EAGLE

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

User Language

Version 5.12

Copyright © 2011 CadSoft *All rights reserved*

Index

User Language

The EAGLE User Language can be used to access the EAGLE data structures and to create a wide variety of output files.

To use this feature you have to write a User Language Program (ULP), and then execute it.

The following sections describe the EAGLE User Language in detail:

Syntax	lists the rules a ULP file has to follow
Data Types	defines the basic data types
Object Types	defines the EAGLE objects
Definitions	shows how to write a definition
Operators	lists the valid operators
Expressions	shows how to write expressions
Statements	defines the valid statements
Builtins	lists the builtin constants, functions etc.
Dialogs	shows how to implement a graphical frontent to a ULP

Writing a ULP

A User Language Program is a plain text file which is written in a C-like syntax. User Language Programs use the extension `.ulp`. You can create a ULP file with any text editor (provided it does not insert any additional control characters into the file) or you can use the builtin text editor.

A User Language Program consists of two major items, definitions and statements.

Definitions are used to define constants, variables and functions to be used by statements.

A simple ULP could look like this:

```
#usage "Add the characters in the word 'Hello'\n"
       "Usage: RUN sample.ulp"
// Definitions:
string hello = "Hello";
int count(string s)
{
  int c = 0;
  for (int i = 0; s[i]; ++i)
      c += s[i];
  return c;
}
// Statements:
output("sample") {
  printf("Count is: %d\n", count(hello));
  }
```

If the `#usage` directive is present, its value will be used in the Control Panel to display a description of the program.

If the result of the ULP shall be a specific command that shall be executed in the editor window, the `exit()` function can be used to send that command to the editor window.

Executing a ULP

User Language Programs are executed by the RUN command from an editor window's command line.

A ULP can return information on whether it has run successfully or not. You can use the `exit()` function to terminate the program and set the return value.

A return value of `0` means the ULP has ended "normally" (i.e. successfully), while any other value is considered as an abnormal program termination.

The default return value of any ULP is `0`.

When the RUN command is executed as part of a script file, the script is terminated if the ULP has exited with a return value other than `0`.

A special variant of the `exit()` function can be used to send a command to the editor window as a result of the ULP.

Syntax

The basic building blocks of a User Language Program are

- Whitespace
- Comments
- Directives
- Keywords
- Identifiers
- Constants
- Punctuators

All of these have to follow certain syntactical rules, which are described in their respective sections.

Whitespace

Before a User Language Program can be executed, it has to be read in from a file. During this read in process, the file contents is *parsed* into tokens and *whitespace*.

Any spaces (blanks), tabs, newline characters and comments are considered *whitespace* and are discarded.

The only place where ASCII characters representing *whitespace* are not discarded is within literal strings, like in

```
string s = "Hello World";
```

where the blank character between `'o'` and `'W'` remains part of the string.

If the final newline character of a line is preceded by a backslash (`\`), the backslash and newline character are both discarded, and the two lines are treated as one line:

```
"Hello \
World"
```

is parsed as "Hello World"

Comments

When writing a User Language Program it is good practice to add some descriptive text, giving the reader an idea about what this particular ULP does. You might also want to add your name (and, if available, your email address) to the ULP file, so that other people who use your program could contact you in case they have a problem or would like to suggest an improvement.

There are two ways to define a comment. The first one uses the syntax

```
/* some comment text */
```

which marks any characters between (and including) the opening /* and the closing */ as comment. Such comments may expand over more than one lines, as in

```
/* This is a
   multi line comment
*/
```

but they do not nest. The first */ that follows any /* will end the comment.

The second way to define a comment uses the syntax

```
int i; // some comment text
```

which marks any characters after (and including) the // and up to (but not including) the newline character at the end of the line as comment.

Directives

The following *directives* are available:

#include
#require
#usage

#include

A User Language Program can reuse code in other ULP files through the #include directive. The syntax is

#include "*filename*"

The file filename is first looked for in the same directory as the current source file (that is the file that contains the #include directive). If it is not found there, it is searched for in the directories contained in the ULP directory path.

The maximum include depth is 10.

Each #include directive is processed only **once**. This makes sure that there are no multiple definitions of the same variables or functions, which would cause errors.

Portability note

If *filename* contains a directory path, it is best to always use the **forward slash** as directory separator (even under Windows!). Windows drive letters should be avoided. This way a User Language Program will run on all platforms.

#require

Over time it may happen that newer versions of EAGLE implement new or modified User Language features, which can cause error messages when such a ULP is run from an older version of EAGLE. In order to give the user a dedicated message that this ULP requires at least a certain version of EAGLE, a ULP can contain the `#require` directive. The syntax is

```
#require version
```

The *version* must be given as a real constant of the form

```
V.RRrr
```

where `V` is the version number, `RR` is the release number and `rr` is the (optional) revision number (both padded with leading zeros if they are less than 10). For example, if a ULP requires at least EAGLE version 4.11r06 (which is the beta version that first implemented the `#require` directive), it could use

```
#require 4.1106
```

The proper directive for version 5.1.2 would be

```
#require 5.0102
```

#usage

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

```
#usage text [, text...]
```

implements a standard way to make this information available.

If the `#usage` directive is present, its `text` (which has to be a string constant) will be used in the Control Panel to display a description of the program.

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

Only the `#usage` directive of the main program file (that is the one started with the RUN command) will take effect. Therefore pure include files can (and should!) also have `#usage` directives of their own.

It is best to have the `#usage` directive at the beginning of the file, so that the Control Panel doesn't have to parse all the rest of the text when looking for the information to display.

If the usage information shall be made available in several langauges, the texts of the individual languages have to be separated by commas. Each of these texts has to start with the two letter code of the respective language (as delivered by the language() function), followed by a colon and any number of blanks. If no suitable text is found for the language used on the actual system, the first given text will be used (this one should generally be English in order to make the program accessible to the largest number of users).

Example

```
#usage "en: A sample ULP\n"
           "Implements an example that shows how to use the EAGLE User
Language\n"
           "Usage: RUN sample.ulp\n"
           "Author: john@home.org",
       "de: Beispiel eines ULPs\n"
           "Implementiert ein Beispiel das zeigt, wie man die EAGLE User
Language benutzt\n"
           "Aufruf: RUN sample.ulp\n"
           "Author: john@home.org"
```

Keywords

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

break
case
char
continue
default
do
else
enum
for
if
int
numeric
real
return
string
switch
void
while

In addition, the names of builtins and object types are also reserved and must not be used as identifier names.

Identifiers

An *identifier* is a name that is used to introduce a user defined constant, variable or function.

Identifiers consist of a sequence of letters (`a b c`..., `A B C`...), digits (`1 2 3`...) and underscores (`_`). The first character of an identifier **must** be a letter or an underscore.

Identifiers are case-sensitive, which means that

```
int Number, number;
```

would define two **different** integer variables.

The maximum length of an identifier is 100 characters, and all of these are significant.

Constants

Constants are literal data items written into a User Language Program. According to the different data types, there are also different types of constants.

- Character constants
- Integer constants
- Real constants
- String constants

Character Constants

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

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

The type of a character constant is `char`.

Integer Constants

Depending on the first (and possibly the second) character, an *integer constant* is assumed to be expressed in different base values:

first	second	constant interpreted as
`0`	`1-7`	octal (base 8)
`0`	`x,X`	hexadecimal (base 16)
`1-9`		decimal (base 10)

The type of an integer constant is `int`.

Examples

`16`	decimal
`020`	octal
`0x10`	hexadecimal

Real Constants

A *real constant* follows the general pattern

`[-]`*`int`*`.`*`frac`*`[e|E[±]`*`exp`*`]`

which stands for

- optional sign
- decimal integer
- decimal point
- decimal fraction
- `e` or `E` and a signed integer exponent

You can omit either the decimal integer or the decimal fraction (but not both). You can omit either the decimal point or the letter `e` or `E` and the signed integer exponent (but not both).

The type of an real constant is `real`.

Examples

Constant	Value
`23.45e6`	23.45 x 10^6
`.0`	0.0
`0.`	0.0
`1.`	1.0
`-1.23`	-1.23
`2e-5`	2.0 x 10^-5
`3E+10`	3.0 x 10^10
`.09E34`	0.09 x 10^34

String Constants

A *string constant* consists of a sequence of characters or escape sequences enclosed in double quotes, like

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

The type of a string constant is `string`.

String constants can be of any length (provided there is enough free memory available).

String constants can be concatenated by simply writing them next to each other to form larger strings:

```
string s = "Hello" " world\n";
```

It is also possible to extend a string constant over more than one line by escaping the newline character with a backslash (`\`):

```
string s = "Hello \
world\n";
```

Escape Sequences

An *escape sequence* consists of a backslash (`\`), followed by one or more special characters:

Sequence	Value

\a	audible bell
\b	backspace
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\'	single quote
\"	double quote
\O	O = up to 3 octal digits
\xH	H = up to 2 hex digits

Any character following the initial backslash that is not mentioned in this list will be treated as that character (without the backslash).

Escape sequences can be used in character constants and string constants.

Examples

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

Punctuators

The *punctuators* used in a User Language Program are

[]	Brackets
()	Parentheses
{}	Braces
,	Comma
;	Semicolon
:	Colon
=	Equal sign

Other special characters are used as operators in a ULP.

Brackets

Brackets are used in array definitions

```
int ai[];
```

in array subscripts

```
n = ai[2];
```

and in string subscripts to access the individual characters of a string

```
string s = "Hello world";
```

```
char c = s[2];
```

Parentheses

Parentheses group expressions (possibly altering normal operator precedence), isolate conditional expressions, and indicate function calls and function parameters:

```
d = c * (a + b);
if (d == z) ++x;
func();
void func2(int n) { ... }
```

Braces

Braces indicate the start and end of a compound statement:

```
if (d == z) {
   ++x;
   func();
   }
```

and are also used to group the values of an array initializer:

```
int ai[] = { 1, 2, 3 };
```

Comma

The *comma* separates the elements of a function argument list or the parameters of a function call:

```
int func(int n, real r, string s) { ... }
int i = func(1, 3.14, "abc");
```

It also delimits the values of an array initializer:

```
int ai[] = { 1, 2, 3 };
```

and it separates the elements of a variable definition:

```
int i, j, k;
```

Semicolon

The *semicolon* terminates a statement, as in

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

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

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

Colon

The *colon* indicates the end of a label in a switch statement:

```
switch (c) {
  case 'a': printf("It was an 'a'\n"); break;
  case 'b': printf("It was a  'b'\n"); break;
  default:  printf("none of them\n");
  }
```

Equal Sign

The *equal sign* separates variable definitions from initialization lists:

```
int i = 10;
char c[] = { 'a', 'b', 'c' };
```

It is also used as an assignment operator.

Data Types

A User Language Program can define variables of different types, representing the different kinds of information available in the EAGLE data structures.

The four basic data types are

char	for single characters
int	for integral values
real	for floating point values
string	for textual information

Besides these basic data types there are also high level Object Types, which represent the data structures stored in the EAGLE data files.

The special data type `void` is used only as a return type of a function, indicating that this function does **not** return any value.

char

The data type `char` is used to store single characters, like the letters of the alphabet, or small unsigned numbers.

A variable of type `char` has a size of 8 bit (one byte), and can store any value in the range `0..255`.

See also Operators, Character Constants

int

The data type `int` is used to store signed integral values, like the coordinates of an object.

A variable of type `int` has a size of 32 bit (four byte), and can store any value in the range `-2147483648..2147483647`.

See also Integer Constants

real

The data type `real` is used to store signed floating point values, like the grid distance.

A variable of type `real` has a size of 64 bit (eight byte), and can store any value in the range `±2.2e-308..±1.7e+308` with a precision of 15 digits.

See also Real Constants

string

The data type `string` is used to store textual information, like the name of a part or net.

A variable of type `string` is not limited in it's size (provided there is enough memory available).

Variables of type `string` are defined without an explicit *size*. They grow automatically as necessary during program execution.

The elements of a `string` variable are of type `char` 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, Builtin Functions, String Constants

Implementation details

The data type `string` is actually implemented like 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()` function, as in

```
for (int i = 0; s[i]; ++i) {
    // do something with s[i]
    }
```

It is also perfectly ok to "cut off" part of a string by "punching" a zero character into it:

```
string s = "abcde";
s[3] = 0;
```

This will result in `s` having the value `"abc"`. Note that everything following the zero character will actually be gone, and it won't come back by restoring the original character. The same applies to any other operation that sets a character to 0, for instance --s[3].

Type Conversions

The result type of an arithmetic expression, such as `a + b`, where `a` and `b` are different arithmetic types, is equal to the "larger" of the two operand types.

Arithmetic types are `char`, `int` and `real` (in that order). So if, e.g. `a` is of type `int` and `b` is of type `real`, the result of the expression `a + b` would be `real`.

See also Typecast

Typecast

The result type of an arithmetic expression can be explicitly converted to a different arithmetic type by applying a *typecast* to it.

The general syntax of a typecast is

```
type(expression)
```

where `type` is one of `char`, `int` or `real`, and `expression` is any arithmetic expression.

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

See also Type Conversions

Object Types

The EAGLE data structures are stored in three binary file types:

- Library (*.lbr)
- Schematic (*.sch)
- Board (*.brd)

These data files contain a hierarchy of objects. In a User Language Program you can access these hierarchies through their respective builtin access statements:

```
library(L) { ... }
schematic(S) { ... }
board(B) { ... }
```

These access statements set up a context within which you can access all of the objects contained in the library, schematic or board.

The properties of these objects can be accessed through *members*.

There are two kinds of members:

- Data members
- Loop members

Data members immediately return the requested data from an object. For example, in

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

the data member *name* of the board object *B* returns the board's name.
Data members can also return other objects, as in

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

where the board's *grid* data member returns a grid object, of which the *size* data member then returns the grid's size.

Loop members are used to access multiple objects of the same kind, which are contained in a higher level object:

```
board(B) {
  B.elements(E) {
    printf("%-8s %-8s\n", E.name, E.value);
    }
  }
```

This example uses the board's *elements()* loop member function to set up a loop through all of the board's elements. The block following the `B.elements(E)` statement is executed in turn for each element, and the current element can be referenced inside the block through the name `E`.

Loop members process objects in alpha-numerical order, provided they have a name.

A loop member function creates a variable of the type necessary to hold the requested objects. You are free to use any valid name for such a variable, so the above example might also be written as

```
board(MyBoard) {
  MyBoard.elements(TheCurrentElement) {
    printf("%-8s %-8s\n", TheCurrentElement.name, TheCurrentElement.value);
    }
  }
```

and would do the exact same thing. The scope of the variable created by a loop member function is limited to the statement (or block) immediately following the loop function call.

Object hierarchy of a Library:

```
LIBRARY
  GRID
  LAYER
  DEVICESET
    DEVICE
    GATE
  PACKAGE
    CONTACT
      PAD
      SMD
    CIRCLE
    HOLE
    RECTANGLE
    FRAME
    TEXT
    WIRE
```

```
    POLYGON
      WIRE
  SYMBOL
    PIN
    CIRCLE
    RECTANGLE
    FRAME
    TEXT
    WIRE
    POLYGON
      WIRE
```

Object hierarchy of a Schematic:

```
SCHEMATIC
  GRID
  LAYER
  LIBRARY
  SHEET
    CIRCLE
    RECTANGLE
    FRAME
    TEXT
    WIRE
    POLYGON
      WIRE
    PART
      INSTANCE
        ATTRIBUTE
    BUS
      SEGMENT
        LABEL
          TEXT
          WIRE
        WIRE
    NET
      SEGMENT
        JUNCTION
        PINREF
        TEXT
        WIRE
```

Object hierarchy of a Board:

```
BOARD
  GRID
  LAYER
  LIBRARY
  CIRCLE
  HOLE
  RECTANGLE
  FRAME
  TEXT
  WIRE
  POLYGON
    WIRE
  ELEMENT
    ATTRIBUTE
  SIGNAL
```

CONTACTREF
POLYGON
WIRE
VIA
WIRE

UL_ARC

Data members

angle1	real (start angle, 0.0...359.9)
angle2	real (end angle, 0.0...719.9)
cap	int (CAP_...)
layer	int
radius	int
width	int
x1, y1	int (starting point)
x2, y2	int (end point)
xc, yc	int (center point)

See also UL_WIRE

Constants

CAP_FLAT	flat arc ends
CAP_ROUND	round arc ends

Note

Start and end angles are defined mathematically positive (i.e. counterclockwise), with angle1 < angle2. In order to assure this condition, the start and end point of an UL_ARC may be exchanged with respect to the UL_WIRE the arc has been derived from.

Example

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

UL_AREA

Data members

x1, y1	int (lower left corner)
x2, y2	int (upper right corner)

See also UL_BOARD, UL_DEVICE, UL_PACKAGE, UL_SHEET, UL_SYMBOL

A UL_AREA is an abstract object which gives information about the area covered by an

object. For a UL_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_ATTRIBUTE

Data members

`constant`	int (0=variable, i.e. allows overwriting, 1=constant - see note)
`defaultvalue`	string (see note)
`display`	int (`ATTRIBUTE_DISPLAY_FLAG_...`)
`name`	string
`text`	UL_TEXT (see note)
`value`	string

See also UL_DEVICE, UL_PART, UL_INSTANCE, UL_ELEMENT

Constants

`ATTRIBUTE_DISPLAY_FLAG_OFF`	nothing is displayed
`ATTRIBUTE_DISPLAY_FLAG_VALUE`	value is displayed
`ATTRIBUTE_DISPLAY_FLAG_NAME`	name is displayed

A UL_ATTRIBUTE can be used to access the *attributes* that have been defined in the library for a device, or assigned to a part in the schematic or board.

Note

`display` contains a bitwise or'ed value consisting of `ATTRIBUTE_DISPLAY_FLAG_...` and defines which parts of the attribute are actually drawn. This value is only valid if `display` is used in a UL_INSTANCE or UL_ELEMENT context.

In a UL_ELEMENT context `constant` only returns an actual value if f/b annotation is active, otherwise it returns 0.

The `defaultvalue` member returns the value as defined in the library (if different from the actual value, otherwise the same as `value`). In a UL_ELEMENT context `defaultvalue` only returns an actual value if f/b annotation is active, otherwise an empty string is returned.

The `text` member is only available in a UL_INSTANCE or UL_ELEMENT context and returns a UL_TEXT object that contains all the text parameters. The value of this text object

is the string as it will be displayed according to the UL_ATTRIBUTE's 'display' parameter. If called from a different context, the data of the returned UL_TEXT object is undefined.

For global attributes only `name` and `value` are defined.

Example

```
schematic(SCH) {
  SCH.parts(P) {
    P.attributes(A) {
      printf("%s = %s\n", A.name, A.value);
      }
    }
  }
schematic(SCH) {
  SCH.attributes(A) { // global attributes
    printf("%s = %s\n", A.name, A.value);
    }
  }
```

UL_BOARD

Data members

`area`	UL_AREA
`grid`	UL_GRID
`name`	string (see note)

Loop members

`attributes()`	UL_ATTRIBUTE (see note)
`circles()`	UL_CIRCLE
`classes()`	UL_CLASS
`elements()`	UL_ELEMENT
`frames()`	UL_FRAME
`holes()`	UL_HOLE
`layers()`	UL_LAYER
`libraries()`	UL_LIBRARY
`polygons()`	UL_POLYGON
`rectangles()`	UL_RECTANGLE
`signals()`	UL_SIGNAL
`texts()`	UL_TEXT
`wires()`	UL_WIRE

See also UL_LIBRARY, UL_SCHEMATIC

Note

The `name` member returns the full file name, including the directory.

The `attributes()` loop member loops through the *global* attributes.

