has the general syntax

```
break;
```

and immediately terminates the **nearest** enclosing do...while »Page 221, for »Page 221, switch »Page 223 or while »Page 224 statement.

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 »Page 221, while »Page 224 , or for »Page 221 statement, or to the increment expression of the **nearest** enclosing for »Page 224 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 »Page 220` or `return »Page 222` 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 »Page 220` or `return »Page 222` 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 »Page 160 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 »Page 210 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 »Page 220 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 »Page 220` or `return »Page 222` 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 »Page 225
- Builtin Variables »Page 225
- Builtin Functions »Page 226
- Builtin Statements »Page 263

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 »Page 166 have their own **Constants** section which lists the builtin constants for that particular object (see e.g. UL_PIN »Page 192).

The following builtin constants are defined in addition to the ones listed for the various object types:

`EAGLE_VERSION`	EAGLE program version number (int »Page 163)
`EAGLE_RELEASE`	EAGLE program release number (int »Page 163)
`EAGLE_SIGNATURE`	a string »Page 164 containing EAGLE program name, version and copyright information
`REAL_EPSILON`	the minimum positive real »Page 163 number such that `r + REAL_EPSILON != r`
`REAL_MAX`	the largest possible real »Page 163 value
`REAL_MIN`	the smallest possible (positive!) real »Page 163 value the smallest representable number is `-REAL_MAX`
`INT_MAX`	the largest possible int »Page 163 value
`INT_MIN`	the smallest possible int »Page 163 value
`PI`	the value of "pi" (3.14..., real »Page 163)
`usage`	a string »Page 164 containing the text from the `#usage »Page 154` directive

Builtin Variables

Builtin variables are used to provide information at runtime.

`int argc`	number of arguments given to the RUN »Page 107 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 »Page 210 and use them to structure your User Language Program.

The builtin functions are grouped into the following categories:

- Character Functions »Page 228
- File Handling Functions »Page 231
- Mathematical Functions »Page 238
- Miscellaneous Functions »Page 243
- Printing Functions »Page 249
- String Functions »Page 254
- Time Functions »Page 261

Alphabetical reference of all builtin functions:

- abs() »Page 239
- acos() »Page 241
- asin() »Page 241
- atan() »Page 241
- ceil() »Page 240
- cos() »Page 241
- exit() »Page 244
- exp() »Page 242
- filedir() »Page 234
- fileerror() »Page 232
- fileext() »Page 234
- fileglob() »Page 233
- filename() »Page 234
- fileread() »Page 237
- filesetext() »Page 234
- filesize() »Page 235
- filetime() »Page 235
- floor() »Page 240
- frac() »Page 240
- isalnum() »Page 229
- isalpha() »Page 229
- iscntrl() »Page 229
- isdigit() »Page 229
- isgraph() »Page 229
- islower() »Page 229
- isprint() »Page 229
- ispunct() »Page 229
- isspace() »Page 229
- isupper() »Page 229
- isxdigit() »Page 229
- log() »Page 242
- log10() »Page 242
- lookup() »Page 245
- max() »Page 239
- min() »Page 239
- pow() »Page 242
- printf() »Page 250
- round() »Page 240
- sin() »Page 241
- sort() »Page 247
- sprintf() »Page 253

- sqrt() »Page 242
- strchr() »Page 255
- strjoin() »Page 255
- strlen() »Page 256
- strlwr() »Page 256
- strrchr() »Page 257
- strrstr() »Page 257
- strsplit() »Page 258
- strstr() »Page 258
- strsub() »Page 259
- strtod() »Page 259
- strtol() »Page 260
- strupr() »Page 260
- t2day() »Page 262
- t2dayofweek() »Page 262
- t2hour() »Page 262
- t2minute() »Page 262
- t2month() »Page 262
- t2second() »Page 262
- t2string() »Page 262
- t2year() »Page 262
- tan() »Page 241
- time() »Page 261
- tolower() »Page 230
- toupper() »Page 230
- trunc() »Page 240
- u2inch() »Page 248
- u2mic() »Page 248
- u2mil() »Page 248
- u2mm() »Page 248

Character Functions

Character functions are used to manipulate single characters.