Example

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

UL_BUS

Data members

name string (BUS_NAME_LENGTH)

Loop members

segments() UL_SEGMENT

See also UL_SHEET

Constants

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

Example

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

UL_CIRCLE

Data members

layer int
radius int
width int
x, y int (center point)

See also UL_BOARD, UL_PACKAGE, UL_SHEET, UL_SYMBOL

Example

```
board(B) {
  B.circles(C) {
    printf("Circle: (%d %d), r=%d, w=%d\n",
           C.x, C.y, C.radius, C.width);
    }
  }
```

UL_CLASS

Data members

clearance[number] int (see note)

drill	int
name	string (see note)
number	int
width	int

See also Design Rules, UL_NET, UL_SIGNAL, UL_SCHEMATIC, UL_BOARD

Note

The clearance member returns the clearance value between this net class and the net class with the given number. If the number (and the square brackets) is ommitted, the net class's own clearance value is returned. If a number is given, it must be between 0 and the number of this net class.

If the name member returns an empty string, the net class is not defined and therefore not in use by any signal or net.

Example

```
board(B) {
  B.signals(S) {
    printf("%-10s %d %s\n", S.name, S.class.number, S.class.name);
    }
  }
```

UL_CONTACT

Data members

name	string (CONTACT_NAME_LENGTH)
pad	UL_PAD
signal	string
smd	UL_SMD
x, y	int (center point, see note)

See also UL_PACKAGE, UL_PAD, UL_SMD, UL_CONTACTREF, UL_PINREF

Constants

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

Note

The signal data member returns the signal this contact is connected to (only available in a board context).

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

- if the contact is derived from a UL_LIBRARY context, the coordinates of the contact will be the same as defined in the package drawing
- in all other cases, they will have the actual values from the board

Example

```
library(L) {
  L.packages(PAC) {
    PAC.contacts(C) {
      printf("Contact: '%s', (%d %d)\n",
             C.name, C.x, C.y);
      }
    }
  }
```

UL_CONTACTREF

Data members

contact	UL_CONTACT
element	UL_ELEMENT

See also UL_SIGNAL, UL_PINREF

Example

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

UL_DEVICE

Data members

area	UL_AREA
description	string
headline	string
library	string
name	string (DEVICE_NAME_LENGTH)
package	UL_PACKAGE (see note)
prefix	string (DEVICE_PREFIX_LENGTH)
technologies	string (see note)
value	string ("On" or "Off")

Loop members

attributes()	UL_ATTRIBUTE (see note)
gates()	UL_GATE

See also UL_DEVICESET, UL_LIBRARY, UL_PART

Constants

DEVICE_NAME_LENGTH	max. recommended length of a device name (used in formatted output only)

DEVICE_PREFIX_LENGTH	max. recommended length of a device prefix (used in formatted output only)

All members of UL_DEVICE, except for name and technologies, return the same values as the respective members of the UL_DEVICESET in which the UL_DEVICE has been defined. The name member returns the name of the package variant this device has been created for using the PACKAGE command. When using the description text keep in mind that it may contain newline characters ('\n').

Note

The package data member returns the package that has been assigned to the device through a PACKAGE command. It can be used as a boolean function to check whether a package has been assigned to a device (see example below).

The value returned by the technologies member depends on the context in which it is called:

- if the device is derived from a UL_DEVICESET, technologies will return a string containing all of the device's technologies, separated by blanks
- if the device is derived from a UL_PART, only the actual technology used by the part will be returned.

The attributes() loop member takes an additional parameter that specifies for which technology the attributes shall be delivered (see the second example below).

Examples

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

library(L) {
  L.devicesets(DS) {
    DS.devices(D) {
      string t[];
      int n = strsplit(t, D.technologies, ' ');
      for (int i = 0; i < n; i++) {
          D.attributes(A, t[i]) {
            printf("%s = %s\n", A.name, A.value);
            }
          }
      }
    }
  }
```

UL_DEVICESET

Data members

area	UL_AREA
description	string
headline	string (see note)
library	string
name	string (DEVICE_NAME_LENGTH)
prefix	string (DEVICE_PREFIX_LENGTH)
value	string ("On" or "Off")

Loop members

devices()	UL_DEVICE
gates()	UL_GATE

See also UL_DEVICE, UL_LIBRARY, UL_PART

Constants

DEVICE_NAME_LENGTH	max. recommended length of a device name (used in formatted output only)
DEVICE_PREFIX_LENGTH	max. recommended length of a device prefix (used in formatted output only)

Note

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

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 (0.0...359.9)
attribute[]	string (see note)
column	string (see note)
locked	int
mirror	int

name	string (ELEMENT_NAME_LENGTH)
package	UL_PACKAGE
row	string (see note)
smashed	int (see note)
spin	int
value	string (ELEMENT_VALUE_LENGTH)
x, y	int (origin point)

Loop members

attributes()	UL_ATTRIBUTE
texts()	UL_TEXT (see note)

See also UL_BOARD, UL_CONTACTREF

Constants

ELEMENT_NAME_LENGTH	max. recommended length of an element name (used in formatted output only)
ELEMENT_VALUE_LENGTH	max. recommended length of an element value (used in formatted output only)

Note

The attribute[] member can be used to query a UL_ELEMENT for the value of a given attribute (see the second example below). The returned string is empty if there is no attribute by the given name, or if this attribute is explicitly empty.

The texts() member only loops through those texts of the element that have been detached using **SMASH**, and through the visible texts of any attributes assigned to this element. To process all texts of an element (e.g. when drawing it), you have to loop through the element's own texts() member as well as the texts() member of the element's package.

angle defines how many degrees the element is rotated counterclockwise around its origin.

The column and row members return the column and row location within the frame in the board drawing. If there is no frame in the drawing, or the element is placed outside the frame, a '?' (question mark) is returned.

The smashed member tells whether the element is smashed. This function can also be used to find out whether there is a detached text parameter by giving the name of that parameter in square brackets, as in smashed["VALUE"]. This is useful in case you want to select such a text with the MOVE command by doing MOVE R5>VALUE. Valid parameter names are "NAME" and "VALUE", as well as the names of any user defined attributes. They are treated case insensitive, and they may be preceded by a '>' character.

Examples

```
board(B) {
  B.elements(E) {
    printf("Element: %s, (%d %d), Package=%s\n",
```

```
            E.name, E.x, E.y, E.package.name);
    }
  }

board(B) {
  B.elements(E) {
    if (E.attribute["REMARK"])
       printf("%s: %s\n", E.name, E.attribute["REMARK"]);
    }
  }
```

UL_FRAME

Data members

columns	int (-127...127)
rows	int (-26...26)
border	int (FRAME_BORDER_...)
layer	int
x1, y1	int (lower left corner)
x2, y2	int (upper right corner)

Loop members

texts()	UL_TEXT
wires()	UL_WIRE

See also UL_BOARD, UL_PACKAGE, UL_SHEET, UL_SYMBOL

Constants

FRAME_BORDER_BOTTOM	bottom border is drawn
FRAME_BORDER_RIGHT	right border is drawn
FRAME_BORDER_TOP	top border is drawn
FRAME_BORDER_LEFT	left border is drawn

Note

border contains a bitwise or'ed value consisting of FRAME_BORDER_... and defines which of the four borders are actually drawn.

The texts() and wires() loop members loop through all the texts and wires the frame consists of.

Example

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

UL_GATE

Data members

addlevel	int (GATE_ADDLEVEL_...)
name	string (GATE_NAME_LENGTH)
swaplevel	int
symbol	UL_SYMBOL
x, y	int (origin point, see note)

See also UL_DEVICE

Constants

GATE_ADDLEVEL_MUST	must
GATE_ADDLEVEL_CAN	can
GATE_ADDLEVEL_NEXT	next
GATE_ADDLEVEL_REQUEST	request
GATE_ADDLEVEL_ALWAYS	always
GATE_NAME_LENGTH	max. recommended length of a gate name (used in formatted output only)

Note

The coordinates of the origin point (x, y) are always those of the gate's position within the device, even if the UL_GATE has been derived from a UL_INSTANCE.

Example

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

UL_GRID

Data members

distance	real
dots	int (0=lines, 1=dots)
multiple	int
on	int (0=off, 1=on)
unit	int (GRID_UNIT_...)
unitdist	int (GRID_UNIT_...)

See also UL_BOARD, UL_LIBRARY, UL_SCHEMATIC, Unit Conversions

Constants

GRID_UNIT_MIC	microns
GRID_UNIT_MM	millimeter
GRID_UNIT_MIL	mil
GRID_UNIT_INCH	inch

Note

unitdist returns the grid unit that was set to define the actual grid size (returned by distance), while unit returns the grid unit that is used to display values or interpret user input.

Example

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

UL_HOLE

Data members

diameter[layer]	int (see note)
drill	int
drillsymbol	int
x, y	int (center point)

See also UL_BOARD, UL_PACKAGE

Note

diameter[] is only defined vor layers LAYER_TSTOP and LAYER_BSTOP and returns the diameter of the solder stop mask in the given layer.

drillsymbol returns the number of the drill symbol that has been assigned to this drill diameter (see the manual for a list of defined drill symbols). A value of 0 means that no symbol has been assigned to this drill diameter.

Example

```
board(B) {
  B.holes(H) {
    printf("Hole: (%d %d), drill=%d\n",
           H.x, H.y, H.drill);
    }
  }
```

UL_INSTANCE

Data members

angle	real (0, 90, 180 and 270)

column	string (see note)
gate	UL_GATE
mirror	int
name	string (INSTANCE_NAME_LENGTH)
row	string (see note)
sheet	int (0=unused, >0=sheet number)
smashed	int (see note)
value	string (PART_VALUE_LENGTH)
x, y	int (origin point)

Loop members

attributes()	UL_ATTRIBUTE (see note)
texts()	UL_TEXT (see note)
xrefs()	UL_GATE (see note)

See also UL_PART, UL_PINREF

Constants

INSTANCE_NAME_LENGTH	max. recommended length of an instance name (used in formatted output only)
PART_VALUE_LENGTH	max. recommended length of a part value (instances do not have a value of their own!)

Note

The attributes() member only loops through those attributes that have been explicitly assigned to this instance (including *smashed* attributes).

The texts() member only loops through those texts of the instance that have been detached using **SMASH**, and through the visible texts of any attributes assigned to this instance. To process all texts of an instance, you have to loop through the instance's own texts() member as well as the texts() member of the instance's gate's symbol. If attributes have been assigned to an instance, texts() delivers their texts in the form as they are currently visible.

The column and row members return the column and row location within the frame on the sheet on which this instance is invoked. If there is no frame on that sheet, or the instance is placed outside the frame, a '?' (question mark) is returned. These members can only be used in a sheet context.

The smashed member tells whether the instance is smashed. This function can also be used to find out whether there is a detached text parameter by giving the name of that parameter in square brackets, as in smashed["VALUE"]. This is useful in case you want to select such a text with the MOVE command by doing MOVE R5>VALUE. Valid parameter names are "NAME", "VALUE", "PART" and "GATE", as well as the names of any user defined attributes. They are treated case insensitive, and they may be preceded by a '>' character.

The xrefs() member loops through the contact cross-reference gates of this instance. These are only of importance if the ULP is going to create a drawing of some sort (for instance a DXF file).

Example

```
schematic(S) {
  S.parts(P) {
    printf("Part: %s\n", P.name);
    P.instances(I) {
      if (I.sheet != 0)
         printf("\t%s used on sheet %d\n", I.name, I.sheet);
      }
    }
  }
```

UL_JUNCTION

Data members

`diameter`	int
`x, y`	int (center point)

See also UL_SEGMENT

Example

```
schematic(SCH) {
  SCH.sheets(SH) {
    SH.nets(N) {
      N.segments(SEG) {
        SEG.junctions(J) {
          printf("Junction: (%d %d)\n", J.x, J.y);
          }
        }
      }
    }
  }
```

UL_LABEL

Data members

`angle`	real (`0.0...359.9`)
`layer`	int
`mirror`	int
`spin`	int
`text`	UL_TEXT
`x, y`	int (origin point)
`xref`	int (0=plain, 1=cross-reference)

Loop members

`wires()`	UL_WIRE (see note)

See also UL_SEGMENT

Note

If `xref` returns a non-zero value, the `wires()` loop member loops through the wires that form the flag of a cross-reference label. Otherwise it is an empty loop.

The angle, layer, mirror and spin members always return the same values as those of the UL_TEXT object returned by the text member. The x and y members of the text return slightly offset values for cross-reference labels (non-zero xref), otherwise they also return the same values as the UL_LABEL.

xref is only meaningful for net labels. For bus labels it always returns 0.

Example

```
sheet(SH) {
  SH.nets(N) {
    N.segments(S) {
      S.labels(L) {
        printf("Label: %d %d '%s'\n", L.x, L.y, L.text.value);
        }
      }
    }
  }
```

UL_LAYER

Data members

color	int
fill	int
name	string (LAYER_NAME_LENGTH)
number	int
used	int (0=unused, 1=used)
visible	int (0=off, 1=on)

See also UL_BOARD, UL_LIBRARY, UL_SCHEMATIC

Constants

LAYER_NAME_LENGTH	max. recommended length of a layer name (used in formatted output only)
LAYER_TOP	layer numbers
LAYER_BOTTOM	
LAYER_PADS	
LAYER_VIAS	
LAYER_UNROUTED	
LAYER_DIMENSION	
LAYER_TPLACE	
LAYER_BPLACE	
LAYER_TORIGINS	
LAYER_BORIGINS	
LAYER_TNAMES	
LAYER_BNAMES	
LAYER_TVALUES	
LAYER_BVALUES	

LAYER_TSTOP
LAYER_BSTOP
LAYER_TCREAM
LAYER_BCREAM
LAYER_TFINISH
LAYER_BFINISH
LAYER_TGLUE
LAYER_BGLUE
LAYER_TTEST
LAYER_BTEST
LAYER_TKEEPOUT
LAYER_BKEEPOUT
LAYER_TRESTRICT
LAYER_BRESTRICT
LAYER_VRESTRICT
LAYER_DRILLS
LAYER_HOLES
LAYER_MILLING
LAYER_MEASURES
LAYER_DOCUMENT
LAYER_REFERENCE
LAYER_TDOCU
LAYER_BDOCU
LAYER_NETS
LAYER_BUSSES
LAYER_PINS
LAYER_SYMBOLS
LAYER_NAMES
LAYER_VALUES
LAYER_INFO
LAYER_GUIDE
LAYER_USER lowest number for user defined layers (100)

Example

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

UL_LIBRARY

Data members

description	string (see note)
grid	UL_GRID
headline	string

`name`	string (`LIBRARY_NAME_LENGTH`, see note)

Loop members

`devices()`	UL_DEVICE
`devicesets()`	UL_DEVICESET
`layers()`	UL_LAYER
`packages()`	UL_PACKAGE
`symbols()`	UL_SYMBOL

See also UL_BOARD, UL_SCHEMATIC

Constants

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

The `devices()` member loops through all the package variants and technologies of all UL_DEVICESETs in the library, thus resulting in all the actual device variations available. The `devicesets()` member only loops through the UL_DEVICESETs, which in turn can be queried for their UL_DEVICE members.

Note

The `description` member returns the complete descriptive text as defined with the DESCRIPTION command, while the `headline` member returns only the first line of the description, without any HTML tags. When using the `description` text keep in mind that it may contain newline characters (`'\n'`). The `description` and `headline` information is only available within a library drawing, not if the library is derived form a UL_BOARD or UL_SCHEMATIC context.

If the library is derived form a UL_BOARD or UL_SCHEMATIC context, `name` returns the pure library name (without path or extension). Otherwise it returns the full library file name.

Example

```
library(L) {
  L.devices(D)     printf("Dev: %s\n", D.name);
  L.devicesets(D)  printf("Dev: %s\n", D.name);
  L.packages(P)    printf("Pac: %s\n", P.name);
  L.symbols(S)     printf("Sym: %s\n", S.name);
  }
schematic(S) {
  S.libraries(L) printf("Library: %s\n", L.name);
  }
```

UL_NET

Data members

`class`	UL_CLASS
`column`	string (see note)
`name`	string (`NET_NAME_LENGTH`)

`row` string (see note)

Loop members

`pinrefs()` UL_PINREF (see note)

`segments()` UL_SEGMENT (see note)

See also UL_SHEET, UL_SCHEMATIC

Constants

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

Note

The `pinrefs()` loop member can only be used if the net is in a schematic context.
The `segments()` loop member can only be used if the net is in a sheet context.

The `column` and `row` members return the column and row locations within the frame on the sheet on which this net is drawn. Since a net can extend over a certain area, each of these functions returns two values, separated by a blank. In case of `column` these are the left- and rightmost columns touched by the net, and in case of `row` it's the top- and bottommost row.

When determining the column and row of a net on a sheet, first the column and then the row within that column is taken into account. Here XREF labels take precedence over normal labels, which again take precedence over net wires.

If there is no frame on that sheet, `"? ?"` (two question marks) is returned. If any part of the net is placed outside the frame, either of the values may be `'?'` (question mark). These members can only be used in a sheet context.

Example

```
schematic(S) {
  S.nets(N) {
    printf("Net: %s\n", N.name);
    // N.segments(SEG) will NOT work here!
    }
  }
schematic(S) {
  S.sheets(SH) {
    SH.nets(N) {
      printf("Net: %s\n", N.name);
      N.segments(SEG) {
        SEG.wires(W) {
          printf("\tWire: (%d %d) (%d %d)\n",
                 W.x1, W.y1, W.x2, W.y2);
        }
      }
    }
  }
```

UL_PACKAGE

Data members

area	UL_AREA
description	string
headline	string
library	string
name	string (PACKAGE_NAME_LENGTH)

Loop members

circles()	UL_CIRCLE
contacts()	UL_CONTACT
frames()	UL_FRAME
holes()	UL_HOLE
polygons()	UL_POLYGON
rectangles()	UL_RECTANGLE
texts()	UL_TEXT (see note)
wires()	UL_WIRE

See also UL_DEVICE, UL_ELEMENT, UL_LIBRARY

Constants

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

Note

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

If the UL_PACKAGE is derived from a UL_ELEMENT, the texts() member only loops through the non-detached texts of that element.

Example

```
library(L) {
  L.packages(PAC) {
    printf("Package: %s\n", PAC.name);
    PAC.contacts(C) {
      if (C.pad)
         printf("\tPad: %s, (%d %d)\n",
                 C.name, C.pad.x, C.pad.y);
      else if (C.smd)
         printf("\tSmd: %s, (%d %d)\n",
                 C.name, C.smd.x, C.smd.y);
      }
    }
  }
board(B) {
  B.elements(E) {
```

```
    printf("Element: %s, Package: %s\n", E.name, E.package.name);
    }
  }
```

UL_PAD

Data members

angle	real (0.0...359.9)
diameter[layer]	int
drill	int
drillsymbol	int
elongation	int
flags	int (PAD_FLAG_...)
name	string (PAD_NAME_LENGTH)
shape[layer]	int (PAD_SHAPE_...)
signal	string
x, y	int (center point, see note)

See also UL_PACKAGE, UL_CONTACT, UL_SMD

Constants

PAD_FLAG_STOP	generate stop mask
PAD_FLAG_THERMALS	generate thermals
PAD_FLAG_FIRST	use special "first pad" shape
PAD_SHAPE_SQUARE	square
PAD_SHAPE_ROUND	round
PAD_SHAPE_OCTAGON	octagon
PAD_SHAPE_LONG	long
PAD_SHAPE_OFFSET	offset
PAD_SHAPE_ANNULUS	annulus (only if supply layers are used)
PAD_SHAPE_THERMAL	thermal (only if supply layers are used)
PAD_NAME_LENGTH	max. recommended length of a pad name (same as CONTACT_NAME_LENGTH)

Note

The parameters of the pad depend on the context in which it is accessed:

- if the pad is derived from a UL_LIBRARY context, the coordinates (x, y) and angle will be the same as defined in the package drawing
- in all other cases, they will have the actual values from the board

The diameter and shape of the pad depend on the layer for which they shall be retrieved, because they may be different in each layer depending on the Design Rules. If one of the layers LAYER_TOP..LAYER_BOTTOM, LAYER_TSTOP or LAYER_BSTOP is given as the

index to the diameter or shape data member, the resulting value will be calculated according to the Design Rules. If LAYER_PADS is given, the raw value as defined in the library will be returned.

`drillsymbol` returns the number of the drill symbol that has been assigned to this drill diameter (see the manual for a list of defined drill symbols). A value of `0` means that no symbol has been assigned to this drill diameter.

`angle` defines how many degrees the pad is rotated counterclockwise around its center.

`elongation` is only valid for shapes PAD_SHAPE_LONG and PAD_SHAPE_OFFSET and defines how many percent the long side of such a pad is longer than its small side. This member returns 0 for any other pad shapes.

The value returned by `flags` must be masked with the `PAD_FLAG_...` constants to determine the individual flag settings, as in