The following character functions are available:

- isalnum() »Page 229
- isalpha() »Page 229
- iscntrl() »Page 229
- isdigit() »Page 229
- isgraph() »Page 229
- islower() »Page 229
- isprint() »Page 229
- ispunct() »Page 229
- isspace() »Page 229
- isupper() »Page 229
- isxdigit() »Page 229
- tolower() »Page 230
- toupper() »Page 230

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 »Page 260, strlwr »Page 256

File Handling Functions

Filename handling functions are used to work with file names, sizes and timestamps.

The following file handling functions are available:

- fileerror() »Page 232
- fileglob() »Page 233
- filedir() »Page 234
- fileext() »Page 234
- filename() »Page 234
- fileread() »Page 237
- filesetext() »Page 234
- filesize() »Page 235
- filetime() »Page 235

See output() »Page 267 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.

Description

`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.

See also output »Page 267, printf »Page 250, fileread »Page 237

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`.

Description

`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.

See also dlgFileOpen() »Page 273, dlgFileSave() »Page 273

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 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).
`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 »Page 235

If a filename has been split into its three separate parts, these parts can be put together again with the `+ operator »Page 214` to form the original full filename.

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 »Page 261.

See also time »Page 261, Filename Functions »Page 234

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() »Page 237

See output() »Page 267 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 »Page 245, strsplit »Page 258, fileerror »Page 232

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() »Page 239
- acos() »Page 241
- asin() »Page 241
- atan() »Page 241
- ceil() »Page 240
- cos() »Page 241
- exp() »Page 242
- floor() »Page 240
- frac() »Page 240
- log() »Page 242
- log10() »Page 242
- max() »Page 239
- min() »Page 239
- pow() »Page 242
- round() »Page 240
- sin() »Page 241
- sqrt() »Page 242
- trunc() »Page 240
- tan() »Page 241

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 »Page 163`, `int »Page 163` or `real »Page 163`.

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...)

Example

```
real x = 2.0;
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`.

Note

The "n-th" root can be calculated using the `pow` function with a negative exponent.

Example

```
real x = 2.1;
printf("The square root of %f is %f\n", x, sqrt(x));
```

Miscellaneous Functions

Miscellaneous functions are used to perform various tasks.

The following miscellaneous functions are available:

- exit() »Page 244
- lookup() »Page 245
- sort() »Page 247
- Unit Conversions »Page 248

exit()

Function

Exits from a User Language Program.

Syntax

```
void exit(int result);
void exit(string command);
```

Constants

`EXIT_SUCCESS`	return value for successful program execution (value `0`)
`EXIT_FAILURE`	return value for failed program execution (value `-1`)

See also RUN »Page 107

Description

The `exit` function terminates execution of a User Language Program.
If an integer `result` is given it will be used as the return value »Page 151 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`.

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 »Page 237, strsplit »Page 258

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');
   }
```

sort()

Function

Sorts an array or a set of arrays.

Syntax

```
void sort(int number, array1[, array2,...]);
```

Description

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 »Page 163` 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.

Unit Conversions

Function

Converts internal units.

Syntax

```
real u2inch(int n);
real u2mic(int n);
real u2mil(int n);
real u2mm(int 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*.

See also UL_GRID »Page 181

EAGLE stores all coordinate and size values as `int »Page 163` values with a resolution of 1/10000mm (0.1µ). The above unit conversion functions can be used to convert these internal units to the desired measurement units.

Example

```
board(B) {
  B.elements(E) {
    printf("%s at (%f, %f)\n", E.name,
           u2mm(E.x), u2mm(E.y));
    }
  }
```

Printing Functions

Printing functions are used to print formatted strings.

The following printing functions are available:

- printf() »Page 250
- sprintf() »Page 253

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 »Page 267 statement.

In case of an error, `printf` returns `-1`.

See also sprintf »Page 253, output »Page 267, fileerror »Page 232

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 next argument in the call (which must be an `int`) specifies 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 zeroes.
`*`	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 next argument in the call (which must be an `int`) specifies 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 »Page 253, 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 »Page 250

Format string

See printf »Page 250.

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 »Page 250 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() »Page 255
- strjoin() »Page 255
- strlen() »Page 256
- strlwr() »Page 256
- strrchr() »Page 257
- strrstr() »Page 257
- strsplit() »Page 258
- strstr() »Page 258
- strsub() »Page 259
- strtod() »Page 259
- strtol() »Page 260
- strupr() »Page 260

strchr()

Function

Scans a string for the first occurrence of a given character.

Syntax