```
if (pad.flags & PAD_FLAG_STOP) {
   ...
   }
```

Note that if your ULP just wants to draw the objects, you don't need to check these flags explicitly. The `diameter[]` and `shape[]` members will return the proper data; for instance, if `PAD_FLAG_STOP` is set, `diameter[LAYER_TSTOP]` will return `0`, which should result in nothing being drawn in that layer. The `flags` member is mainly for ULPs that want to create script files that create library objects.

Example

```
library(L) {
  L.packages(PAC) {
    PAC.contacts(C) {
      if (C.pad)
         printf("Pad: '%s', (%d %d), d=%d\n",
                C.name, C.pad.x, C.pad.y, C.pad.diameter[LAYER_BOTTOM]);
      }
    }
  }
```

UL_PART

Data members

`attribute[]`	string (see note)
`device`	UL_DEVICE
`deviceset`	UL_DEVICESET
`name`	string (`PART_NAME_LENGTH`)
`value`	string (`PART_VALUE_LENGTH`)

Loop members

`attributes()`	UL_ATTRIBUTE (see note)
`instances()`	UL_INSTANCE (see note)

See also UL_SCHEMATIC, UL_SHEET

Constants

PART_NAME_LENGTH	max. recommended length of a part name (used in formatted output only)
PART_VALUE_LENGTH	max. recommended length of a part value (used in formatted output only)

Note

The attribute[] member can be used to query a UL_PART for the value of a given attribute (see the second example below). The returned string is empty if there is no attribute by the given name, or if this attribute is explicitly empty.

When looping through the attributes() of a UL_PART, only the name, value, defaultvalue and constant members of the resulting UL_ATTRIBUTE objects are valid.

If the part is in a sheet context, the instances() loop member loops only through those instances that are actually used on that sheet. If the part is in a schematic context, all instances are looped through.

Example

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

schematic(SCH) {
  SCH.parts(P) {
    if (P.attribute["REMARK"])
       printf("%s: %s\n", P.name, P.attribute["REMARK"]);
    }
  }
```

UL_PIN

Data members

angle	real (0, 90, 180 and 270)
contact	UL_CONTACT (see note)
direction	int (PIN_DIRECTION_...)
function	int (PIN_FUNCTION_FLAG_...)
length	int (PIN_LENGTH_...)
name	string (PIN_NAME_LENGTH)
net	string (see note)
swaplevel	int
visible	int (PIN_VISIBLE_FLAG_...)
x, y	int (connection point)

Loop members

circles()	UL_CIRCLE
texts()	UL_TEXT

`wires()` UL_WIRE

See also UL_SYMBOL, UL_PINREF, UL_CONTACTREF

Constants

`PIN_DIRECTION_NC`	not connected
`PIN_DIRECTION_IN`	input
`PIN_DIRECTION_OUT`	output (totem-pole)
`PIN_DIRECTION_IO`	in/output (bidirectional)
`PIN_DIRECTION_OC`	open collector
`PIN_DIRECTION_PWR`	power input pin
`PIN_DIRECTION_PAS`	passive
`PIN_DIRECTION_HIZ`	high impedance output
`PIN_DIRECTION_SUP`	supply pin
`PIN_FUNCTION_FLAG_NONE`	no symbol
`PIN_FUNCTION_FLAG_DOT`	inverter symbol
`PIN_FUNCTION_FLAG_CLK`	clock symbol
`PIN_LENGTH_POINT`	no wire
`PIN_LENGTH_SHORT`	0.1 inch wire
`PIN_LENGTH_MIDDLE`	0.2 inch wire
`PIN_LENGTH_LONG`	0.3 inch wire
`PIN_NAME_LENGTH`	max. recommended length of a pin name (used in formatted output only)
`PIN_VISIBLE_FLAG_OFF`	no name drawn
`PIN_VISIBLE_FLAG_PAD`	pad name drawn
`PIN_VISIBLE_FLAG_PIN`	pin name drawn

Note

The `contact` data member returns the contact that has been assigned to the pin through a CONNECT 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

The `name` data member always returns the name of the pin as it was defined in the library, with any `'@'` character for pins with the same name left intact (see the PIN command for details).
The `texts` loop member, on the other hand, returns the pin name (if it is visible) in the same way as it is displayed in the current drawing type.

The `net` data member returns the name of the net to which this pin is connected (only available in a schematic context).

Example

```
library(L) {
  L.symbols(S) {
    printf("Symbol: %s\n", S.name);
    S.pins(P) {
      printf("\tPin: %s, (%d %d)", P.name, P.x, P.y);
      if (P.direction == PIN_DIRECTION_IN)
         printf(" input");
      if ((P.function & PIN_FUNCTION_FLAG_DOT) != 0)
         printf(" inverted");
      printf("\n");
      }
    }
  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
`part`	UL_PART
`pin`	UL_PIN

See also UL_SEGMENT, UL_CONTACTREF

Example

```
schematic(SCH) {
  SCH.sheets(SH) {
    printf("Sheet: %d\n", SH.number);
    SH.nets(N) {
      printf("\tNet: %s\n", N.name);
      N.segments(SEG) {
        SEG.pinrefs(P) {
          printf("connected to: %s, %s, %s\n",
                 P.part.name, P.instance.name, P.pin.name);
          }
        }
      }
    }
  }
```

UL_POLYGON

Data members

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

Loop members

`contours()`	UL_WIRE (see note)
`fillings()`	UL_WIRE
`wires()`	UL_WIRE

See also UL_BOARD, UL_PACKAGE, UL_SHEET, UL_SIGNAL, UL_SYMBOL

Constants

`POLYGON_POUR_SOLID`	solid
`POLYGON_POUR_HATCH`	hatch

Note

The `contours()` and `fillings()` loop members loop through the wires that are used to draw the calculated polygon if it is part of a signal and the polygon has been calculated by the RATSNEST command. The `wires()` loop member always loops through the polygon wires as they were drawn by the user. For an uncalculated signal polygon `contours()` does the same as `wires()`, and `fillings()` does nothing.

If the `contours()` loop member is called without a second parameter, it loops through all of the contour wires, regardless whether they belong to a positive or a negative polygon. If you are interested in getting the positive and negative contour wires separately, you can call `contours()` with an additional integer parameter (see the second example below). The sign of that parameter determines whether a positive or a negative polygon will be handled, and the value indicates the index of that polygon. If there is no polygon with the given index, the statement will not be executed. Another advantage of this method is that you don't need to determine the beginning and end of a particular polygon yourself (by comparing coordinates). For any given index, the statement will be executed for all the wires of that polygon. With the second parameter `0` the behavior is the same as without a second parameter.

Polygon width

When using the `fillings()` loop member to get the fill wires of a solid polygon, make sure the *width* of the polygon is not zero (actually it should be quite a bit larger than zero, for example at least the hardware resolution of the output device you are going to draw

on). **Filling a polygon with zero width may result in enormous amounts of data, since it will be calculated with the smallest editor resolution of 1/10000mm!**

Partial polygons

A calculated signal polygon may consist of several distinct parts (called *positive* polygons), each of which can contain extrusions (*negative* polygons) resulting from other objects being subtracted from the polygon. Negative polygons can again contain other positive polygons and so on.

The wires looped through by `contours()` always start with a positive polygon. To find out where one partial polygon ends and the next one begins, simply store the (x1,y1) coordinates of the first wire and check them against (x2,y2) of every following wire. As soon as these are equal, the last wire of a partial polygon has been found. It is also guaranteed that the second point (x2,y2) of one wire is identical to the first point (x1,y1) of the next wire in that partial polygon.

To find out where the "inside" and the "outside" of the polygon lays, take any contour wire and imagine looking from its point (x1,y1) to (x2,y2). The "inside" of the polygon is always on the right side of the wire. Note that if you simply want to draw the polygon you won't need all these details.

Example

```
board(B) {
  B.signals(S) {
    S.polygons(P) {
      int x0, y0, first = 1;
      P.contours(W) {
        if (first) {
           // a new partial polygon is starting
           x0 = W.x1;
           y0 = W.y1;
           }
        // ...
        // do something with the wire
        // ...
        if (first)
           first = 0;
        else if (W.x2 == x0 && W.y2 == y0) {
           // this was the last wire of the partial polygon,
           // so the next wire (if any) will be the first wire
           // of the next partial polygon
           first = 1;
           }
        }
      }
    }
  }

board(B) {
  B.signals(S) {
    S.polygons(P) {
      // handle only the "positive" polygons:
```

```
      int i = 1;
      int active;
      do {
         active = 0;
         P.contours(W, i) {
           active = 1;
           // do something with the wire
           }
         i++;
         } while (active);
      }
    }
  }
```

UL_RECTANGLE

Data members

`angle`	real (`0.0...359.9`)
`layer`	int
`x1, y1`	int (lower left corner)
`x2, y2`	int (upper right corner)

See also UL_BOARD, UL_PACKAGE, UL_SHEET, UL_SYMBOL

`angle` defines how many degrees the rectangle is rotated counterclockwise around its center. The center coordinates are given by `(x1+x2)/2` and `(y1+y2)/2`.

Example

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

UL_SCHEMATIC

Data members

`grid`	UL_GRID
`name`	string (see note)
`xreflabel`	string

Loop members

`attributes()`	UL_ATTRIBUTE (see note)
`classes()`	UL_CLASS
`layers()`	UL_LAYER
`libraries()`	UL_LIBRARY
`nets()`	UL_NET
`parts()`	UL_PART
`sheets()`	UL_SHEET

See also UL_BOARD, UL_LIBRARY

Note

The `name` member returns the full file name, including the directory.

The `xreflabel` member returns the format string used to display cross-reference labels.

The `attributes()` loop member loops through the *global* attributes.

Example

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

UL_SEGMENT

Loop members

`junctions()`	UL_JUNCTION (see note)
`labels()`	UL_LABEL
`pinrefs()`	UL_PINREF (see note)
`texts()`	UL_TEXT (deprecated, see note)
`wires()`	UL_WIRE

See also UL_BUS, UL_NET

Note

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

The `texts()` loop member was used in older EAGLE versions to loop through the labels of a segment, and is only present for compatibility. It will not deliver the text of cross-reference labels at the correct position. Use the `labels()` loop member to access a segment's labels.

Example

```
schematic(SCH) {
  SCH.sheets(SH) {
    printf("Sheet: %d\n", SH.number);
    SH.nets(N) {
      printf("\tNet: %s\n", N.name);
      N.segments(SEG) {
        SEG.pinrefs(P) {
          printf("connected to: %s, %s, %s\n",
                 P.part.name, P.instance.name, P.pin.name);
        }
      }
    }
  }
```

UL_SHEET

Data members

area	UL_AREA
number	int

Loop members

busses()	UL_BUS
circles()	UL_CIRCLE
frames()	UL_FRAME
nets()	UL_NET
parts()	UL_PART
polygons()	UL_POLYGON
rectangles()	UL_RECTANGLE
texts()	UL_TEXT
wires()	UL_WIRE

See also UL_SCHEMATIC

Example

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

UL_SIGNAL

Data members

airwireshidden	int
class	UL_CLASS
name	string (SIGNAL_NAME_LENGTH)

Loop members

contactrefs()	UL_CONTACTREF
polygons()	UL_POLYGON
vias()	UL_VIA
wires()	UL_WIRE

See also UL_BOARD

Constants

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

Example

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

UL_SMD

Data members

angle	real (0.0...359.9)
dx[layer], dy[layer]	int (size)
flags	int (SMD_FLAG_...)
layer	int (see note)
name	string (SMD_NAME_LENGTH)
roundness	int (see note)
signal	string
x, y	int (center point, see note)

See also UL_PACKAGE, UL_CONTACT, UL_PAD

Constants

SMD_FLAG_STOP	generate stop mask
SMD_FLAG_THERMALS	generate thermals
SMD_FLAG_CREAM	generate cream mask
SMD_NAME_LENGTH	max. recommended length of an smd name (same as CONTACT_NAME_LENGTH)

Note

The parameters of the smd depend on the context in which it is accessed:

- if the smd is derived from a UL_LIBRARY context, the coordinates (x, y), angle, layer and roundness of the smd will be the same as defined in the package drawing
- in all other cases, they will have the actual values from the board

If the dx and dy data members are called with an optional layer index, the data for that layer is returned according to the Design Rules. Valid layers are LAYER_TOP, LAYER_TSTOP and LAYER_TCREAM for an smd in the Top layer, and LAYER_BOTTOM, LAYER_BSTOP and LAYER_BCREAM for an smd in the Bottom layer, respectively.

angle defines how many degrees the smd is rotated counterclockwise around its center.

The value returned by flags must be masked with the SMD_FLAG_... constants to determine the individual flag settings, as in

```
if (smd.flags & SMD_FLAG_STOP) {
   ...
   }
```

Note that if your ULP just wants to draw the objects, you don't need to check these flags explicitly. The dx[] and dy[] members will return the proper data; for instance, if SMD_FLAG_STOP is set, dx[LAYER_TSTOP] will return 0, which should result in nothing being drawn in that layer. The flags member is mainly for ULPs that want to create script files that create library objects.

Example

```
library(L) {
  L.packages(PAC) {
    PAC.contacts(C) {
      if (C.smd)
         printf("Smd: '%s', (%d %d), dx=%d, dy=%d\n",
                 C.name, C.smd.x, C.smd.y, C.smd.dx, C.smd.dy);
      }
    }
  }
```

UL_SYMBOL

Data members

area	UL_AREA
library	string
name	string (SYMBOL_NAME_LENGTH)

Loop members

circles()	UL_CIRCLE
frames()	UL_FRAME
rectangles()	UL_RECTANGLE
pins()	UL_PIN
polygons()	UL_POLYGON
texts()	UL_TEXT (see note)
wires()	UL_WIRE

See also UL_GATE, UL_LIBRARY

Constants

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

Note

If the UL_SYMBOL is derived from a UL_INSTANCE, the `texts()` member only loops through the non-detached texts of that instance.

Example

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

UL_TEXT

Data members

angle	real (0.0...359.9)
font	int (FONT_...)
layer	int

`mirror`	int
`ratio`	int
`size`	int
`spin`	int
`value`	string
`x, y`	int (origin point)

Loop members

`wires()`	UL_WIRE (see note)

See also UL_BOARD, UL_PACKAGE, UL_SHEET, UL_SYMBOL

Constants

`FONT_VECTOR`	vector font
`FONT_PROPORTIONAL`	proportional font
`FONT_FIXED`	fixed font

Note

The `wires()` loop member always accesses the individual wires the text is composed of when using the vector font, even if the actual font is not `FONT_VECTOR`.

If the UL_TEXT is derived from a UL_ELEMENT or UL_INSTANCE context, the member values will be those of the actual text as located in the board or sheet drawing.

Example

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

UL_VIA

Data members

`diameter[layer]`	int
`drill`	int
`drillsymbol`	int
`end`	int
`flags`	int (`VIA_FLAG_...`)
`shape[layer]`	int (`VIA_SHAPE_...`)
`start`	int
`x, y`	int (center point)

See also UL_SIGNAL

Constants

`VIA_FLAG_STOP`	always generate stop mask

VIA_SHAPE_SQUARE	square
VIA_SHAPE_ROUND	round
VIA_SHAPE_OCTAGON	octagon
VIA_SHAPE_ANNULUS	annulus
VIA_SHAPE_THERMAL	thermal

Note

The diameter and shape of the via depend on the layer for which they shall be retrieved, because they may be different in each layer depending on the Design Rules. If one of the layers LAYER_TOP..LAYER_BOTTOM, LAYER_TSTOP or LAYER_BSTOP is given as the index to the diameter or shape data member, the resulting value will be calculated according to the Design Rules. If LAYER_VIAS is given, the raw value as defined in the via will be returned.

Note that `diameter` and `shape` will always return the diameter or shape that a via would have in the given layer, even if that particular via doesn't cover that layer (or if that layer isn't used in the layer setup at all).
If the given layer is a supply layer on which the via is connected, and the Design Rules parameter "Supply/Generate thermals for vias" is turned off, `diameter` will be 0 and `shape` will be `VIA_SHAPE_ROUND`.

`start` and `end` return the layer numbers in which that via starts and ends. The value of `start` will always be less than that of `end`.

`drillsymbol` returns the number of the drill symbol that has been assigned to this drill diameter (see the manual for a list of defined drill symbols). A value of `0` means that no symbol has been assigned to this drill diameter.

Example

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

UL_WIRE

Data members

arc	UL_ARC
cap	int (CAP_...)
curve	real
layer	int
style	int (WIRE_STYLE_...)
width	int
x1, y1	int (starting point)

`x2, y2` int (end point)

Loop members

`pieces()` UL_WIRE (see note)

See also UL_BOARD, UL_PACKAGE, UL_SEGMENT, UL_SHEET, UL_SIGNAL, UL_SYMBOL, UL_ARC

Constants

`CAP_FLAT`	flat arc ends
`CAP_ROUND`	round arc ends
`WIRE_STYLE_CONTINUOUS`	continuous
`WIRE_STYLE_LONGDASH`	long dash
`WIRE_STYLE_SHORTDASH`	short dash
`WIRE_STYLE_DASHDOT`	dash dot

Wire Style

A UL_WIRE that has a *style* other than `WIRE_STYLE_CONTINUOUS` can use the `pieces()` loop member to access the individual segments that constitute for example a dashed wire. If `pieces()` is called for a UL_WIRE with `WIRE_STYLE_CONTINUOUS`, a single segment will be accessible which is just the same as the original UL_WIRE. The `pieces()` loop member can't be called from a UL_WIRE that itself has been returned by a call to `pieces()` (this would cause an infinite recursion).

Arcs at Wire level

Arcs are basically wires, with a few additional properties. At the first level arcs are treated exactly the same as wires, meaning they have a start and an end point, a width, layer and wire style. In addition to these an arc, at the wire level, has a *cap* and a *curve* parameter. *cap* defines whether the arc endings are round or flat, and *curve* defines the "curvature" of the arc. The valid range for *curve* is `-360..+360`, and its value means what part of a full circle the arc consists of. A value of `90`, for instance, would result in a `90°` arc, while `180` would give you a semicircle. The maximum value of `360` can only be reached theoretically, since this would mean that the arc consists of a full circle, which, because the start and end points have to lie on the circle, would have to have an infinitely large diameter. Positive values for *curve* mean that the arc is drawn in a mathematically positive sense (i.e. counterclockwise). If *curve* is `0`, the arc is a straight line ("no curvature"), which is actually a wire.

The *cap* parameter only has a meaning for actual arcs, and will always return `CAP_ROUND` for a straight wire.

Whether or not an UL_WIRE is an arc can be determined by checking the boolean return value of the `arc` data member. If it returns `0`, we have a straight wire, otherwise an arc. If `arc` returns a non-zero value it may be further dereferenced to access the UL_ARC specific parameters start and end angle, radius and center point. Note that you may only need these additional parameters if you are going to draw the arc or process it in other ways where the actual shape is important.

Example

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

Definitions

The data items to be used in a User Language Program must be defined before they can be used.

There are three kinds of definitions:

- Constant Definitions
- Variable Definitions
- Function Definitions

The scope of a *constant* or *variable* definition goes from the line in which it has been defined to the end of the current block, or to the end of the User Language Program, if the definition appeared outside any block.

The scope of a *function* definition goes from the closing brace (`}`) of the function body to the end of the User Language Program.

Constant Definitions

Constants are defined using the keyword `enum`, as in