```
int strchr(string s, char c);
```

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 »Page 257, strstr »Page 258

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`.

Description

`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() »Page 237 function and split »Page 258 into an array of N strings to be joined to the original string as read from the file.

See also strsplit »Page 258, lookup »Page 245, fileread »Page 237

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 »Page 260, tolower »Page 230

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);
```

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 »Page 255, strrstr »Page 257

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);
```

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 »Page 258, strrchr »Page 257

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`.

Description

`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() »Page 237 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.

See also strjoin »Page 255, lookup »Page 245, fileread »Page 237

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);
```

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 »Page 257, strchr »Page 255

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 »Page 158. If an error occurs during conversion of the string `0.0` will be returned.

See also strtol »Page 260

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 »Page 157. If an error occurs during conversion of the string `0` will be returned.

See also strtod »Page 259

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 »Page 256, toupper »Page 230

Example

```
string s = "This Is A String";
string r = strupr(s);
printf("Prior to strupr: %s - after strupr: %s\n", s, r);
```

Time Functions

Time functions are used to get and process time and date information.

The following time functions are available:

- t2day() »Page 262
- t2dayofweek() »Page 262
- t2hour() »Page 262
- t2minute() »Page 262
- t2month() »Page 262
- t2second() »Page 262
- t2string() »Page 262
- t2year() »Page 262
- time() »Page 261

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 »Page 262, filetime »Page 235

Example

```
int CurrentTime = time();
```

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);
```

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 »Page 261

Example

```
int t = time();
printf("It is now %02d:%02d:%02d\n",
       t2hour(t), t2minute(t), t2second(t));
```

Builtin Statements

Builtin statements are generally used to open a certain context in which data structures of 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() »Page 264
- deviceset() »Page 265
- library() »Page 266
- output() »Page 267
- package() »Page 268
- schematic() »Page 269
- sheet() »Page 270
- symbol() »Page 271

board()

Function

Opens a board context.

Syntax

```
board(identifier) statement
```

Description

The `board` statement opens a board context if the current editor window contains a board drawing. A variable of type UL_BOARD »Page 171 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.

See also schematic »Page 269, library »Page 266

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
```

Description

The `deviceset` statement opens a device set context if the current editor window contains a device drawing. A variable of type UL_DEVICESET »Page 178 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.

See also package »Page 268, symbol »Page 271, library »Page 266

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
```

Description

The `library` statement opens a library context if the current editor window contains a library drawing. A variable of type UL_LIBRARY »Page 187 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.

See also board »Page 264, schematic »Page 269, deviceset »Page 265, package »Page 268, symbol »Page 271

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
```

Description

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.

See also printf »Page 250, fileerror »Page 232

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

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`

Description

The `package` statement opens a package context if the current editor window contains a package drawing. A variable of type UL_PACKAGE »Page 189 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.

See also library »Page 266, deviceset »Page 265, symbol »Page 271

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
```

Description

The `schematic` statement opens a schematic context if the current editor window contains a schematic drawing. A variable of type UL_SCHEMATIC »Page 198 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.

See also board »Page 264, library »Page 266, sheet »Page 270

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 »Page 270` 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`

Description

The `sheet` statement opens a sheet context if the current editor window contains a sheet drawing. A variable of type UL_SHEET »Page 200 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.

See also schematic »Page 269

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.parts(P)
       printf("Part: %s\n", P.name);
     }
```

symbol()

Function

Opens a symbol context.

Syntax

`symbol(identifier) statement`

Description

The `symbol` statement opens a symbol context if the current editor window contains a symbol drawing. A variable of type UL_SYMBOL »Page 203 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.