```
enum { a, b, c };
```

which would define the three constants `a`, `b` and `c`, giving them the values `0`, `1` and `2`, respectively.

Constants may also be initialized to specific values, like

```
enum { a, b = 5, c };
```

where `a` would be `0`, `b` would be `5` and `c` would be `6`.

Variable Definitions

The general syntax of a *variable definition* is

```
[numeric] type identifier [= initializer][, ...];
```

where `type` is one of the data or object types, `identifier` is the name of the variable, and `initializer` is a optional initial value.

Multiple variable definitions of the same `type` are separated by commas (`,`).

If `identifier` is followed by a pair of brackets (`[]`), this defines an array of variables of

the given `type`. The size of an array is automatically adjusted at runtime.

The optional keyword `numeric` can be used with string arrays to have them sorted alphanumerically by the sort() function.

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

Examples

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

The members of array elements of object types can't be accessed directly:

```
UL_SIGNAL signals[];
...
UL_SIGNAL s = signals[0];
printf("%s", s.name);
```

Function Definitions

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

The general syntax of a *function definition* is

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

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

Functions that do not return a value have the type `void`.

A function must be defined **before** it can be called, and function calls can not be recursive (a function cannot call itself).

The statements in the function body may modify the values of the parameters, but this will not have any effect on the arguments of the function call.

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

A call to the exit() function will terminate the entire User Language Program.

The special function main()

If your User Language Program contains a function called main(), that function will be explicitly called as the main function, and it's return value will be the return value of the program.

Command line arguments are available to the program through the global Builtin Variables argc and argv.

Example

```
int CountDots(string s)
{
  int dots = 0;
  for (int i = 0; s[i]; ++i)
      if (s[i] == '.')
         ++dots;
  return dots;
}
string dotted = "This.has.dots...";
output("test") {
  printf("Number of dots: %d\n",
                 CountDots(dotted));
  }
```

Operators

The following table lists all of the User Language operators, in order of their precedence (*Unary* having the highest precedence, *Comma* the lowest):

Unary	! ~ + - ++ --
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< <= > >=
Equality	== !=
Bitwise AND	&
Bitwise XOR	^
Bitwise OR	|
Logical AND	&&
Logical OR	||
Conditional	?:
Assignment	= *= /= %= += -= &= ^= |= <<= >>=
Comma	,

Associativity is **left to right** for all operators, except for *Unary*, *Conditional* and *Assignment*, which are **right to left** associative.

The normal operator precedence can be altered by the use of parentheses.

Bitwise Operators

Bitwise operators work only with data types char and int.

Unary

~	Bitwise (1's) complement

Binary

<<	Shift left
>>	Shift right
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR

Assignment

&=	Assign bitwise AND
^=	Assign bitwise XOR
|=	Assign bitwise OR
<<=	Assign left shift
>>=	Assign right shift

Logical Operators

Logical operators work with expressions of any data type.

Unary

!	Logical NOT

Binary

&&	Logical AND
||	Logical OR

Using a string expression with a logical operator checks whether the string is empty.

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

Comparison Operators

Comparison operators work with expressions of any data type, except Object Types.

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Evaluation Operators

Evaluation operators are used to evaluate expressions based on a condition, or to group a sequence of expressions and have them evaluated as one expression.

?:	Conditional

`,` Comma

The *Conditional* operator is used to make a decision within an expression, as in

```
int a;
// ...code that calculates 'a'
string s = a ? "True" : "False";
```

which is basically the same as

```
int a;
string s;
// ...code that calculates 'a'
if (a)
   s = "True";
else
   s = "False";
```

but the advantage of the conditional operator is that it can be used in an expression.

The *Comma* operator is used to evaluate a sequence of expressions from left to right, using the type and value of the right operand as the result.

Note that arguments in a function call as well as multiple variable declarations also use commas as delimiters, but in that case this is **not** a comma operator!

Arithmetic Operators

Arithmetic operators work with data types `char`, `int` and `real` (except for `++`, `--`, `%` and `%=`).

Unary

`+`	Unary plus
`-`	Unary minus
`++`	Pre- or postincrement
`--`	Pre- or postdecrement

Binary

`*`	Multiply
`/`	Divide
`%`	Remainder (modulus)
`+`	Binary plus
`-`	Binary minus

Assignment

`=`	Simple assignment
`*=`	Assign product
`/=`	Assign quotient
`%=`	Assign remainder (modulus)
`+=`	Assign sum
`-=`	Assign difference

See also String Operators

String Operators

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

Binary

+	Concatenation

Assignment

=	Simple assignment
+=	Append to string

The + operator concatenates two strings, or adds a character to the end of a string and returns the resulting string.

The += operator appends a string or a character to the end of a given string.

See also Arithmetic Operators

Expressions

An *expression* can be one of the following:

- Arithmetic Expression
- Assignment Expression
- String Expression
- Comma Expression
- Conditional Expression
- Function Call

Expressions can be grouped using parentheses, and may be recursive, meaning that an expression can consist of subexpressions.

Arithmetic Expression

An *arithmetic expression* is any combination of numeric operands and an arithmetic operator or a bitwise operator.

Examples

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

Assignment Expression

An *assignment expression* consists of a variable on the left side of an assignment operator, and an expression on the right side.

Examples

```
a = x + 42
b += c
```

```
s = "Hello"
```

String Expression

A *string expression* is any combination of string and char operands and a string operator.

Examples

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

Comma Expression

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

Comma expressions are evaluated left to right, and the result of a comma expression is the type and value of the rightmost expression.

Example

```
i++, j++, k++
```

Conditional Expression

A *conditional expression* uses the conditional operator to make a decision within an expression.

Example

```
int a;
// ...code that calculates 'a'
string s = a ? "True" : "False";
```

Function Call

A *function call* transfers the program flow to a user defined function or a builtin function. The formal parameters defined in the function definition are replaced with the values of the expressions used as the actual arguments of the function call.

Example

```
int p = strchr(s, 'b');
```

Statements

A *statement* can be one of the following:

- Compound Statement
- Control Statement

- Expression Statement
- Builtin Statement
- Constant Definition
- Variable Definition

Statements specify the flow of control as a User Language Program executes. In absence of specific control statements, statements are executed sequentially in the order of appearance in the ULP file.

Compound Statement

A *compound statement* (also known as *block*) is a list (possibly empty) of statements enclosed in matching braces (`{}`). Syntactically, a block can be considered to be a single statement, but it also controls the scoping of identifiers. An identifier declared within a block has a scope starting at the point of declaration and ending at the closing brace.

Compound statements can be nested to any depth.

Expression Statement

An *expression statement* is any expression followed by a semicolon.

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

A special case is the *empty statement*, consisting of only a semicolon. An empty statement does nothing, but it may be useful in situations where the ULP syntax expects a statement but your program does not need one.

Control Statements

Control statements are used to control the program flow.

Iteration statements are

`do...while`
`for`
`while`

Selection statements are

`if...else`
`switch`

Jump statements are

`break`
`continue`
`return`

break

The *break* statement has the general syntax

```
break;
```

and immediately terminates the **nearest** enclosing do...while, for, switch or while statement. This also applies to *loop members* of object types.

Since all of these statements can be intermixed and nested to any depth, take care to ensure that your `break` exits from the correct statement.

continue

The *continue* statement has the general syntax

```
continue;
```

and immediately transfers control to the test condition of the **nearest** enclosing do...while, while, or for statement, or to the increment expression of the **nearest** enclosing for statement.

Since all of these statements can be intermixed and nested to any depth, take care to ensure that your `continue` affects the correct statement.

do...while

The *do...while* statement has the general syntax

```
do statement while (condition);
```

and executes the `statement` until the `condition` expression becomes zero.

The `condition` is tested **after** the first execution of `statement`, which means that the statement is always executed at least one time.

If there is no `break` or `return` inside the `statement`, the `statement` must affect the value of the `condition`, or `condition` itself must change during evaluation in order to avoid an endless loop.

Example

```
string s = "Trust no one!";
int i = -1;
do {
   ++i;
   } while (s[i]);
```

for

The *for* statement has the general syntax

```
for ([init]; [test]; [inc]) statement
```

and performs the following steps:

1. If an initializing expression `init` is present, it is executed.
2. If a `test` expression is present, it is executed. If the result is nonzero (or if there is no `test` expression at all), the `statement` is executed.
3. If an `inc` expression is present, it is executed.
4. Finally control returns to step 2.

If there is no break or return inside the `statement`, the `inc` expression (or the `statement`) must affect the value of the `test` expression, or `test` itself must change during evaluation in order to avoid an endless loop.

The initializing expression `init` normally initializes one or more loop counters. It may also define a new variable as a loop counter. The scope of such a variable is valid until the end of the active block.

Example

```
string s = "Trust no one!";
int sum = 0;
for (int i = 0; s[i]; ++i)
    sum += s[i]; // sums up the characters in s
```

if...else

The *if...else* statement has the general syntax

```
if (expression)
   t_statement
[else
   f_statement]
```

The conditional `expression` is evaluated, and if its value is nonzero the `t_statement` is executed. Otherwise the `f_statement` is executed in case there is an `else` clause.

An `else` clause is always matched to the last encountered `if` without an `else`. If this is not what you want, you need to use braces to group the statements, as in

```
if (a == 1) {
   if (b == 1)
      printf("a == 1 and b == 1\n");
   }
else
   printf("a != 1\n");
```

return

A function with a return type other than `void` must contain at least one *return* statement with the syntax

```
return expression;
```

where `expression` must evaluate to a type that is compatible with the function's return

type. The value of `expression` is the value returned by the function.

If the function is of type `void`, a `return` statement without an `expression` can be used to return from the function call.

switch

The *switch* statement has the general syntax

```
switch (sw_exp) {
  case case_exp: case_statement
  ...
  [default: def_statement]
  }
```

and allows for the transfer of control to one of several `case`-labeled statements, depending on the value of `sw_exp` (which must be of integral type).

Any `case_statement` can be labeled by one or more `case` labels. The `case_exp` of each `case` label must evaluate to a constant integer which is unique within it's enclosing `switch` statement.

There can also be at most one `default` label.

After evaluating `sw_exp`, the `case_exp` are checked for a match. If a match is found, control passes to the `case_statement` with the matching `case` label.

If no match is found and there is a `default` label, control passes to `def_statement`. Otherwise none of the statements in the `switch` is executed.

Program execution is not affected when `case` and `default` labels are encountered. Control simply passes through the labels to the following statement.

To stop execution at the end of a group of statements for a particular `case`, use the break statement.

Example

```
string s = "Hello World";
int vowels = 0, others = 0;
for (int i = 0; s[i]; ++i)
    switch (toupper(s[i])) {
      case 'A':
      case 'E':
      case 'I':
      case 'O':
      case 'U': ++vowels;
                break;
      default: ++others;
      }
printf("There are %d vowels in '%s'\n", vowels, s);
```

while

The *while* statement has the general syntax

```
while (condition) statement
```

and executes the `statement` as long as the `condition` expression is not zero.

The `condition` is tested **before** the first possible execution of `statement`, which means that the statement may never be executed if `condition` is initially zero.

If there is no `break` or `return` inside the `statement`, the `statement` must affect the value of the `condition`, or `condition` itself must change during evaluation in order to avoid an endless loop.

Example

```
string s = "Trust no one!";
int i = 0;
while (s[i])
      ++i;
```

Builtins

Builtins are *Constants*, *Variables*, *Functions* and *Statements* that provide additional information and allow for data manipulations.

- Builtin Constants
- Builtin Variables
- Builtin Functions
- Builtin Statements

Builtin Constants

Builtin constants are used to provide information about object parameters, such as maximum recommended name length, flags etc.

Many of the object types have their own **Constants** section which lists the builtin constants for that particular object (see e.g. UL_PIN).

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

`EAGLE_VERSION`	EAGLE program version number (int)
`EAGLE_RELEASE`	EAGLE program release number (int)
`EAGLE_SIGNATURE`	a string containing EAGLE program name, version and copyright information
`REAL_EPSILON`	the minimum positive real number such that `1.0 + REAL_EPSILON != 1.0`
`REAL_MAX`	the largest possible real value
`REAL_MIN`	the smallest possible (positive!) real value

	the smallest representable number is -REAL_MAX
INT_MAX	the largest possible int value
INT_MIN	the smallest possible int value
PI	the value of "pi" (3.14..., real)
usage	a string containing the text from the #usage directive

These builtin constants contain the directory paths defined in the directories dialog, with any of the special variables ($HOME and $EAGLEDIR) replaced by their actual values. Since each path can consist of several directories, these constants are string arrays with an individual directory in each member. The first empty member marks the end of the path:

path_lbr[]	Libraries
path_dru[]	Design Rules
path_ulp[]	User Language Programs
path_scr[]	Scripts
path_cam[]	CAM Jobs
path_epf[]	Projects

When using these constants to build a full file name, you need to use a directory separator, as in

```
string s = path_lbr[0] + '/' + "mylib.lbr";
```

The libraries that are currently in use through the USE command:

```
used_libraries[]
```

Builtin Variables

Builtin variables are used to provide information at runtime.

int argc	number of arguments given to the RUN command
string argv[]	arguments given to the RUN command (argv[0] is the full ULP file name)

Builtin Functions

Builtin functions are used to perform specific tasks, like printing formatted strings, sorting data arrays or the like.

You may also write your own functions and use them to structure your User Language Program.

The builtin functions are grouped into the following categories:

- Character Functions
- File Handling Functions
- Mathematical Functions
- Miscellaneous Functions
- Network Functions
- Printing Functions
- String Functions

- Time Functions
- Object Functions
- XML Functions

Alphabetical reference of all builtin functions:

- abs()
- acos()
- asin()
- atan()
- ceil()
- cfgget()
- cfgset()
- clrgroup()
- country()
- cos()
- exit()
- exp()
- fdlsignature()
- filedir()
- fileerror()
- fileext()
- fileglob()
- filename()
- fileread()
- filesetext()
- filesize()
- filetime()
- floor()
- frac()
- ingroup()
- isalnum()
- isalpha()
- iscntrl()
- isdigit()
- isgraph()
- islower()
- isprint()
- ispunct()
- isspace()
- isupper()
- isxdigit()
- language()
- log()
- log10()
- lookup()
- max()

- xmlattributes()
- xmlelement()
- xmlelements()
- xmltags()
- xmltext()

Character Functions

Character functions are used to manipulate single characters.

The following character functions are available:

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

is...()

Function

Check whether a character falls into a given category.

Syntax

```
int isalnum(char c);
int isalpha(char c);
int iscntrl(char c);
int isdigit(char c);
int isgraph(char c);
int islower(char c);
int isprint(char c);
int ispunct(char c);
int isspace(char c);
int isupper(char c);
int isxdigit(char c);
```

Returns

The `is...` functions return nonzero if the given character falls into the category, zero otherwise.

Character categories

`isalnum`	letters (`A` to `Z` or `a` to `z`) or digits (`0` to `9`)
`isalpha`	letters (`A` to `Z` or `a` to `z`)
`iscntrl`	delete characters or ordinary control characters (`0x7F` or `0x00` to `0x1F`)
`isdigit`	digits (`0` to `9`)
`isgraph`	printing characters (except space)
`islower`	lowercase letters (`a` to `z`)
`isprint`	printing characters (`0x20` to `0x7E`)
`ispunct`	punctuation characters (`iscntrl` or `isspace`)
`isspace`	space, tab, carriage return, new line, vertical tab, or formfeed (`0x09` to `0x0D`, `0x20`)
`isupper`	uppercase letters (`A` to `Z`)
`isxdigit`	hex digits (`0` to `9`, `A` to `F`, `a` to `f`)

Example

```
char c = 'A';
if (isxdigit(c))
   printf("%c is hex\n", c);
else
   printf("%c is not hex\n", c);
```

to...()

Function

Convert a character to upper- or lowercase.

Syntax

```
char tolower(char c);
char toupper(char c);
```

Returns

The `tolower` function returns the converted character if `c` is uppercase. All other characters are returned unchanged.

The `toupper` function returns the converted character if `c` is lowercase. All other characters are returned unchanged.

See also strupr, strlwr

File Handling Functions

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

The following file handling functions are available:

- fileerror()
- fileglob()
- filedir()
- fileext()
- filename()

- fileread()
- filesetext()
- filesize()
- filetime()

See output() for information about how to write into a file.

fileerror()

Function

Returns the status of I/O operations.

Syntax

```
int fileerror();
```

Returns

The `fileerror` function returns `0` if everything is ok.

See also output, printf, fileread

`fileerror` checks the status of any I/O operations that have been performed since the last call to this function and returns `0` if everything was ok. If any of the I/O operations has caused an error, a value other than `0` will be returned.

You should call `fileerror` before any I/O operations to reset any previous error state, and call it again after the I/O operations to see if they were successful.

When `fileerror` returns a value other than `0` (thus indicating an error) a proper error message has already been given to the user.

Example

```
fileerror();
output("file.txt", "wt") {
  printf("Test\n");
  }
if (fileerror())
   exit(1);
```

fileglob()

Function

Perform a directory search.

Syntax

```
int fileglob(string &array[], string pattern);
```

Returns

The `fileglob` function returns the number of entries copied into `array`.

See also dlgFileOpen(), dlgFileSave()

`fileglob` performs a directory search using `pattern`.

`pattern` may contain `'*'` and `'?'` as wildcard characters. If `pattern` ends with a `'/'`,

the contents of the given directory will be returned.

Names in the resulting `array` that end with a `'/'` are directory names.

The `array` is sorted alphabetically, with the directories coming first.

The special entries `'.'` and `'..'` (for the current and parent directories) are never returned in the `array`.

If `pattern` doesn't match, or if you don't have permission to search the given directory, the resulting `array` will be empty.

Note for Windows users

The directory delimiter in the `array` is always a **forward slash**. This makes sure User Language Programs will work platform independently. In the `pattern` the **backslash** (`'\'`) is also treated as a directory delimiter.

Sorting filenames under Windows is done case insensitively.

Example

```
string a[];
int n = fileglob(a, "*.brd");
```

Filename Functions

Function

Split a filename into its separate parts.

Syntax

```
string filedir(string file);
string fileext(string file);
string filename(string file);
string filesetext(string file, string newext);
```

Returns

`filedir` returns the directory of `file` (including the drive letter under Windows).
`fileext` returns the extension of `file`.
`filename` returns the file name of `file` (including the extension).
`filesetext` returns `file` with the extension set to `newext`.

See also Filedata Functions

Example

```
if (board) board(B) {
  output(filesetext(B.name, ".out")) {
    ...
    }
  }
```

Filedata Functions

Function

Gets the timestamp and size of a file.

Syntax

```
int filesize(string filename);
int filetime(string filename);
```

Returns

`filesize` returns the size (in byte) of the given file.

`filetime` returns the timestamp of the given file in a format to be used with the time functions.

See also time, Filename Functions

Example

```
board(B)
  printf("Board: %s\nSize: %d\nTime: %s\n",
         B.name, filesize(B.name),
         t2string(filetime(B.name)));
```

File Input Functions

File input functions are used to read data from files.

The following file input is available:

- fileread()

See output() for information about how to write into a file.

fileread()

Function

Reads data from a file.

Syntax

```
int fileread(dest, string file);
```

Returns

`fileread` returns the number of objects read from the file.

The actual meaning of the return value depends on the type of `dest`.

See also lookup, strsplit, fileerror

If `dest` is a character array, the file will be read as raw binary data and the return value reflects the number of bytes read into the character array (which is equal to the file size).

If `dest` is a string array, the file will be read as a text file (one line per array member) and the return value will be the number of lines read into the string array. Newline characters will be stripped.

If `dest` is a string, the entire file will be read into that string and the return value will be

the length of that string (which is not necessarily equal to the file size, if the operating system stores text files with "cr/lf" instead of a "newline" character).

Example