See also library »Page 266, deviceset »Page 265, package »Page 268

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 »Page 272	describes the ready to use standard dialogs
Dialog Objects »Page 275	defines the objects that can be used in a dialog
Layout Information »Page 299	explains how to define the location of objects within a dialog
Dialog Functions »Page 300	describes special functions for use with dialogs
A Complete Example »Page 306	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() »Page 273
- dlgFileOpen() »Page 273
- dlgFileSave() »Page 273
- dlgMessageBox() »Page 274

See Dialog Objects »Page 275 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 cancelled the dialog, the result will be an empty string.

Description

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`.

See also dlgFileOpen »Page 273

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 cancelled the dialog, the result will be an empty string.

Description

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.

See also dlgDirectory »Page 273

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`.

Description

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`.

`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 »Page 305.

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 »Page 305.

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*:

dlgDialog »Page 279	the basic container of any dialog
dlgGridLayout »Page 280	a grid based layout context
dlgCell »Page 276	a grid cell context
dlgHBoxLayout »Page 282	a horizontal box layout context
dlgVBoxLayout »Page 298	a vertical box layout context
dlgLabel »Page 284	a text label
dlgSpacing »Page 290	a layout spacing object
dlgStretch »Page 292	a layout stretch object
dlgSpinBox »Page 291	a spin box selection field
dlgIntEdit »Page 283	an integer entry field
dlgComboBox »Page 278	a combo box selection field
dlgGroup »Page 281	a group field
dlgTabWidget »Page 295	a tab page container
dlgTabPage »Page 294	a tab page
dlgListBox »Page 285	a list box
dlgListView »Page 286	a list view
dlgStringEdit »Page 293	a string entry field
dlgTextEdit »Page 296	a text entry field
dlgTextView »Page 297	a text viewer field
dlgRealEdit »Page 289	a real entry field
dlgCheckBox »Page 277	a checkbox
dlgRadioButton »Page 288	a radio button
dlgPushButton »Page 287	a push button

dlgCell

Function

Defines a cell location within a grid layout context.

Syntax

`dlgCell(int row, int column[, int row2, int column2])` *`statement`*

Description

The `dlgCell` statement defines the location of a cell within a grid layout context »Page 280.

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 »Page 282, so if the cell contains more than one dialog object, they will be placed next to each other horizontally.

See also dlgGridLayout »Page 280, dlgHBoxLayout »Page 282, dlgVBoxLayout »Page 298, Layout Information »Page 299, A Complete Example »Page 306

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`* `]`

Description

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 »Page 305.

`dlgCheckBox` is mainly used within a dlgGroup »Page 281, 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.

See also dlgRadioButton »Page 288, dlgGroup »Page 281, Layout Information »Page 299, A Complete Example »Page 306

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`*` ]`

Description

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`.

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`.

See also dlgListBox »Page 285, dlgLabel »Page 284, Layout Information »Page 299, A Complete Example »Page 306

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() »Page 301` function.
If the dialog is simply closed, the return value will be `0`.

Description

The `dlgDialog` function executes the dialog defined by `block »Page 218`. 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 »Page 275, 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 »Page 298, so a simple dialog doesn't have to worry about the layout.

A `dlgDialog` should at some point contain a call to the `dlgAccept() »Page 301` 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 »Page 272 instead.

See also dlgGridLayout »Page 280, dlgHBoxLayout »Page 282, dlgVBoxLayout »Page 298, dlgAccept »Page 301, dlgReset »Page 303, dlgReject »Page 304, A Complete Example »Page 306

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`*

Description

The `dlgGridLayout` statement opens a grid layout context.

The only dialog object that can be used directly in `statement` is dlgCell »Page 276, 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.

See also dlgCell »Page 276, dlgHBoxLayout »Page 282, dlgVBoxLayout »Page 298, Layout Information »Page 299, A Complete Example »Page 306

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`*

Description

The `dlgGroup` statement defines a group with the given `Title`.

By default a `dlgGroup` contains a dlgVBoxLayout »Page 298, so a simple group doesn't have to worry about the layout.