```
char b[];
int nBytes = fileread(b, "data.bin");
string lines[];
int nLines = fileread(lines, "data.txt");
string text;
int nChars = fileread(text, "data.txt");
```

Mathematical Functions

Mathematical functions are used to perform mathematical operations.

The following mathematical functions are available:

- abs()
- acos()
- asin()
- atan()
- ceil()
- cos()
- exp()
- floor()
- frac()
- log()
- log10()
- max()
- min()
- pow()
- round()
- sin()
- sqrt()
- trunc()
- tan()

Error Messages

If the arguments of a mathematical function call lead to an error, the error message will show the actual values of the arguments. Thus the statements

```
real x = -1.0;
real r = sqrt(2 * x);
```

will lead to the error message

```
Invalid argument in call to 'sqrt(-2)'
```

Absolute, Maximum and Minimum Functions

Function

Absolute, maximum and minimum functions.

Syntax

```
type abs(type x);
type max(type x, type y);
type min(type x, type y);
```

Returns

`abs` returns the absolute value of `x`.
`max` returns the maximum of `x` and `y`.
`min` returns the minimum of `x` and `y`.

The return type of these functions is the same as the (larger) type of the arguments. `type` must be one of `char`, `int` or `real`.

Example

```
real x = 2.567, y = 3.14;
printf("The maximum is %f\n", max(x, y));
```

Rounding Functions

Function

Rounding functions.

Syntax

```
real ceil(real x);
real floor(real x);
real frac(real x);
real round(real x);
real trunc(real x);
```

Returns

`ceil` returns the smallest integer not less than `x`.
`floor` returns the largest integer not greater than `x`.
`frac` returns the fractional part of `x`.
`round` returns `x` rounded to the nearest integer.
`trunc` returns the integer part of `x`.

Example

```
real x = 2.567;
printf("The rounded value of %f is %f\n", x, round(x));
```

Trigonometric Functions

Function

Trigonometric functions.

Syntax

```
real acos(real x);
real asin(real x);
real atan(real x);
real cos(real x);
real sin(real x);
real tan(real x);
```

Returns

`acos` returns the arc cosine of `x`.
`asin` returns the arc sine of `x`.
`atan` returns the arc tangent of `x`.
`cos` returns the cosine of `x`.
`sin` returns the sine of `x`.
`tan` returns the tangent of `x`.

Constants

`PI`	the value of "pi" (3.14...)

Note

Angles are given in radian.

Example

```
real x = PI / 2;
printf("The sine of %f is %f\n", x, sin(x));
```

Exponential Functions

Function

Exponential Functions.

Syntax

```
real exp(real x);
real log(real x);
real log10(real x);
real pow(real x, real y);
real sqrt(real x);
```

Returns

`exp` returns the exponential *e* to the power of `x`.
`log` returns the natural logarithm of `x`.
`log10` returns the base 10 logarithm of `x`.
`pow` returns the value of `x` to the power of `y`.
`sqrt` returns the square root of `x`.

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:

- country()
- exit()
- fdlsignature()
- language()
- lookup()
- palette()
- sort()
- status()
- system()
- Configuration Parameters
- Unit Conversions

Configuration Parameters

Function

Store and retrieve configuration parameters.

Syntax

```
string cfgget(string name[, string default]);
void cfgset(string name, string value);
```

Returns

`cfgget` returns the value of the parameter stored under the given `name`. If no such parameter has been stored, yet, the value of the optional `default` is returned (or an empty string, if no `default` is given).

The `cfgget` function retrieves values that have previously been stored with a call to `cfgset()`.

The `cfgset` function sets the parameter with the given `name` to the given `value`.

The valid characters for `name` are `'A'-'Z'`, `'a'-'z'`, `'0'-'9'`, `'.'` and `'_'`. Parameter names are case sensitive.

The parameters are stored in the user's eaglerc file. To ensure that different User Language Programs don't overwrite each other's parameters in case they use the same parameter names, it is recommended to put the name of the ULP at the beginning of the parameter

name. For example, a ULP named `mytool.ulp` that uses a parameter named `MyParam` could store that parameter under the name

```
mytool.MyParam
```

Because the configuration parameters are stored in the eaglerc file, which also contains all of EAGLE's other user specific parameters, it is also possible to access the EAGLE parameters with `cfgget()` and `cfgset()`. In order to make sure no ULP parameters collide with any EAGLE parameters, the EAGLE parameters must be prefixed with `"EAGLE:"`, as in

```
EAGLE:Option.XrefLabelFormat
```

Note that there is no documentation of all of EAGLE's internal parameters and how they are stored in the eaglerc file. Also, be very careful when changing any of these parameters! As with the eaglerc file itself, you should only manipulate these parameters if you know what you are doing! Some EAGLE parameters may require a restart of EAGLE for changes to take effect.

In the eaglerc file the User Language parameters are stored with the prefix `"ULP:"`. Therefore this prefix may be optionally put in front of User Language parameter names, as in

```
ULP:mytool.MyParam
```

Example

```
string MyParam = cfgget("mytool.MyParam", "SomeDefault");
MyParam = "OtherValue";
cfgset("mytool.MyParam", MyParam);
```

country()

Function

Returns the country code of the system in use.

Syntax

```
string country();
```

Returns

`country` returns a string consisting of two uppercase characters that identifies the country used on the current system. If no such country setting can be determined, the default "US" will be returned.

See also language

Example

```
dlgMessageBox("Your country code is: " + country());
```

exit()

Function

Exits from a User Language Program.

Syntax

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

See also RUN

The `exit` function terminates execution of a User Language Program.
If an integer `result` is given it will be used as the return value of the program.
If a string `command` is given, that command will be executed as if it were entered into the command line immediately after the RUN command. In that case the return value of the ULP is set to `EXIT_SUCCESS`.

Constants

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

fdlsignature()

Function

Calculates a digital signature for Premier Farnell's *Design Link*.

Syntax

```
string fdlsignature(string s, string key);
```

The `fdlsignature` function is used to calculate a digital signature when accessing Premier Farnell's *Design Link* interface.

language()

Function

Returns the language code of the system in use.

Syntax

```
string language();
```

Returns

`language` returns a string consisting of two lowercase characters that identifies the language used on the current system. If no such language setting can be determined, the default "en" will be returned.

See also country

The `language` function can be used to make a ULP use different message string, depending on which language the current system is using.

In the example below all the strings used in the ULP are listed in the string array `I18N[]`,

preceeded by a string containing the various language codes supported by this ULP. Note the `vtab` characters used to separate the individual parts of each string (they are important for the `lookup` function) and the use of the commas to separate the strings. The actual work is done in the function `tr()`, which returns the translated version of the given string. If the original string can't be found in the `I18N` array, or there is no translation for the current language, the original string will be used untranslated.

The first language defined in the `I18N` array must be the one in which the strings used throughout the ULP are written, and should generally be English in order to make the program accessible to the largest number of users.

Example

```
string I18N[] = {
  "en\v"
  "de\v"
  "it\v"
  ,
  "I18N Demo\v"
  "Beispiel für Internationalisierung\v"
  "Esempio per internazionalizzazione\v"
  ,
  "Hello world!\v"
  "Hallo Welt!\v"
  "Ciao mondo!\v"
  ,
  "+Ok\v"
  "+Ok\v"
  "+Approvazione\v"
  ,
  "-Cancel\v"
  "-Abbrechen\v"
  "-Annullamento\v"
  };
int Language = strstr(I18N[0], language()) / 3;
string tr(string s)
{
  string t = lookup(I18N, s, Language, '\v');
  return t ? t : s;
}
dlgDialog(tr("I18N Demo")) {
  dlgHBoxLayout dlgSpacing(350);
  dlgLabel(tr("Hello world!"));
  dlgHBoxLayout {
    dlgPushButton(tr("+Ok")) dlgAccept();
    dlgPushButton(tr("-Cancel")) dlgReject();
    }
  };
```

lookup()

Function

Looks up data in a string array.

Syntax

```
string lookup(string array[], string key, int field_index[,
char separator]);
string lookup(string array[], string key, string field_name[,
char separator]);
```

Returns

`lookup` returns the value of the field identified by `field_index` or `field_name`. If the field doesn't exist, or no string matching `key` is found, an empty string is returned.

See also fileread, strsplit

An `array` that can be used with `lookup()` consists of strings of text, each string representing one data record.

Each data record contains an arbitrary number of fields, which are separated by the character `separator` (default is `'\t'`, the tabulator). The first field in a record is used as the `key` and is numbered `0`.

All records must have unique `key` fields and none of the `key` fields may be empty - otherwise it is undefined which record will be found.

If the first string in the `array` contains a "Header" record (i.e. a record where each field describes its contents), using `lookup` with a `field_name` string automatically determines the index of that field. This allows using the `lookup` function without exactly knowing which field index contains the desired data.
It is up to the user to make sure that the first record actually contains header information.

If the `key` parameter in the call to `lookup()` is an empty string, the first string of the `array` will be used. This allows a program to determine whether there is a header record with the required field names.

If a field contains the `separator` character, that field must be enclosed in double quotes (as in `"abc;def"`, assuming the semicolon (`';'`) is used as separator). The same applies if the field contains double quotes (`"`), in which case the double quotes inside the field have to be doubled (as in `"abc;""def"";ghi"`, which would be `abc;"def";ghi`).
It is best to use the default "tab" separator, which doesn't have these problems (no field can contain a tabulator).

Here's an example data file (`';'` has been used as separator for better readability):

```
Name;Manufacturer;Code;Price
7400;Intel;I-01-234-97;$0.10
68HC12;Motorola;M68HC1201234;$3.50
```

Example

```
string OrderCodes[];
if (fileread(OrderCodes, "ordercodes") > 0) {
   if (lookup(OrderCodes, "", "Code", ';')) {
      schematic(SCH) {
        SCH.parts(P) {
          string OrderCode;
```

```
          // both following statements do exactly the same:
          OrderCode = lookup(OrderCodes, P.device.name, "Code", ';');
          OrderCode = lookup(OrderCodes, P.device.name, 2, ';');
          }
        }
      }
   else
      dlgMessageBox("Missing 'Code' field in file 'ordercodes');
   }
```

palette()

Function

Returns color palette information.

Syntax

```
int palette(int index[, int type]);
```

Returns

The `palette` function returns an integer ARGB value in the form 0xaarrggbb, or the type of the currently used palette (depending on the value of `index`).

The `palette` function returns the ARGB value of the color with the given `index` (which may be in the range 0..PALETTE_ENTRIES-1). If `type` is not given (or is `-1`) the palette assigned to the current editor window will be used. Otherwise `type` specifies which color palette to use (PALETTE_BLACK, PALETTE_WHITE or PALETTE_COLORED).

The special value `-1` for `index` makes the function return the type of the palette that is currently in use by the editor window.

If either `index` or `type` is out of range, an error message will be given and the ULP will be terminated.

Constants

`PALETTE_TYPES`	the number of palette types (3)
`PALETTE_BLACK`	the black background palette (0)
`PALETTE_WHITE`	the white background palette (1)
`PALETTE_COLORED`	the colored background palette (2)
`PALETTE_ENTRIES`	the number of colors per palette (64)

sort()

Function

Sorts an array or a set of arrays.

Syntax

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

The `sort` function either directly sorts a given `array1`, or it sorts a set of arrays (starting with `array2`), in which case `array1` is supposed to be an array of **int**, which will be used

as a pointer array.

In any case, the number argument defines the number of items in the array(s).

Sorting a single array

If the sort function is called with one single array, that array will be sorted directly, as in the following example:

```
string A[];
int n = 0;
A[n++] = "World";
A[n++] = "Hello";
A[n++] = "The truth is out there...";
sort(n, A);
for (int i = 0; i < n; ++i)
    printf(A[i]);
```

Sorting a set of arrays

If the sort function is called with more than one array, the first array must be an array of **int**, while all of the other arrays may be of any array type and hold the data to be sorted. The following example illustrates how the first array will be used as a pointer:

```
numeric string Nets[], Parts[], Instances[], Pins[];
int n = 0;
int index[];
schematic(S) {
  S.nets(N) N.pinrefs(P) {
    Nets[n] = N.name;
    Parts[n] = P.part.name;
    Instances[n] = P.instance.name;
    Pins[n] = P.pin.name;
    ++n;
    }
  sort(n, index, Nets, Parts, Instances, Pins);
  for (int i = 0; i < n; ++i)
      printf("%-8s %-8s %-8s %-8s\n",
             Nets[index[i]], Parts[index[i]],
             Instances[index[i]], Pins[index[i]]);
  }
```

The idea behind this is that one net can have several pins connected to it, and in a netlist you might want to have the net names sorted, and within one net you also want the part names sorted and so on.

Note the use of the keyword numeric in the string arrays. This causes the strings to be sorted in a way that takes into account a numeric part at the end of the strings, which leads to IC1, IC2,... IC9, IC10 instead of the alphabetical order IC1, IC10, IC2,...IC9.

When sorting a set of arrays, the first (index) array must be of type int and need not be initialized. Any contents the index array might have before calling the sort function will be overwritten by the resulting index values.

status()

Function

Displays a status message in the status bar.

Syntax

```
void status(string message);
```

See also dlgMessageBox()

The `status` function displays the given `message` in the status bar of the editor window in which the ULP is running.

system()

Function

Executes an external program.

Syntax

```
int system(string command);
```

Returns

The `system` function returns the exit status of the command. This is typically `0` if everything was ok, and non-zero in case of an error.

The `system` function executes the external program given by the `command` string, and waits until the program ends.

Input/Output redirection

If the external program shall read its standard input from (or write its standard output to) a particular file, input/output needs to be redirected.


On **Linux** and **Mac OS X** this is done by simply adding a `'<'` or `'>'` to the command line, followed by the desired file name, as in

```
system("program < infile > outfile");
```

which runs `program` and makes it read from `infile` and write to `outfile`.

On **Windows** you have to explicitly run a command processor to do this, as in

```
system("cmd.exe /c program < infile > outfile");
```

(on DOS based Windows systems use `command.com` instead of `cmd.exe`).

Background execution

The `system` function waits until the given program has ended. This is useful for programs that only run for a few seconds, or completely take over the user's attention.

If an external program runs for a longer time, and you want the system call to return immediately, without waiting for the program to end, you can simply add an `'&'` to

the command string under **Linux** and **Mac OS X**, as in

```
system("program &");
```

Under Windows you need to explicitly run a command processor to do this, as in

```
system("cmd.exe /c start program");
```

(on DOS based Windows systems use `command.com` instead of `cmd.exe`).

Example

```
int result = system("simulate -f filename");
```

This would call a simulation program, giving it a file which the ULP has just created. Note that `simulate` here is just an example, it is not part of the EAGLE package!

If you want to have control over what system commands are actually executed, you can write a wrapper function that prompts the user for confirmation before executing the command, like

```
int MySystem(string command)
{
  if (dlgMessageBox("!Ok to execute the following command?<p><tt>" + command +
"</tt>", "&Yes", "&No") == 0)
     return system(command);
  return -1;
}
int result = MySystem("simulate -f filename");
```

Unit Conversions

Function

Converts internal units.

Syntax

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

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

EAGLE stores all coordinate and size values as `int` 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));
    }
  }
```

Network Functions

Network functions are used to access remote sites on the Internet.

The following network functions are available:

- neterror()
- netget()
- netpost()

neterror()

Function

Returns the error message of the most recent network function call.

Syntax

```
string neterror(void);
```

Returns

`neterror` returns a textual message describing the error that occurred in the most recent call to a network function.
If no error has occurred, the return value is an empty string.

See also netget, netpost

The `neterror` function should be called after any of the other network functions has returned a negative value, indicating that an error has occurred. The return value of `neterror` is a textual string that can be presented to the user.

Example

```
string Result;
if (netget(Result, "http://www.cadsoft.de/cgi-bin/http-test?see=me&hear=them")
>= 0) {
   // process Result
   }
else
   dlgMessageBox(neterror());
```

netget()

Function

Performs a GET request on the network.

Syntax

```
int netget(dest, string url[, int timeout]);
```

Returns

`netget` returns the number of objects read from the network.
The actual meaning of the return value depends on the type of `dest`.
In case of an error, a negative value is returned and neterror() may be called to display an error message to the user.

See also netpost, neterror, fileread

The `netget` function sends the given `url` to the network and stores the result in the `dest` variable.
If no network activity has occurred for `timeout` seconds, the connection will be terminated. The default timeout is 20 seconds.
The `url` must contain the protocol to use (HTTP, HTTPS or FTP) and can contain name=value pairs of parameters, as in

```
http://www.cadsoft.de/cgi-bin/http-test?see=me&hear=them
ftp://ftp.cadsoft.de/eagle/userfiles/README
```

If a user id and password is required to access a remote site, these can be given as

```
https://userid:password@www.secret-site.com/...
```

If `dest` is a character array, the result will be treated as raw binary data and the return value reflects the number of bytes stored in the character array.

If `dest` is a string array, the result will be treated as text data (one line per array member) and the return value will be the number of lines stored in the string array. Newline characters will be stripped.

If `dest` is a string, the result will be stored in that string and the return value will be the length of the string. Note that in case of binary data the result is truncated at the first occurrence of a byte with the value 0x00.

If you need to use a proxy to access the Internet with HTTP or HTTPS, you can set that up in the "Configure" dialog under "Help/Check for Update" in the Control Panel.

Example

```
string Result;
if (netget(Result, "http://www.cadsoft.de/cgi-bin/http-test?see=me&hear=them")
>= 0) {
   // process Result
   }
else
   dlgMessageBox(neterror());
```

netpost()

Function

Performs a POST request on the network.

Syntax

```
int netpost(dest, string url, string data[, int timeout]);
```

Returns

`netpost` returns the number of objects read from the network.
The actual meaning of the return value depends on the type of `dest`.
In case of an error, a negative value is returned and neterror() may be called to display an error message to the user.

See also netget, neterror, fileread

The `netpost` function sends the given `data` to the given `url` on the network and stores the result in the `dest` variable.
If no network activity has occurred for `timeout` seconds, the connection will be terminated. The default timeout is 20 seconds.
The `url` must contain the protocol to use (HTTP or HTTPS).

If a user id and password is required to access a remote site, these can be given as

```
https://userid:password@www.secret-site.com/...
```

If `dest` is a character array, the result will be treated as raw binary data and the return value reflects the number of bytes stored in the character array.

If `dest` is a string array, the result will be treated as text data (one line per array member) and the return value will be the number of lines stored in the string array. Newline characters will be stripped.

If `dest` is a string, the result will be stored in that string and the return value will be the length of the string. Note that in case of binary data the result is truncated at the first occurrence of a byte with the value 0x00.

If you need to use a proxy to access the Internet with HTTP or HTTPS, you can set that up in the "Configure" dialog under "Help/Check for Update" in the Control Panel.

Example

```
string Data = "see=me\nhear=them";
string Result;
if (netpost(Result, "http://www.cadsoft.de/cgi-bin/http-test", Data) >= 0) {
   // process Result
   }
else
   dlgMessageBox(neterror());
```

Printing Functions

Printing functions are used to print formatted strings.

The following printing functions are available:

- printf()
- sprintf()

printf()

Function

Writes formatted output to a file.

Syntax

```
int printf(string format[, argument, ...]);
```

Returns

The `printf` function returns the number of characters written to the file that has been opened by the most recent output statement.

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

See also sprintf, output, fileerror

Format string

The format string controls how the arguments will be converted, formatted and printed. There must be exactly as many arguments as necessary for the format. The number and type of arguments will be checked against the format, and any mismatch will lead to an error message.