`dlgGroup` is mainly used to contain a set of radio buttons »Page 288 or check boxes »Page 277, but may as well contain any other objects in its `statement`.
Radio buttons within a `dlgGroup` are numbered starting with `0`.

See also dlgCheckBox »Page 277, dlgRadioButton »Page 288, Layout Information »Page 299, A Complete Example »Page 306

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`*

Description

The `dlgHBoxLayout` statement opens a horizontal box layout context for the given `statement`.

See also dlgGridLayout »Page 280, dlgVBoxLayout »Page 298, Layout Information »Page 299, A Complete Example »Page 306

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)
```

Description

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.

See also dlgRealEdit »Page 289, dlgStringEdit »Page 293, dlgLabel »Page 284, Layout Information »Page 299, A Complete Example »Page 306

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])`

Description

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 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 »Page 287, 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 `'&'`, 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 »Page 305.

See also Layout Information »Page 299, A Complete Example »Page 306

Example

```
string Name = "Linus";
dlgHBoxLayout {
  dlgLabel("Enter &Name");
  dlgStringEdit(Name, 1);
  }
```

dlgListBox

Function

Defines a list box selection field.

Syntax

`dlgListBox(string array[], int &Selected) [ statement ]`

Description

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`. 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.

See also dlgComboBox »Page 278, dlgListView »Page 286, dlgLabel »Page 284, Layout Information »Page 299, A Complete Example »Page 306

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) [ *statement* ]`

Description

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`.

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).

The optional `statement` is executed whenever the user double clicks on an entry of the `dlgListView`. 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.

See also dlgListBox »Page 285, dlgLabel »Page 284, Layout Information »Page 299, A Complete Example »Page 306

Example

```
string Colors[] = { "red\tThe color RED", "green\tThe color GREEN", "blue\tThe
color BLUE" };
int Selected = 0; // initially selects "red"
dlgListView("Name\tDescription", Data, Selected) dlgMessageBox("You have
selected " + Colors[Selected]);
```

dlgPushButton

Function

Defines a push button.

Syntax

`dlgPushButton(string Text)` *`statement`*

Description

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 »Page 305.

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() »Page 304! 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 »Page 305.

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.

See also Layout Information »Page 299, Dialog Functions »Page 300, A Complete Example »Page 306

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`*` ]`

Description

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 »Page 305.

`dlgRadioButton` can only be used within a dlgGroup »Page 281.
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.

See also dlgCheckBox »Page 277, dlgGroup »Page 281, Layout Information »Page 299, A Complete Example »Page 306

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)`

Description

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.

See also dlgIntEdit »Page 283, dlgStringEdit »Page 293, dlgLabel »Page 284, Layout Information »Page 299, A Complete Example »Page 306

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)
```

Description

The `dlgSpacing` statement defines additional space in a vertical or horizontal box layout context.

`Size` defines the number of pixels of the additional space.

See also dlgHBoxLayout »Page 282, dlgVBoxLayout »Page 298, dlgStretch »Page 292, Layout Information »Page 299, A Complete Example »Page 306

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)
```

Description

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.

See also dlgIntEdit »Page 283, dlgLabel »Page 284, Layout Information »Page 299, A Complete Example »Page 306

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)
```

Description

The `dlgStretch` statement defines an empty stretchable space in a vertical or horizontal box layout context.

`Factor` defines the stretch factor of the space.

See also dlgHBoxLayout »Page 282, dlgVBoxLayout »Page 298, dlgSpacing »Page 290, Layout Information »Page 299, A Complete Example »Page 306

Example

```
dlgHBoxLayout {
  dlgStretch(1);
  dlgPushButton("+OK")    { dlgAccept(); };
  dlgPushButton("Cancel") { dlgReject(); };
  }
```

dlgStringEdit

Function

Defines a string entry field.

Syntax

```
dlgStringEdit(string &Text)
```

Description

The `dlgStringEdit` statement defines a text entry field with the given `Text`.

See also dlgRealEdit »Page 289, dlgIntEdit »Page 283, dlgTextEdit »Page 296, dlgLabel »Page 284, Layout Information »Page 299, A Complete Example »Page 306

Example

```
string Name = "Linus";
dlgHBoxLayout {
  dlgLabel("Enter &Name");
  dlgStringEdit(Name);
  }
```

dlgTabPage

Function

Defines a tab page.

Syntax

`dlgTabPage(string Title)` *`statement`*

Description

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 »Page 305.

Tab pages can only be used within a dlgTabWidget »Page 295.

By default a `dlgTabPage` contains a dlgVBoxLayout »Page 298, so a simple tab page doesn't have to worry about the layout.

See also dlgTabWidget »Page 295, Layout Information »Page 299, A Complete Example »Page 306

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`*

Description

The `dlgTabWidget` statement defines a container for a set of tab pages.

`statement` must be a sequence of one or more dlgTabPage »Page 294 objects. There must be no other dialog objects in this sequence.

See also dlgTabPage »Page 294, Layout Information »Page 299, A Complete Example »Page 306

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)`

Description

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.

See also dlgStringEdit »Page 293, dlgTextView »Page 297, dlgLabel »Page 284, Layout Information »Page 299, A Complete Example »Page 306

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)`

Description

The `dlgTextView` statement defines a multiline text viewer field with the given `Text`.

The `Text` may contain Rich Text »Page 307 tags.

See also dlgTextEdit »Page 296, dlgLabel »Page 284, Layout Information »Page 299, A Complete Example »Page 306

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`*

Description

The `dlgVBoxLayout` statement opens a vertical box layout context for the given `statement`.

By default a dlgDialog »Page 279 contains a `dlgVBoxLayout`, so a simple dialog doesn't have to worry about the layout.

See also dlgGridLayout »Page 280, dlgHBoxLayout »Page 282, Layout Information »Page 299, A Complete Example »Page 306

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 »Page 280, horizontal »Page 282 or vertical »Page 298.

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 »Page 292 and dlgSpacing »Page 290 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 »Page 306 for a demonstration of this.

Dialog Functions

The following functions can be used with User Language Dialogs:

dlgAccept() »Page 301	closes the dialog and accepts its contents
dlgRedisplay() »Page 302	immediately redisplays the dialog after changes to any values
dlgReset() »Page 303	resets all dialog objects to their initial values
dlgReject() »Page 304	closes the dialog and rejects its contents

dlgAccept()

Function

Closes the dialog and accepts its contents.

Syntax

```
void dlgAccept([ int Result ]);
```

Description

The `dlgAccept` function causes the dlgDialog »Page 279 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 »Page 275 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!
See also dlgReject »Page 304, dlgDialog »Page 279, A Complete Example »Page 306

Example

```
int Result = dlgDialog("Test") {
               dlgPushButton("+OK")    dlgAccept(42);
               dlgPushButton("Cancel") dlgReject();
               };
```

dlgRedisplay()

Function

Redisplays the dialog after changing values.

Syntax

```
void dlgRedisplay(void);
```

Description

The `dlgRedisplay` function can be called to immediately refresh the dlgDialog »Page 279 after changes have been made to the variables used when defining the dialog objects »Page 275.

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.

See also dlgReset »Page 303, dlgDialog »Page 279, A Complete Example »Page 306

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);
```

Description

The `dlgReset` function copies the initial values back into all dialog objects »Page 275 of the current dlgDialog »Page 279.

Any changes the user has made to the dialog values will be discarded.

Calling `dlgReject() »Page 304` implies a call to `dlgReset()`.

See also dlgReject »Page 304, dlgDialog »Page 279, A Complete Example »Page 306

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 ]);
```

Description

The `dlgReject` function causes the dlgDialog »Page 279 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 »Page 275 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() »Page 303`.
See also dlgAccept »Page 301, dlgReset »Page 303, dlgDialog »Page 279, A Complete Example »Page 306

Example

```
int Result = dlgDialog("Test") {
               dlgPushButton("+OK")     dlgAccept(42);
               dlgPushButton("Cancel") dlgReject();
               };
```