The format string contains two types of objects - *plain characters* and *format specifiers*:

- Plain characters are simply copied verbatim to the output
- Format specifiers fetch arguments from the argument list and apply formatting to them

Format specifiers

A format specifier has the following form:

```
% [flags] [width] [.prec] type
```

Each format specification begins with the percent character (`%`). After the `%` comes the following, in this order:

- an optional sequence of flag characters, `[flags]`
- an optional width specifier, `[width]`
- an optional precision specifier, `[.prec]`
- the conversion type character, `type`

Conversion type characters

`d` **signed** decimal **int**
`o` **unsigned** octal **int**
`u` **unsigned** decimal **int**
`x` **unsigned** hexadecimal **int** (with **a**, **b**,...)
`X` **unsigned** hexadecimal **int** (with **A**, **B**,...)
`f` **signed real** value of the form `[-]dddd.dddd`
`e` **signed real** value of the form `[-]d.ddde[±]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 zeros.
`*`	The argument list supplies the width specifier, which must precede the actual argument being formatted.

Precision specifiers

A precision specifier always begins with a period (`.`) to separate it from any preceding width specifier. Then, like width, precision is specified either directly through a decimal digit string, or indirectly, through an asterisk (`*`). If you use an asterisk for the precision specifier, the 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, the `printf` function can print binary zero characters (0x00).

```
char c = 0x00;
printf("%c", c);
```

Example

```
int i = 42;
real r = 3.14;
char c = 'A';
string s = "Hello";
printf("Integer: %8d\n", i);
printf("Hex:     %8X\n", i);
printf("Real:    %8f\n", r);
printf("Char:    %-8c\n", c);
printf("String:  %-8s\n", s);
```

sprintf()

Function

Writes formatted output into a string.

Syntax

```
int sprintf(string result, string format[, argument, ...]);
```

Returns

The `sprintf` function returns the number of characters written into the `result` string.

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

See also printf

Format string

See printf.

Binary zero characters

Note that `sprintf` can not return strings with embedded binary zero characters (0x00). If the resulting string contains a binary zero character, any characters following that zero character will be dropped. Use printf if you need to output binary data.

Example

```
string result;
int number = 42;
sprintf(result, "The number is %d", number);
```

String Functions

String functions are used to manipulate character strings.

The following string functions are available:

- strchr()
- strjoin()
- strlen()
- strlwr()
- strrchr()
- strrstr()
- strsplit()
- strstr()
- strsub()
- strtod()
- strtol()
- strupr()
- strxstr()

strchr()

Function

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

Syntax

```
int strchr(string s, char c[, int index]);
```

Returns

The `strchr` function returns the integer offset of the character in the string, or `-1` if the character does not occur in the string.

See also strrchr, strstr

If `index` is given, the search starts at that position. Negative values are counted from the end of the string.

Example

```
string s = "This is a string";
char c = 'a';
int pos = strchr(s, c);
if (pos >= 0)
   printf("The character %c is at position %d\n", c, pos);
else
   printf("The character was not found\n");
```

strjoin()

Function

Joins a string array to form a single string.

Syntax

```
string strjoin(string array[], char separator);
```

Returns

The `strjoin` function returns the combined entries of `array`.

See also strsplit, lookup, fileread

`strjoin` joins all entries in `array`, delimited by the given `separator` and returns the resulting string.

If `separator` is the newline character (`'\n'`) the resulting string will be terminated with a newline character. This is done to have a text file that consists of N lines (each of which is terminated with a newline) and is read in with the fileread() function and split into an array of N strings to be joined to the original string as read from the file.

Example

```
string a[] = { "Field 1", "Field 2", "Field 3" };
string s = strjoin(a, ':');
```

strlen()

Function

Calculates the length of a string.

Syntax

```
int strlen(string s);
```

Returns

The `strlen` function returns the number of characters in the string.

Example

```
string s = "This is a string";
int l = strlen(s);
printf("The string is %d characters long\n", l);
```

strlwr()

Function

Converts uppercase letters in a string to lowercase.

Syntax

```
string strlwr(string s);
```

Returns

The `strlwr` function returns the modified string. The original string (given as parameter) is not changed.

See also strupr, tolower

Example

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

strrchr()

Function

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

Syntax

```
int strrchr(string s, char c[, int index]);
```

Returns

The `strrchr` function returns the integer offset of the character in the string, or `-1` if the character does not occur in the string.

See also strchr, strrstr

If `index` is given, the search starts at that position. Negative values are counted from the end of the string.

Example

```
string s = "This is a string";
char c = 'a';
int pos = strrchr(s, c);
if (pos >= 0)
   printf("The character %c is at position %d\n", c, pos);
else
   printf("The character was not found\n");
```

strrstr()

Function

Scans a string for the last occurrence of a given substring.

Syntax

```
int strrstr(string s1, string s2[, int index]);
```

Returns

The `strrstr` function returns the integer offset of the first character of s2 in s1, or `-1` if the substring does not occur in the string.

See also strstr, strrchr

If `index` is given, the search starts at that position. Negative values are counted from the end of the string.

Example

```
string s1 = "This is a string", s2 = "is a";
int pos = strrstr(s1, s2);
if (pos >= 0)
   printf("The substring starts at %d\n", pos);
else
   printf("The substring was not found\n");
```

strsplit()

Function

Splits a string into separate fields.

Syntax

```
int strsplit(string &array[], string s, char separator);
```

Returns

The `strsplit` function returns the number of entries copied into `array`.

See also strjoin, lookup, fileread

`strsplit` splits the string `s` at the given `separator` and stores the resulting fields in the `array`.

If `separator` is the newline character (`'\n'`) the last field will be silently dropped if it is empty. This is done to have a text file that consists of N lines (each of which is terminated with a newline) and is read in with the fileread() function to be split into an array of N strings. With any other `separator` an empty field at the end of the string will count, so `"a:b:c:"` will result in 4 fields, the last of which is empty.

Example

```
string a[];
int n = strsplit(a, "Field 1:Field 2:Field 3", ':');
```

strstr()

Function

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

Syntax

```
int strstr(string s1, string s2[, int index]);
```

Returns

The `strstr` function returns the integer offset of the first character of s2 in s1, or `-1` if the substring does not occur in the string.

See also strrstr, strchr, strxstr

If `index` is given, the search starts at that position. Negative values are counted from the end of the string.

Example

```
string s1 = "This is a string", s2 = "is a";
int pos = strstr(s1, s2);
if (pos >= 0)
   printf("The substring starts at %d\n", pos);
else
   printf("The substring was not found\n");
```

strsub()

Function

Extracts a substring from a string.

Syntax

```
string strsub(string s, int start[, int length]);
```

Returns

The `strsub` function returns the substring indicated by the `start` and `length` value.

The value for `length` must be positive, otherwise an empty string will be returned. If `length` is ommitted, the rest of the string (beginning at `start`) is returned.

If `start` points to a position outside the string, an empty string is returned.

Example

```
string s = "This is a string";
string t = strsub(s, 4, 7);
printf("The extracted substring is: %s\n", t);
```

strtod()

Function

Converts a string to a real value.

Syntax

```
real strtod(string s);
```

Returns

The `strtod` function returns the numerical representation of the given string as a `real` value. Conversion ends at the first character that does not fit into the format of a real constant. If an error occurs during conversion of the string `0.0` will be returned.

See also strtol

Example

```
string s = "3.1415";
real r = strtod(s);
printf("The value is %f\n", r);
```

strtol()

Function

Converts a string to an integer value.

Syntax

```
int strtol(string s);
```

Returns

The `strtol` function returns the numerical representation of the given string as an `int` value. Conversion ends at the first character that does not fit into the format of an integer constant. If an error occurs during conversion of the string `0` will be returned.

See also strtod

Example

```
string s = "1234";
int i = strtol(s);
printf("The value is %d\n", i);
```

strupr()

Function

Converts lowercase letters in a string to uppercase.

Syntax

```
string strupr(string s);
```

Returns

The `strupr` function returns the modified string. The original string (given as parameter) is not changed.

See also strlwr, toupper

Example

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

strxstr()

Function

Scans a string for the first occurrence of a given regular expression.

Syntax

```
int strxstr(string s1, string s2[, int index[, int &length]]);
```

Returns

The `strxstr` function returns the integer offset of the substring in s1 that matches the regular expression in s2, or `-1` if the regular expression does not match in the string.

See also strstr, strchr, strrstr

If `index` is given, the search starts at that position. Negative values are counted from the end of the string.

If `length` is given, the actual length of the matching substring is returned in that variable.

Regular expressions allow you to find a pattern within a text string. For instance, the regular expression "i.*a" would find a sequence of characters that starts with an 'i', followed by any character ('.') any number of times ('*'), and ends with an 'a'. It would match on "is a" as well as "is this a" or "ia".
Details on regular expressions can be found, for instance, in the book *Mastering Regular Expressions* by Jeffrey E. F. Friedl.

Example

```
string s1 = "This is a string", s2 = "i.*a";
int len = 0;
int pos = strxstr(s1, s2, 0, len);
if (pos >= 0)
   printf("The substring starts at %d and is %d charcaters long\n", pos, len);
else
   printf("The substring was not found\n");
```

Time Functions

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

The following time functions are available:

- t2day()
- t2dayofweek()

- t2hour()
- t2minute()
- t2month()
- t2second()
- t2string()
- t2year()
- time()
- timems()

time()

Function

Gets the current system time.

Syntax

```
int time(void);
```

Returns

The `time` function returns the current system time as the number of seconds elapsed since a system dependent reference date.

See also Time Conversions, filetime, timems()

Example

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

timems()

Function

Gets the number of milliseconds since the start of the ULP.

Syntax

```
int timems(void);
```

Returns

The `timems` function returns the number of milliseconds since the start of the ULP.

After 86400000 milliseconds (i.e. every 24 hours), the value starts at 0 again.

See also time

Example

```
int elapsed = timems();
```

Time Conversions

Function

Convert a time value to day, month, year etc.

Syntax

```
int t2day(int t);
int t2dayofweek(int t);
int t2hour(int t);
int t2minute(int t);
int t2month(int t);
int t2second(int t);
int t2year(int t);

string t2string(int t[, string format]);
```

Returns

`t2day` returns the day of the month (`1`..`31`)
`t2dayofweek` returns the day of the week (`0`=sunday..`6`)
`t2hour` returns the hour (`0`..`23`)
`t2minute` returns the minute (`0`..`59`)
`t2month` returns the month (`0`..`11`)
`t2second` returns the second (`0`..`59`)
`t2year` returns the year (including century!)
`t2string` returns a formatted string containing date and time

See also time

The `t2string` function without the optional `format` parameter converts the given time `t` into a country specific string in local time.

If `t2string` is called with a `format` string, that format is used to determine what the result should look like.

The following expressions can be used in a `format` string:

d	the day as a number without a leading zero (1 to 31)
dd	the day as a number with a leading zero (01 to 31)
ddd	the abbreviated localized day name (e.g. "Mon" to "Sun")
dddd	the long localized day name (e.g. "Monday" to "Sunday")
M	the month as a number without a leading zero (1-12)
MM	the month as a number with a leading zero (01-12)
MMM	the abbreviated localized month name (e.g. "Jan" to "Dec")
MMMM	the long localized month name (e.g. "January" to "December")
yy	the year as a two digit number (00-99)
yyyy	the year as a four digit number
h	the hour without a leading zero (0 to 23 or 1 to 12 if AM/PM display)
hh	the hour with a leading zero (00 to 23 or 01 to 12 if AM/PM display)
m	the minute without a leading zero (0 to 59)
mm	the minute with a leading zero (00 to 59)
s	the second without a leading zero (0 to 59)
ss	the second with a leading zero (00 to 59)
z	the milliseconds without leading zeros (always 0, since the given time only has a one second resolution)
zzz	the milliseconds with leading zeros (always 000, since the given time only has a

one second resolution)
AP use AM/PM display (*AP* will be replaced by either "AM" or "PM")
ap use am/pm display (*ap* will be replaced by either "am" or "pm")
U display the given time as UTC (must be the first character; default is local time)
All other characters will be copied "as is". Any sequence of characters that are enclosed in singlequotes will be treated as text and not be used as an expression. Two consecutive single quotes (') are replaced by a single quote in the output.

Example

```
int t = time();
printf("It is now %02d:%02d:%02d\n",
       t2hour(t), t2minute(t), t2second(t));
printf("ISO time is %s\n", t2string(t, "Uyyyy-MM-dd hh:mm:ss"));
```

Object Functions

Object functions are used to access common information about objects.

The following object functions are available:

- clrgroup()
- ingroup()
- setgroup()

clrgroup()

Function

Clears the group flags of an object.

Syntax

```
void clrgroup(object);
```

See also ingroup(), setgroup(), GROUP command

The `clrgroup()` function clears the group flags of the given object, so that it is no longer part of the previously defined group.

When applied to an object that contains other objects (like a UL_BOARD or UL_NET) the group flags of all contained objects are cleared recursively.

Example

```
board(B) {
  B.elements(E)
    clrgroup(E);
  }
```

ingroup()

Function

Checks whether an object is in the group.

Syntax

```
int ingroup(object);
```

Returns

The `ingroup` function returns a non-zero value if the given object is in the group.

See also clrgroup(), setgroup(), GROUP command

If a group has been defined in the editor, the `ingroup()` function can be used to check whether a particular object is part of the group.

Objects with a single coordinate that are individually selectable in the current drawing (like UL_TEXT, UL_VIA, UL_CIRCLE etc.) return a non-zero value in a call to `ingroup()` if that coordinate is within the defined group.

A UL_WIRE returns 0, 1, 2 or 3, depending on whether none, the first, the second or both of its end points are in the group.

A UL_RECTANGLE and UL_FRAME returns a non-zero value if one or more of its corners are in the group. The value has bit 0 set for the upper right corner, bit 1 for the upper left, bit 2 for the bottom left, and bit 3 for the bottom right corner.

Objects that have no coordinates (like UL_NET, UL_SEGMENT, UL_SIGNAL etc.) return a non-zero value if one or more of the objects within them are in the group.

UL_CONTACTREF and UL_PINREF, though not having coordinates of their own, return a non-zero value if the referenced UL_CONTACT or UL_PIN, respectively, is within the group.

Example

```
output("group.txt") {
  board(B) {
    B.elements(E) {
      if (ingroup(E))
         printf("Element %s is in the group\n", E.name);
      }
    }
  }
```

setgroup()

Function

Sets the group flags of an object.

Syntax

```
void setgroup(object[, int flags]);
```

See also clrgroup(), ingroup(), GROUP command

The `setgroup()` function sets the group flags of the given object, so that it becomes part of the group.

If no `flags` are given, the object is added to the group as a whole (i.e. all of its selection points, in case it has more than one).

If `flags` has a non-zero value, only the group flags of the given points of the object are set. For a UL_WIRE this means that `'1'` sets the group flag of the first point, `'2'` that of the second point, and `'3'` sets both. Any previously set group flags remain unchanged by a call to `setgroup()`.

When applied to an object that contains other objects (like a UL_BOARD or UL_NET) the group flags of all contained objects are set recursively.

Example

```
board(B) {
  B.elements(E)
    setgroup(E);
  }
```

XML Functions

XML functions are used to process XML (*Extensible Markup Language*) data.

The following XML functions are available:

- xmlattribute()
- xmlattributes()
- xmlelement()
- xmlelements()
- xmltags()
- xmltext()

xmlattribute(), xmlattributes()

Function

Extract the attributes of an XML tag.

Syntax

```
string xmlattribute(string xml, string tag, string attribute);
int xmlattributes(string &array[], string xml, string tag);
```

See also xmlelement(), xmltags(), xmltext()

The `xmlattribute` function returns the value of the given `attribute` from the given `tag` within the given `xml` code. If an attribute appears more than once in the same tag, the value of its last occurrence is taken.

The `xmlattributes` function stores the names of all attributes from the given `tag` within the given `xml` code in the `array` and returns the number of attributes found. If an attribute appears more than once in the same tag, its name appears only once in the `array`.

The `tag` is given in the form of a *path*.

If the given `xml` code contains an error, the result of any XML function is empty, and a warning dialog is presented to the user, giving information about where in the ULP and

XML code the error occurred. Note that the line and column number within the XML code refers to the actual string given to this function as the `xml` parameter.

Example

```
// XML contains the following data:
<root>
  <body abc="def" xyz="123">
    ...
  </body>
</root>
//
string s[];
int n = xmlattributes(s, XML, "root/body");
Result: { "abc", "xyz" }
string s = xmlattribute(XML, "root/body", "xyz");
Result: "123"
```

xmlelement(), xmlelements()

Function

Extract elements from an XML code.

Syntax

```
string xmlelement(string xml, string tag);
int xmlelements(string &array[], string xml, string tag);
```

See also xmltags(), xmlattribute(), xmltext()

The `xmlelement` function returns the complete XML element of the given `tag` within the given `xml` code. The result still contains the element's outer XML tag, and can thus be used for further processing with the other XML functions. Any whitespace within plain text parts of the element is retained. The overall formatting of the XML tags within the element may be different than the original `xml` code, though.
If there is more than one occurrence of `tag` within `xml`, the first one will be returned. Use `xmlelements` if you want to get all occurrences.

The `xmlelements` function works just like `xmlelement`, but returns all occurrences of elements with the given `tag`. The return value is the number of elements stored in the `array`.

The `tag` is given in the form of a *path*.

If the given `xml` code contains an error, the result of any XML function is empty, and a warning dialog is presented to the user, giving information about where in the ULP and XML code the error occurred. Note that the line and column number within the XML code refers to the actual string given to this function as the `xml` parameter.

Example

```
// XML contains the following data:
<root>
  <body>
```

```
    <contents>
      <string>Some text 1</string>
      <any>anything 1</any>
    </contents>
    <contents>
      <string>Some text 2</string>
      <any>anything 2</any>
    </contents>
    <appendix>
      <string>Some text 3</string>
    </appendix>
  </body>
</root>
//
string s = xmlelement(XML, "root/body/appendix");
Result: " \n  Some text 3\n \n"
string s[];
int n = xmlelements(s, XML, "root/body/contents");
Result: { " <contents>\n  <string>Some text 1</string>\n  <any>anything
1</any>\n </contents>\n",
          " <contents>\n  <string>Some text 2</string>\n  <any>anything
2</any>\n </contents>\n"
        }
```

xmltags()

Function

Extract the list of tag names within an XML code.

Syntax

```
int xmltags(string &array[], string xml, string tag);
```

See also xmlelement(), xmlattribute(), xmltext()

The `xmltags` function returns the names of all the tags on the top level of the given `tag` within the given `xml` code. The return value is the number of tag names stored in the `array`.

Each tag name is returned only once, even if it appears several times in the XML code.

The `tag` is given in the form of a *path*.

If the given `xml` code contains an error, the result of any XML function is empty, and a warning dialog is presented to the user, giving information about where in the ULP and XML code the error occurred. Note that the line and column number within the XML code refers to the actual string given to this function as the `xml` parameter.

Example

```
// XML contains the following data:
<root>
  <body>
    <contents>
      <string>Some text 1</string>
      <any>anything 1</any>
    </contents>
```

```
    <contents>
      <string>Some text 2</string>
      <any>anything 2</any>
    </contents>
    <appendix>
      <string>Some text 3</string>
    </appendix>
  </body>
</root>
//
string s[];
int n = xmltags(s, XML, "root/body");
Result: { "contents", "appendix" }
int n = xmltags(s, XML, "");
Result: "root"
```

xmltext()

Function

Extract the textual data of an XML element.

Syntax

```
string xmltext(string xml, string tag);
```

See also xmlelement(), xmlattribute(), xmltags()

The `xmltext` function returns the textual data from the given `tag` within the given `xml` code.

Any tags within the text are stripped, whitespace (including newline characters) is retained.

The `tag` is given in the form of a *path*.

If the given `xml` code contains an error, the result of any XML function is empty, and a warning dialog is presented to the user, giving information about where in the ULP and XML code the error occurred. Note that the line and column number within the XML code refers to the actual string given to this function as the `xml` parameter.

Example

```
// XML contains the following data:
<root>
  <body>
    Some <b>text</b>.
  </body>
</root>
//
string s = xmltext(XML, "root/body");
Result: "\n    Some text.\n  "
```

Builtin Statements

Builtin statements are generally used to open a certain context in which data structures 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()
- deviceset()
- library()
- output()
- package()
- schematic()
- sheet()
- symbol()

board()

Function

Opens a board context.

Syntax

```
board(identifier) statement
```

See also schematic, library

The `board` statement opens a board context if the current editor window contains a board drawing. A variable of type UL_BOARD is created and is given the name indicated by `identifier`.

Once the board context is successfully opened and a board variable has been created, the `statement` is executed. Within the scope of the `statement` the board variable can be accessed to retrieve further data from the board.

If the current editor window does not contain a board drawing, an error message is given and the ULP is terminated.

Check if there is a board

By using the `board` statement without an argument you can check if the current editor window contains a board drawing. In that case, `board` behaves like an integer constant, returning `1` if there is a board drawing in the current editor window, and `0` otherwise.

Accessing board from a schematic

If the current editor window contains a schematic drawing, you can still access that schematic's board by preceding the `board` statement with the prefix `project`, as in

```
project.board(B) { ... }
```

This will open a board context regardless whether the current editor window contains a board or a schematic drawing. However, there must be an editor window containing that board somewhere on the desktop!

Example

```
if (board)
   board(B) {
     B.elements(E)
       printf("Element: %s\n", E.name);
     }
```

deviceset()

Function

Opens a device set context.

Syntax

```
deviceset(identifier) statement
```

See also package, symbol, library

The `deviceset` statement opens a device set context if the current editor window contains a device drawing. A variable of type UL_DEVICESET is created and is given the name indicated by `identifier`.

Once the device set context is successfully opened and a device set variable has been created, the `statement` is executed. Within the scope of the `statement` the device set variable can be accessed to retrieve further data from the device set.

If the current editor window does not contain a device drawing, an error message is given and the ULP is terminated.

Check if there is a device set

By using the `deviceset` statement without an argument you can check if the current editor window contains a device drawing. In that case, `deviceset` behaves like an integer constant, returning `1` if there is a device drawing in the current editor window, and `0` otherwise.

Example

```
if (deviceset)
   deviceset(D) {
     D.gates(G)
       printf("Gate: %s\n", G.name);
```

```
}
```

library()

Function

Opens a library context.

Syntax

```
library(identifier) statement
```

See also board, schematic, deviceset, package, symbol

The `library` statement opens a library context if the current editor window contains a library drawing. A variable of type UL_LIBRARY is created and is given the name indicated by `identifier`.

Once the library context is successfully opened and a library variable has been created, the `statement` is executed. Within the scope of the `statement` the library variable can be accessed to retrieve further data from the library.

If the current editor window does not contain a library drawing, an error message is given and the ULP is terminated.

Check if there is a library

By using the `library` statement without an argument you can check if the current editor window contains a library drawing. In that case, `library` behaves like an integer constant, returning `1` if there is a library drawing in the current editor window, and `0` otherwise.

Example

```
if (library)
   library(L) {
     L.devices(D)
       printf("Device: %s\n", D.name);
     }
```

output()

Function

Opens an output file for subsequent printf() calls.

Syntax

```
output(string filename[, string mode]) statement
```

See also printf, fileerror

The `output` statement opens a file with the given `filename` and `mode` for output through subsequent printf() calls. If the file has been successfully opened, the `statement` is executed, and after that the file is closed.

If the file cannot be opened, an error message is given and execution of the ULP is

terminated.

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

File Modes

The `mode` parameter defines how the output file is to be opened. If no `mode` parameter is given, the default is `"wt"`.

- `a` append to an existing file, or create a new file if it does not exist
- `w` create a new file (overwriting an existing file)
- `t` open file in text mode
- `b` open file in binary mode
- `D` delete this file when ending the EAGLE session (only works together with `w`)
- `F` force using this file name (normally *.brd, *.sch and *.lbr are rejected)

Mode characters may appear in any order and combination. However, only the last one of `a` and `w` or `t` and `b`, respectively, is significant. For example a mode of `"abtw"` would open a file for textual write, which would be the same as `"wt"`.

Nested Output statements

`output` statements can be nested, as long as there are enough file handles available, and provided that no two active `output` statements access the **same** file.

Example

```
void PrintText(string s)
{
  printf("This also goes into the file: %s\n", s);
}
output("file.txt", "wt") {
  printf("Directly printed\n");
  PrintText("via function call");
  }
```

package()

Function

Opens a package context.

Syntax

```
package(identifier) statement
```

See also library, deviceset, symbol

The `package` statement opens a package context if the current editor window contains a package drawing. A variable of type UL_PACKAGE is created and is given the name indicated by `identifier`.

Once the package context is successfully opened and a package variable has been created,

the `statement` is executed. Within the scope of the `statement` the package variable can be accessed to retrieve further data from the package.

If the current editor window does not contain a package drawing, an error message is given and the ULP is terminated.

Check if there is a package

By using the `package` statement without an argument you can check if the current editor window contains a package drawing. In that case, `package` behaves like an integer constant, returning `1` if there is a package drawing in the current editor window, and `0` otherwise.

Example

```
if (package)
   package(P) {
     P.contacts(C)
       printf("Contact: %s\n", C.name);
     }
```

schematic()

Function

Opens a schematic context.

Syntax

```
schematic(identifier) statement
```

See also board, library, sheet

The `schematic` statement opens a schematic context if the current editor window contains a schematic drawing. A variable of type UL_SCHEMATIC is created and is given the name indicated by `identifier`.

Once the schematic context is successfully opened and a schematic variable has been created, the `statement` is executed. Within the scope of the `statement` the schematic variable can be accessed to retrieve further data from the schematic.

If the current editor window does not contain a schematic drawing, an error message is given and the ULP is terminated.

Check if there is a schematic

By using the `schematic` statement without an argument you can check if the current editor window contains a schematic drawing. In that case, `schematic` behaves like an integer constant, returning `1` if there is a schematic drawing in the current editor window, and `0` otherwise.

Accessing schematic from a board

If the current editor window contains a board drawing, you can still access that board's schematic by preceding the `schematic` statement with the prefix `project`, as in

```
project.schematic(S) { ... }
```

This will open a schematic context regardless whether the current editor window contains a schematic or a board drawing. However, there must be an editor window containing that schematic somewhere on the desktop!

Access the current Sheet

Use the `sheet` statement to directly access the currently loaded sheet.

Example

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

sheet()

Function

Opens a sheet context.

Syntax

```
sheet(identifier) statement
```

See also schematic

The `sheet` statement opens a sheet context if the current editor window contains a sheet drawing. A variable of type UL_SHEET is created and is given the name indicated by `identifier`.

Once the sheet context is successfully opened and a sheet variable has been created, the `statement` is executed. Within the scope of the `statement` the sheet variable can be accessed to retrieve further data from the sheet.

If the current editor window does not contain a sheet drawing, an error message is given and the ULP is terminated.

Check if there is a sheet

By using the `sheet` statement without an argument you can check if the current editor window contains a sheet drawing. In that case, `sheet` behaves like an integer constant, returning `1` if there is a sheet drawing in the current editor window, and `0` otherwise.

Example

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

symbol()

Function

Opens a symbol context.

Syntax

```
symbol(identifier) statement
```

See also library, deviceset, package

The `symbol` statement opens a symbol context if the current editor window contains a symbol drawing. A variable of type UL_SYMBOL is created and is given the name indicated by `identifier`.

Once the symbol context is successfully opened and a symbol variable has been created, the `statement` is executed. Within the scope of the `statement` the symbol variable can be accessed to retrieve further data from the symbol.

If the current editor window does not contain a symbol drawing, an error message is given and the ULP is terminated.

Check if there is a symbol

By using the `symbol` statement without an argument you can check if the current editor window contains a symbol drawing. In that case, `symbol` behaves like an integer constant, returning `1` if there is a symbol drawing in the current editor window, and `0` otherwise.

Example

```
if (symbol)
   symbol(S) {
     S.pins(P)
       printf("Pin: %s\n", P.name);
     }
```

Dialogs

User Language Dialogs allow you to define your own frontend to a User Language Program.

The following sections describe User Language Dialogs in detail:

Predefined Dialogs	describes the ready to use standard dialogs
Dialog Objects	defines the objects that can be used in a dialog
Layout Information	explains how to define the location of objects within a dialog

Predefined Dialogs

Predefined Dialogs implement the typical standard dialogs that are frequently used for selecting file names or issuing error messages.

The following predefined dialogs are available:

- dlgDirectory()
- dlgFileOpen()
- dlgFileSave()
- dlgMessageBox()

See Dialog Objects for information on how to define your own complex user dialogs.

dlgDirectory()

Function

Displays a directory dialog.

Syntax

```
string dlgDirectory(string Title[, string Start])
```

Returns

The `dlgDirectory` function returns the full pathname of the selected directory. If the user has canceled the dialog, the result will be an empty string.

See also dlgFileOpen

The `dlgDirectory` function displays a directory dialog from which the user can select a directory.

`Title` will be used as the dialog's title.

If `Start` is not empty, it will be used as the starting point for the `dlgDirectory`.

Example

```
string dirName;
dirName = dlgDirectory("Select a directory", "");
```

dlgFileOpen(), dlgFileSave()

Function

Displays a file dialog.

Syntax

```
string dlgFileOpen(string Title[, string Start[, string
Filter]])
string dlgFileSave(string Title[, string Start[, string
Filter]])
```

Returns

The dlgFileOpen and dlgFileSave functions return the full pathname of the selected file.
If the user has canceled the dialog, the result will be an empty string.

See also dlgDirectory

The dlgFileOpen and dlgFileSave functions display a file dialog from which the user can select a file.

Title will be used as the dialog's title.

If Start is not empty, it will be used as the starting point for the file dialog. Otherwise the current directory will be used.

Only files matching Filter will be displayed. If Filter is empty, all files will be displayed.

Filter can be either a simple wildcard (as in "*.brd"), a list of wildcards (as in "*.bmp *.jpg") or may even contain descriptive text, as in "Bitmap files (*.bmp)". If the "File type" combo box of the file dialog shall contain several entries, they have to be separated by double semicolons, as in "Bitmap files (*.bmp);;Other images (*.jpg *.png)".

Example

```
string fileName;
fileName = dlgFileOpen("Select a file", "", "*.brd");
```

dlgMessageBox()

Function

Displays a message box.

Syntax

int dlgMessageBox(string Message[, *button_list*])

Returns

The dlgMessageBox function returns the index of the button the user has selected. The first button in button_list has index 0.

See also status()

The dlgMessageBox function displays the given Message in a modal dialog and waits until the user selects one of the buttons defined in button_list.

If Message contains any HTML tags, the characters '<', '>' and '&' must be given as "<", ">" and "&", respectively, if they shall be displayed as such.

button_list is an optional list of comma separated strings, which defines the set of buttons that will be displayed at the bottom of the message box.
A maximum of three buttons can be defined. If no button_list is given, it defaults to "OK".

The first button in `button_list` will become the default button (which will be selected if the user hits ENTER), and the last button in the list will become the "cancel button", which is selected if the user hits ESCape or closes the message box. You can make a different button the default button by starting its name with a `'+'`, and you can make a different button the cancel button by starting its name with a `'-'`. To start a button text with an actual `'+'` or `'-'` it has to be escaped.

If a button text contains an `'&'`, the character following the ampersand will become a hotkey, and when the user hits the corresponding key, that button will be selected. To have an actual `'&'` character in the text it has to be escaped.

The message box can be given an icon by setting the first character of `Message` to
`';'` - for an *Information*
`'!'` - for a *Warning*
`':'` - for an *Error*
If, however, the `Message` shall begin with one of these characters, it has to be escaped.

On **Mac OS X** only the character `':'` will actually result in showing an icon. All others are ignored.

Example

```
if (dlgMessageBox("!Are you sure?", "&Yes", "&No") == 0) {
   // let's do it!
   }
```

Dialog Objects

A User Language Dialog is built from the following *Dialog Objects*:

dlgCell	a grid cell context
dlgCheckBox	a checkbox
dlgComboBox	a combo box selection field
dlgDialog	the basic container of any dialog
dlgGridLayout	a grid based layout context
dlgGroup	a group field
dlgHBoxLayout	a horizontal box layout context
dlgIntEdit	an integer entry field
dlgLabel	a text label
dlgListBox	a list box
dlgListView	a list view
dlgPushButton	a push button
dlgRadioButton	a radio button
dlgRealEdit	a real entry field
dlgSpacing	a layout spacing object
dlgSpinBox	a spin box selection field
dlgStretch	a layout stretch object
dlgStringEdit	a string entry field

dlgTabPage	a tab page
dlgTabWidget	a tab page container
dlgTextEdit	a text entry field
dlgTextView	a text viewer field
dlgVBoxLayout	a vertical box layout context

dlgCell

Function

Defines a cell location within a grid layout context.

Syntax

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

See also dlgGridLayout, dlgHBoxLayout, dlgVBoxLayout, Layout Information, A Complete Example

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

The row and column indexes start at 0, so the upper left cell has the index (0, 0).

With two parameters the dialog object defined by `statement` will be placed in the single cell addresses by `row` and `column`. With four parameters the dialog object will span over all cells from `row/column` to `row2/column2`.

By default a `dlgCell` contains a dlgHBoxLayout, so if the cell contains more than one dialog object, they will be placed next to each other horizontally.

Example

```
string Text;
dlgGridLayout {
  dlgCell(0, 0) dlgLabel("Cell 0,0");
  dlgCell(1, 2, 4, 7) dlgTextEdit(Text);
  }
```

dlgCheckBox

Function

Defines a checkbox.

Syntax

```
dlgCheckBox(string Text, int &Checked) [ statement ]
```

See also dlgRadioButton, dlgGroup, Layout Information, A Complete Example

The `dlgCheckBox` statement defines a check box with the given `Text`.

If `Text` contains an `'&'`, the character following the ampersand will become a hotkey, and when the user hits `Alt+hotkey`, the checkbox will be toggled. To have an actual `'&'` character in the text it has to be escaped.

`dlgCheckBox` is mainly used within a dlgGroup, but can also be used otherwise.

All check boxes within the same dialog must have **different** `Checked` variables!

If the user checks a `dlgCheckBox`, the associated `Checked` variable is set to `1`, otherwise it is set to `0`. The initial value of `Checked` defines whether a checkbox is initially checked. If `Checked` is not equal to `0`, the checkbox is initially checked.

The optional `statement` is executed every time the `dlgCheckBox` is toggled.

Example

```
int mirror = 0;
int rotate = 1;
int flip   = 0;
dlgGroup("Orientation") {
  dlgCheckBox("&Mirror", mirror);
  dlgCheckBox("&Rotate", rotate);
  dlgCheckBox("&Flip", flip);
  }
```

dlgComboBox

Function

Defines a combo box selection field.

Syntax

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

See also dlgListBox, dlgLabel, Layout Information, A Complete Example

The `dlgComboBox` statement defines a combo box selection field with the contents of the given `array`.

`Selected` reflects the index of the selected combo box entry. The first entry has index `0`.

Each element of `array` defines the contents of one entry in the combo box. None of the strings in `array` may be empty (if there is an empty string, all strings after and including that one will be dropped).

The optional `statement` is executed whenever the selection in the `dlgComboBox` changes.
Before the `statement` is executed, all variables that have been used with dialog objects are updated to their current values, and any changes made to these variables inside the `statement` will be reflected in the dialog when the statement returns.

If the initial value of `Selected` is outside the range of the `array` indexes, it is set to `0`.

Example

```
string Colors[] = { "red", "green", "blue", "yellow" };
int Selected = 2; // initially selects "blue"
dlgComboBox(Colors, Selected) dlgMessageBox("You have selected " +
Colors[Selected]);
```

dlgDialog

Function

Executes a User Language Dialog.

Syntax

```
int dlgDialog(string Title) block ;
```

Returns

The `dlgDialog` function returns an integer value that can be given a user defined meaning through a call to the `dlgAccept()` function.
If the dialog is simply closed, the return value will be `-1`.

See also dlgGridLayout, dlgHBoxLayout, dlgVBoxLayout, dlgAccept, dlgReset, dlgReject, A Complete Example

The `dlgDialog` function executes the dialog defined by `block`. This is the only dialog object that actually is a User Language builtin function. Therefore it can be used anywhere where a function call is allowed.

The `block` normally contains only other dialog objects, but it is also possible to use other User Language statements, for example to conditionally add objects to the dialog (see the second example below).

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

A `dlgDialog` should at some point contain a call to the `dlgAccept()` function in order to allow the user to close the dialog and accept its contents.

If all you need is a simple message box or file dialog you might want to use one of the Predefined Dialogs instead.

Examples

```
int Result = dlgDialog("Hello") {
  dlgLabel("Hello world");
  dlgPushButton("+OK") dlgAccept();
  };
int haveButton = 1;
dlgDialog("Test") {
  dlgLabel("Start");
  if (haveButton)
     dlgPushButton("Here") dlgAccept();
  };
```

dlgGridLayout

Function

Opens a grid layout context.

Syntax

```
dlgGridLayout statement
```

See also dlgCell, dlgHBoxLayout, dlgVBoxLayout, Layout Information, A Complete Example

The `dlgGridLayout` statement opens a grid layout context.

The only dialog object that can be used directly in `statement` is dlgCell, which defines the location of a particular dialog object within the grid layout.

The row and column indexes start at 0, so the upper left cell has the index (0, 0).
The number of rows and columns is automatically extended according to the location of dialog objects that are defined within the grid layout context, so you don't have to explicitly define the number of rows and columns.

Example

```
dlgGridLayout {
  dlgCell(0, 0) dlgLabel("Row 0/Col 0");
  dlgCell(1, 0) dlgLabel("Row 1/Col 0");
  dlgCell(0, 1) dlgLabel("Row 0/Col 1");
  dlgCell(1, 1) dlgLabel("Row 1/Col 1");
  }
```

dlgGroup

Function

Defines a group field.

Syntax

`dlgGroup(string Title)` *`statement`*

See also dlgCheckBox, dlgRadioButton, Layout Information, A Complete Example

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

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

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

Example

```
int align = 1;
dlgGroup("Alignment") {
  dlgRadioButton("&Top", align);
  dlgRadioButton("&Center", align);
  dlgRadioButton("&Bottom", align);
  }
```

dlgHBoxLayout

Function

Opens a horizontal box layout context.

Syntax

```
dlgHBoxLayout statement
```

See also dlgGridLayout, dlgVBoxLayout, Layout Information, A Complete Example

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

Example

```
dlgHBoxLayout {
  dlgLabel("Box 1");
  dlgLabel("Box 2");
  dlgLabel("Box 3");
  }
```

dlgIntEdit

Function

Defines an integer entry field.

Syntax

```
dlgIntEdit(int &Value, int Min, int Max)
```

See also dlgRealEdit, dlgStringEdit, dlgLabel, Layout Information, A Complete Example

The `dlgIntEdit` statement defines an integer entry field with the given `Value`.

If `Value` is initially outside the range defined by `Min` and `Max` it will be limited to these values.

Example

```
int Value = 42;
dlgHBoxLayout {
  dlgLabel("Enter a &Number between 0 and 99");
  dlgIntEdit(Value, 0, 99);
  }
```

dlgLabel

Function

Defines a text label.

Syntax

```
dlgLabel(string Text [, int Update])
```

See also Layout Information, A Complete Example, dlgRedisplay()

The `dlgLabel` statement defines a label with the given `Text`.

`Text` can be either a string literal, as in `"Hello"`, or a string variable.

If Text contains any HTML tags, the characters '<', '>' and '&' must be given as "<", ">" and "&", respectively, if they shall be displayed as such.

External hyperlinks in the Text will be opened with the appropriate application program.

If the Update parameter is not 0 and Text is a string variable, its contents can be modified in the statement of, e.g., a dlgPushButton, and the label will be automatically updated. This, of course, is only useful if Text is a dedicated string variable (not, e.g., the loop variable of a for statement).