Escape Character

The characters `'&'`, `'+'` and `'-'` 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();
    }
  };
```

Rich Text

Rich Text is a subset of the tags used to format HTML pages. It can be used to format the text of several User Language Dialog »Page 272 objects, in the `#usage »Page 154` directive or in the description »Page 54 of library objects.

Text is considered to be Rich Text 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 `<qt>...</qt>` tag.

The following table lists all supported Rich Text tags and their available attributes:

Tag	**Description**
<qt>...</qt>	A rich text document. It understands the following attributes

- `bgcolor` - The background color, for example `bgcolor="yellow"` or `bgcolor="#0000FF"`.
- `background` - The background pixmap, for example `background="granit.xpm"`.
- `text` - The default text color, for example `text="red"`.
- `link` - The link color, for example `link="green"`.

<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 junks 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.qml">...</a>`. You can also specify an additional anchor within the specified target document, for example `<a href="target.qml#123">...</a>`.
- `name` - The anchor name, as in `<a name="target">...</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 junks 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="qt.xpm"/>`.
- `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.
<br> 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	**Description**
&lt;	<
&gt;	>
&amp;	&
&nbsp;	
&bull;	&bull;
&aring;	&aring;
&oslash;	&oslash;
&ouml;	&ouml;
&auml;	&auml;
&uuml;	&uuml;
&Ouml;	&Ouml;
&Auml;	&Auml;
&Uuml;	&Uuml;
&szlig;	&szlig;
&copy;	&copy;
&deg;	°
&micro;	µ
&plusmn;	±
&middot;	&middot;

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 with the operating system command

```
DEL *.?#?
```

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 »Page 24.

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 »Page 24.

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 that has a supply layer, the Thermal/Annulus display may be incorrect. In such a case a window refresh will show the correct Thermal/Annulus symbols. The same applies to Undo/Redo operations that involve pads connected to supply layers.
- 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 »Page 59 (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 text window and Forward&Back Annotation will **not** be activated.

The ERC protocol file contains several sections that list the inconsistencies found. Each of these sections may or may not be present, depending on which inconsistencies have been found. **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 reduce the number of error messages you get in the next ERC run.**

```
Parts not found in board:
  IC1
  R7
```

This section lists the names of the *parts* that are present in the schematic, but are missing in the board.

```
Elements not found in schematic:
  C33
  D2
```

This section lists the names of the *elements* that are present in the board, but are missing in the schematic.

The following sections are only present if the part and element names have checked out to be consistent:

```
Parts/Elements with inconsistent packages:
  IC12
  R1
```

This section lists the names of the *parts/elements* that are present on both schematic and board, but which have inconsistent packages. Packages are considered consistent if they contain an equally named set of pads/smds.

```
Parts/Elements with inconsistent values:
  R55      100k       47k
  C99      10n        10p
```

This section lists the names of the *parts/elements* that are present on both schematic and board, but which have different values. The second column lists the value of the *part* in the schematic, the third column lists the value of the *element* in the board.

The following sections are only present if the part and element packages have checked out to be consistent:

```
Pins/Pads with different connections:
  Part   Gate   Pin   Net   Pad   Signal
  IC5    A      2     GND   2     S$42
  R3     R      1     D1    1     D2
```

This section lists the names of the *pins* and *pads* that are connected to different nets/signals in schematic and board. The *Net* column contains the net name in the schematic, while the *Signal* column gives the signal name in the board. If either of these two entries is blank, this means that the pin/pad is not connected.

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 file. This can be done by applying editor commands like NAME »Page 81, VALUE »Page 123, PINSWAP »Page 92, REPLACE »Page 103 etc. After fixing the inconsistencies you must use the ERC »Page 59 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, Inc.
801 South Federal Highway
Delray Beach, FL 33483-5185
USA

Phone	(561)274-8355
Fax	(561)274-8218
Email	Support@CadSoftUSA.COM
URL	www.CadSoftUSA.COM

License

To legally use EAGLE you need a registered user license. Please check your "User License Certificate" whether it contains a holographic sticker with the EAGLE logo as well as a label with your name, address, serial number and installation code. If you have any doubts about the validity or authenticity of your license, please contact our Technical Support »Page 314 staff for verification.

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 run 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.

3-User License

Up to THREE users may use the program simultaneously, on three different computers. The only restriction that applies is that these computers must all belong to the license holder, and that they be located in the same building.

5-User License

Same as "3-User License", but up to FIVE users may use the program simultaneously.

Network License

The program may be installed on ONE LAN Server, and an UNLIMITED number of users who are logged in to that LAN Server may use the program simultaneously.

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.

Product Registration

Before you can work with EAGLE it is necessary to register the program with your personalized license data.

Please make sure your EAGLE license disk with your serial number on it is inserted in your disk drive.

Next you need to enter your `Installation Code` as printed on the label on your User License Certificate. This code consists of 10 lowercase characters and has to be entered exactly as printed on the label.

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 »Page 314 staff for assistance.

Installing additional modules

If you decided to update your license with the schematic/autorouter module you get a new User License Certificate with a new personal 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 »Page 20 in the Help menu the item Product Registration.

EAGLE Editions

EAGLE is available in three different editions to fit various user requirements.

Professional

The *Professional* edition has no limitations.

Standard

The *Standard* edition has the following limitations:

- board area limited to 160x100mm (6.3x4inch), which corresponds to a full Eurocard
- only four routing layers (Top, Route2, Route15 and Bottom)

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

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.

Both the *Standard* and *Light* edition of EAGLE can be used to view files created with the *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/Product information" from the Control Panel's menu.

Index

—A—

A Complete Example 306
Aperture Emulation 139
ASSIGN 38
Assignment Expression 215
Autorouter 146

—B—

Backup 24
Brackets 160
break 220
Builtin Constants 225
Builtin Statements 263
BUS 43

—C—

CAM Processor 134
Character Constants 157
Character Functions 228
Checking the Layout 18
Checking the Schematic 17
CLASS 46
Comma 162
Command Syntax 32
Comparison Operators 212
Compound Statement 218
Conditional Expression 216
Constant Definitions 208
Context Menus 22
Control Panel 20
Control Panel and Editor Windows 16
COPY 50
Creating a Library Device 19

—D—

DESCRIPTION 54
Dialog Functions 300
Dialogs 272
Directives 154
Directories 24
dlgCell 276
dlgDialog 279
dlgFileOpen(), dlgFileSave() 273
dlgHBoxLayout 282
dlgListBox 285
dlgRadioButton 288
dlgReset() 303
dlgSpinBox 291
dlgTabPage 294
dlgTextView 297
Drawing a Schematic 17
Drill Tolerances 139

—E—

EAGLE Editions 316
EDIT 58
Edit Library Object 28
Editor Windows 28
Entering Parameters and Values 16
Equal Sign 162
EXPORT 61

—F—

File Handling Functions 231
Filename Functions 234
fileread() 237
Flag Options 143
for 221
Forward&Back Annotation 310

—G—

General Help 13
Generating a Board from a Schematic 18
GROUP 66

—I—

INFO 69
Installation 14
int 163

—K—

Keyboard and Mouse 26
Keywords 155

—L—

LABEL 72
Library Editor 28
library() 266

—M—

MENU 77

Miscellaneous Functions 243

—N—

NAME 81

—O—

Operators 211
OPTIMIZE 84
Output Device 137

—P—

PASTE 88
Pen Data 141
POLYGON 93
printf() 250
Printing a Drawing 131

—Q—

Quick Introduction 16
QUIT 97

—R—

REDO 100
REPLACE 103
Rounding Functions 240
ROUTE 106

—S—

Schematic Editor 29
schematic() 269
Selecting objects in dense areas 27
SET 109
SMASH 114
sort() 247
strchr() 255
String Constants 158
String Operators 214
strlwr() 256
strsplit() 258
strtod() 259
switch 223
Syntax 152

—T—

Technical Support 314
TECHNOLOGY 117
Time Functions 261
Type Conversions 165

—U—

UL_ARC 169
UL_BUS 172
UL_CONTACT 175
UL_DEVICESET 178
UL_GRID 181
UL_JUNCTION 184
UL_NET 188
UL_PART 191
UL_POLYGON 195
UL_SEGMENT 199
UL_SMD 202
UL_VIA 205
UPDATE 121
User Interface 25
User Language 150

—V—

VIA 124

—W—

WRITE 129