If Text contains an '&', and the object following the label can have the keyboard focus, the character following the ampersand will become a hotkey, and when the user hits Alt+hotkey, the focus will go to the object that was defined immediately following the dlgLabel. To have an actual '&' character in the text it has to be escaped.

Example

```
string OS = "Windows";
dlgHBoxLayout {
  dlgLabel(OS, 1);
  dlgPushButton("&Change OS") { OS = "Linux"; }
  }
```

dlgListBox

Function

Defines a list box selection field.

Syntax

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

See also dlgComboBox, dlgListView, dlgSelectionChanged, dlgLabel, Layout Information, A Complete Example

The dlgListBox statement defines a list box selection field with the contents of the given array.

Selected reflects the index of the selected list box entry. The first entry has index 0.

Each element of array defines the contents of one line in the list box. None of the strings in array may be empty (if there is an empty string, all strings after and including that one will be dropped).

The optional statement is executed whenever the user double clicks on an entry of the dlgListBox (see dlgSelectionChanged for information on how to have the statement called when only the selection in the list changes).
Before the statement is executed, all variables that have been used with dialog objects are updated to their current values, and any changes made to these variables inside the statement will be reflected in the dialog when the statement returns.

If the initial value of Selected is outside the range of the array indexes, no entry will be selected.

Example

```
string Colors[] = { "red", "green", "blue", "yellow" };
int Selected = 2; // initially selects "blue"
dlgListBox(Colors, Selected) dlgMessageBox("You have selected " +
Colors[Selected]);
```

dlgListView

Function

Defines a multi column list view selection field.

Syntax

`dlgListView(string Headers, string array[], int &Selected[, int &Sort]) [ statement ]`

See also dlgListBox, dlgSelectionChanged, dlgLabel, Layout Information, A Complete Example

The `dlgListView` statement defines a multi column list view selection field with the contents of the given `array`.

`Headers` is the tab separated list of column headers.

`Selected` reflects the index of the selected list view entry in the `array` (the sequence in which the entries are actually displayed may be different, because the contents of a `dlgListView` can be sorted by the various columns). The first entry has index `0`.
If no particular entry shall be initially selected, `Selected` should be initialized to `-1`. If it is set to `-2`, the first item according to the current sort column is made current.

`Sort` defines which column should be used to sort the list view. The leftmost column is numbered `1`. The sign of this parameter defines the direction in which to sort (positive values sort in ascending order). If `Sort` is `0` or outside the valid number of columns, no sorting will be done. The returned value of `Sort` reflects the column and sort mode selected by the user by clicking on the list column headers. By default `dlgListView` sorts by the first column, in ascending order.

Each element of `array` defines the contents of one line in the list view, and must contain tab separated values. If there are fewer values in an element of `array` than there are entries in the `Headers` string the remaining fields will be empty. If there are more values in an element of `array` than there are entries in the `Headers` string the superfluous elements will be silently dropped. None of the strings in `array` may be empty (if there is an empty string, all strings after and including that one will be dropped).

A list entry that contains line feeds (`'\n'`) will be displayed in several lines accordingly.

The optional `statement` is executed whenever the user double clicks on an entry of the `dlgListView` (see dlgSelectionChanged for information on how to have the `statement` called when only the selection in the list changes).
Before the `statement` is executed, all variables that have been used with dialog objects are updated to their current values, and any changes made to these variables inside the

`statement` will be reflected in the dialog when the statement returns.

If the initial value of `Selected` is outside the range of the `array` indexes, no entry will be selected.

If `Headers` is an empty string, the first element of the `array` is used as the header string. Consequently the index of the first entry is then `1`.

The contents of a `dlgListView` can be sorted by any column by clicking on that column's header. Columns can also be swapped by "click&dragging" a column header. Note that none of these changes will have any effect on the contents of the `array`. If the contents shall be sorted alphanumerically a `numeric string[]` array can be used.

Example

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

dlgPushButton

Function

Defines a push button.

Syntax

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

See also Layout Information, Dialog Functions, A Complete Example

The `dlgPushButton` statement defines a push button with the given `Text`.

If `Text` contains an `'&'`, the character following the ampersand will become a hotkey, and when the user hits `Alt+hotkey`, the button will be selected. To have an actual `'&'` character in the text it has to be escaped.

If `Text` starts with a `'+'` character, this button will become the default button, which will be selected if the user hits ENTER.
If `Text` starts with a `'-'` character, this button will become the cancel button, which will be selected if the user closes the dialog.
CAUTION: Make sure that the `statement` of such a marked cancel button contains a call to dlgReject()! Otherwise the user may be unable to close the dialog at all!
To have an actual `'+'` or `'-'` character as the first character of the text it has to be escaped.

If the user selects a `dlgPushButton`, the given `statement` is executed.
Before the `statement` is executed, all variables that have been used with dialog objects are updated to their current values, and any changes made to these variables inside the `statement` will be reflected in the dialog when the statement returns.

Example

```
int defaultWidth = 10;
int defaultHeight = 20;
int width = 5;
int height = 7;
dlgPushButton("&Reset defaults") {
  width = defaultWidth;
  height = defaultHeight;
  }
dlgPushButton("+&Accept") dlgAccept();
dlgPushButton("-Cancel") { if (dlgMessageBox("Are you sure?", "Yes", "No") == 0)
dlgReject(); }
```

dlgRadioButton

Function

Defines a radio button.

Syntax

`dlgRadioButton(string Text, int &Selected) [ statement ]`

See also dlgCheckBox, dlgGroup, Layout Information, A Complete Example

The `dlgRadioButton` statement defines a radio button with the given `Text`.

If `Text` contains an `'&'`, the character following the ampersand will become a hotkey, and when the user hits `Alt+hotkey`, the button will be selected. To have an actual `'&'` character in the text it has to be escaped.

`dlgRadioButton` can only be used within a dlgGroup.
All radio buttons within the same group must use the **same** `Selected` variable!

If the user selects a `dlgRadioButton`, the index of that button within the `dlgGroup` is stored in the `Selected` variable.
The initial value of `Selected` defines which radio button is initially selected. If `Selected` is outside the valid range for this group, no radio button will be selected. In order to get the correct radio button selection, `Selected` must be set **before** the first `dlgRadioButton` is defined, and must not be modified between adding subsequent radio buttons. Otherwise it is undefined which (if any) radio button will be selected.

The optional `statement` is executed every time the `dlgRadioButton` is selected.

Example

```
int align = 1;
dlgGroup("Alignment") {
  dlgRadioButton("&Top", align);
  dlgRadioButton("&Center", align);
  dlgRadioButton("&Bottom", align);
  }
```

dlgRealEdit

Function

Defines a real entry field.

Syntax

```
dlgRealEdit(real &Value, real Min, real Max)
```

See also dlgIntEdit, dlgStringEdit, dlgLabel, Layout Information, A Complete Example

The `dlgRealEdit` statement defines a real entry field with the given `Value`.

If `Value` is initially outside the range defined by `Min` and `Max` it will be limited to these values.

Example

```
real Value = 1.4142;
dlgHBoxLayout {
  dlgLabel("Enter a &Number between 0 and 99");
  dlgRealEdit(Value, 0.0, 99.0);
  }
```

dlgSpacing

Function

Defines additional space in a box layout context.

Syntax

```
dlgSpacing(int Size)
```

See also dlgHBoxLayout, dlgVBoxLayout, dlgStretch, Layout Information, A Complete Example

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

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

Example

```
dlgVBoxLayout {
  dlgLabel("Label 1");
  dlgSpacing(40);
  dlgLabel("Label 2");
  }
```

dlgSpinBox

Function

Defines a spin box selection field.

Syntax

```
dlgSpinBox(int &Value, int Min, int Max)
```

See also dlgIntEdit, dlgLabel, Layout Information, A Complete Example

The `dlgSpinBox` statement defines a spin box entry field with the given `Value`.

If `Value` is initially outside the range defined by `Min` and `Max` it will be limited to these values.

Example

```
int Value = 42;
dlgHBoxLayout {
  dlgLabel("&Select value");
  dlgSpinBox(Value, 0, 99);
  }
```

dlgStretch

Function

Defines an empty stretchable space in a box layout context.

Syntax

```
dlgStretch(int Factor)
```

See also dlgHBoxLayout, dlgVBoxLayout, dlgSpacing, Layout Information, A Complete Example

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

`Factor` defines the stretch factor of the space.

Example

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

dlgStringEdit

Function

Defines a string entry field.

Syntax

```
dlgStringEdit(string &Text[, string &History[][, int Size]])
```

See also dlgRealEdit, dlgIntEdit, dlgTextEdit, dlgLabel, Layout Information, A Complete Example

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

If `History` is given, the strings the user has entered over time are stored in that string array. The entry field then has a button that allows the user to select from previously

entered strings. If a `Size` greater than zero is given, only at most that number of strings are stored in the array. If `History` contains data when the dialog is newly opened, that data will be used to initialize the history. The most recently entered user input is stored at index 0.
None of the strings in `History` may be empty (if there is an empty string, all strings after and including that one will be dropped).

Example

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

dlgTabPage

Function

Defines a tab page.

Syntax

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

See also dlgTabWidget, Layout Information, A Complete Example

The `dlgTabPage` statement defines a tab page with the given `Title` containing the given `statement`.

If `Title` contains an `'&'`, the character following the ampersand will become a hotkey, and when the user hits `Alt+hotkey`, this tab page will be opened. To have an actual `'&'` character in the text it has to be escaped.

Tab pages can only be used within a dlgTabWidget.

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

Example

```
dlgTabWidget {
  dlgTabPage("Tab &1") {
    dlgLabel("This is page 1");
    }
  dlgTabPage("Tab &2") {
    dlgLabel("This is page 2");
    }
  }
```

dlgTabWidget

Function

Defines a container for tab pages.

Syntax

```
dlgTabWidget statement
```

See also dlgTabPage, Layout Information, A Complete Example

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

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

Example

```
dlgTabWidget {
  dlgTabPage("Tab &1") {
    dlgLabel("This is page 1");
    }
  dlgTabPage("Tab &2") {
    dlgLabel("This is page 2");
    }
  }
```

dlgTextEdit

Function

Defines a multiline text entry field.

Syntax

```
dlgTextEdit(string &Text)
```

See also dlgStringEdit, dlgTextView, dlgLabel, Layout Information, A Complete Example

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

The lines in the `Text` have to be delimited by a newline character (`'\n'`). Any whitespace characters at the end of the lines contained in `Text` will be removed, and upon return there will be no whitespace characters at the end of the lines. Empty lines at the end of the text will be removed entirely.

Example

```
string Text = "This is some text.\nLine 2\nLine 3";
dlgVBoxLayout {
  dlgLabel("&Edit the text");
  dlgTextEdit(Text);
  }
```

dlgTextView

Function

Defines a multiline text viewer field.

Syntax

```
dlgTextView(string Text)
```

```
dlgTextView(string Text, string &Link) statement
```

See also dlgTextEdit, dlgLabel, Layout Information, A Complete Example

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

The `Text` may contain HTML tags.

External hyperlinks in the `Text` will be opened with the appropriate application program.

If `Link` is given and the `Text` contains hyperlinks, `statement` will be executed every time the user clicks on a hyperlink, with the value of `Link` set to whatever the `<a href=...>` tag defines as the value of *href*. If, after the execution of `statement`, the `Link` variable is not empty, the default handling of hyperlinks will take place. This is also the case if `Link` contains some text before dlgTextView is opened, which allows for an initial scrolling to a given position. If a `Link` is given, external hyperlinks will not be opened.

Example

```
string Text = "This is some text.\nLine 2\nLine 3";
dlgVBoxLayout {
  dlgLabel("&View the text");
  dlgTextView(Text);
  }
```

dlgVBoxLayout

Function

Opens a vertical box layout context.

Syntax

```
dlgVBoxLayout statement
```

See also dlgGridLayout, dlgHBoxLayout, Layout Information, A Complete Example

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

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

Example

```
dlgVBoxLayout {
  dlgLabel("Box 1");
  dlgLabel("Box 2");
  dlgLabel("Box 3");
  }
```

Layout Information

All objects within a User Language Dialog a placed inside a *layout context*.

Layout contexts can be either grid, horizontal or vertical.

Grid Layout Context

Objects in a grid layout context must specify the grid coordinates of the cell or cells into which they shall be placed. To place a text label at row 5, column 2, you would write

```
dlgGridLayout {
  dlgCell(5, 2) dlgLabel("Text");
  }
```

If the object shall span over more than one cell you need to specify the coordinates of the starting cell and the ending cell. To place a group that extends from row 1, column 2 up to row 3, column 5, you would write

```
dlgGridLayout {
  dlgCell(1, 2, 3, 5) dlgGroup("Title") {
    //...
    }
  }
```

Horizontal Layout Context

Objects in a horizontal layout context are placed left to right.

The special objects dlgStretch and dlgSpacing can be used to further refine the distribution of the available space.

To define two buttons that are pushed all the way to the right edge of the dialog, you would write

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

Vertical Layout Context

Objects in a vertical layout context follow the same rules as those in a horizontal layout context, except that they are placed top to bottom.

Mixing Layout Contexts

Vertical, horizontal and grid layout contexts can be mixed to create the desired layout structure of a dialog. See the Complete Example for a demonstration of this.

Dialog Functions

The following functions can be used with User Language Dialogs:

dlgAccept()	closes the dialog and accepts its contents
dlgRedisplay()	immediately redisplays the dialog after changes to any values

dlgReset()	resets all dialog objects to their initial values
dlgReject()	closes the dialog and rejects its contents
dlgSelectionChanged()	tells whether the current selection in a dlgListView or dlgListBox has changed

dlgAccept()

Function

Closes the dialog and accepts its contents.

Syntax

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

See also dlgReject, dlgDialog, A Complete Example

The `dlgAccept` function causes the dlgDialog to be closed and return after the current statement sequence has been completed.

Any changes the user has made to the dialog values will be accepted and are copied into the variables that have been given when the dialog objects were defined.

The optional `Result` is the value that will be returned by the dialog. Typically this should be a positive integer value. If no value is given, it defaults to `1`.

Note that `dlgAccept()` does return to the normal program execution, so in a sequence like

```
dlgPushButton("OK") {
  dlgAccept();
  dlgMessageBox("Accepting!");
  }
```

the statement after `dlgAccept()` will still be executed!

Example

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

dlgRedisplay()

Function

Redisplays the dialog after changing values.

Syntax

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

See also dlgReset, dlgDialog, A Complete Example

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

You only need to call `dlgRedisplay()` if you want the dialog to be refreshed while still executing program code. In the example below the status is changed to "Running..." and `dlgRedisplay()` has to be called to make this change take effect before the "program action" is performed. After the final status change to "Finished." there is no need to call `dlgRedisplay()`, since all dialog objects are automatically updated after leaving the statement.

Example

```
string Status = "Idle";
int Result = dlgDialog("Test") {
               dlgLabel(Status, 1); // note the '1' to tell the label to be
updated!
               dlgPushButton("+OK")    dlgAccept(42);
               dlgPushButton("Cancel") dlgReject();
               dlgPushButton("Run") {
                 Status = "Running...";
                 dlgRedisplay();
                 // some program action here...
                 Status = "Finished.";
                 }
               };
```

dlgReset()

Function

Resets all dialog objects to their initial values.

Syntax

```
void dlgReset(void);
```

See also dlgReject, dlgDialog, A Complete Example

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

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

Calling `dlgReject()` implies a call to `dlgReset()`.

Example

```
int Number = 1;
int Result = dlgDialog("Test") {
               dlgIntEdit(Number);
               dlgPushButton("+OK")    dlgAccept(42);
               dlgPushButton("Cancel") dlgReject();
               dlgPushButton("Reset")  dlgReset();
               };
```

dlgReject()

Function

Closes the dialog and rejects its contents.

Syntax

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

See also dlgAccept, dlgReset, dlgDialog, A Complete Example

The `dlgReject` function causes the dlgDialog to be closed and return after the current statement sequence has been completed.

Any changes the user has made to the dialog values will be discarded. The variables that have been given when the dialog objects were defined will be reset to their original values when the dialog returns.

The optional `Result` is the value that will be returned by the dialog. Typically this should be `0` or a negative integer value. If no value is given, it defaults to `0`.

Note that `dlgReject()` does return to the normal program execution, so in a sequence like

```
dlgPushButton("Cancel") {
  dlgReject();
  dlgMessageBox("Rejecting!");
  }
```

the statement after `dlgReject()` will still be executed!

Calling `dlgReject()` implies a call to `dlgReset()`.

Example

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

dlgSelectionChanged()

Function

Tells whether the current selection in a dlgListView or dlgListBox has changed.

Syntax

```
int dlgSelectionChanged(void);
```

Returns

The `dlgSelectionChanged` function returns a nonzero value if only the selection in the list has changed.

See also dlgListView, dlgListBox

The `dlgSelectionChanged` function can be used in a list context to determine whether the statement of the `dlgListView` or `dlgListBox` was called because the user double clicked on an item, or whether only the current selection in the list has changed.

If the statement of a `dlgListView` or `dlgListBox` doesn't contain any call to

`dlgSelectionChanged`, that statement is only executed when the user double clicks on an item in the list. However, if a ULP needs to react on changes to the current selection in the list, it can call `dlgSelectionChanged` within the list's statement. This causes the statement to also be called if the current selection in the list changes.

If a list item is initially selected when the dialog is opened and the list's statement contains a call to `dlgSelectionChanged`, the statement is executed with `dlgSelectionChanged` returning true in order to indicate the initial change from "no selection" to an actual selection. Any later programmatical changes to the strings or the selection of the list will not trigger an automatic execution of the list's statement. This is important to remember in case the current list item controls another dialog object, for instance a `dlgTextView` that shows an extended representation of the currently selected item.

Example

```
string Colors[] = { "red\tThe color RED", "green\tThe color GREEN", "blue\tThe
color BLUE" };
int Selected = 0; // initially selects "red"
string MyColor;
dlgLabel(MyColor, 1);
dlgListView("Name\tDescription", Colors, Selected) {
  if (dlgSelectionChanged())
     MyColor = Colors[Selected];
  else
     dlgMessageBox("You have chosen " + Colors[Selected]);
  }
```

Escape Character

Some characters have special meanings in button or label texts, so they need to be *escaped* if they shall appear literally.

To do this you need to prepend the character with a *backslash*, as in

```
dlgLabel("Miller \\& Co.");
```

This will result in "Miller & Co." displayed in the dialog.

Note that there are actually **two** backslash characters here, since this line will first go through the User Language parser, which will strip the first backslash.

A Complete Example

Here's a complete example of a User Language Dialog.

```
int hor = 1;
int ver = 1;
string fileName;
int Result = dlgDialog("Enter Parameters") {
  dlgHBoxLayout {
    dlgStretch(1);
    dlgLabel("This is a simple dialog");
```

```
    dlgStretch(1);
    }
  dlgHBoxLayout {
    dlgGroup("Horizontal") {
      dlgRadioButton("&Top", hor);
      dlgRadioButton("&Center", hor);
      dlgRadioButton("&Bottom", hor);
      }
    dlgGroup("Vertical") {
      dlgRadioButton("&Left", ver);
      dlgRadioButton("C&enter", ver);
      dlgRadioButton("&Right", ver);
      }
    }
  dlgHBoxLayout {
    dlgLabel("File &name:");
    dlgStringEdit(fileName);
    dlgPushButton("Bro&wse") {
      fileName = dlgFileOpen("Select a file", fileName);
      }
    }
  dlgGridLayout {
    dlgCell(0, 0) dlgLabel("Row 0/Col 0");
    dlgCell(1, 0) dlgLabel("Row 1/Col 0");
    dlgCell(0, 1) dlgLabel("Row 0/Col 1");
    dlgCell(1, 1) dlgLabel("Row 1/Col 1");
    }
  dlgSpacing(10);
  dlgHBoxLayout {
    dlgStretch(1);
    dlgPushButton("+OK")    dlgAccept();
    dlgPushButton("Cancel") dlgReject();
    }
  };
```