EAGLE

EINFACH ANZUWENDENDER GRAPHISCHER LAYOUT-EDITOR

User-Language

Version 5.4

Inhaltsverzeichnis

User Language....12
Schreiben eines ULP....12
ULP ausführen....13
Syntax....13
Whitespace....13
Kommentare....14
Direktiven....14
#include....15
    Hinweis zur Kompatibilität zwischen den Betriebssystemen....15
#require....15
#usage....16
    Beispiel....16
Schlüsselwörter (Keywords)....16
Identifier....17
Konstanten....17
Character-Konstanten....17
Integer-Konstanten....18
    Beispiele....18
Real-Konstanten....18
    Beispiele....19
String-Konstanten....19
Escape-Sequenzen....19
    Beispiele....20
Punctuator-Zeichen....20
Eckige Klammern....21
Runde Klammern....21
Geschweifte Klammern....21
Komma....22
Semikolon....22
Doppelpunkt....22
Gleichheitszeichen....22
Datentypen....22
char....23
int....23
real....23
string....23
    Implementierungs-Details....24
Typ-Umwandlung....24

Typecast....24
Objekt-Typen....25
UL_ARC....28
Konstanten....28
Anmerkung....28
Beispiel....28
UL_AREA....29
Beispiel....29
UL_ATTRIBUTE....29
Konstanten....29
Anmerkung....30
Beispiel....30
UL_BOARD....30
Anmerkung....31
Beispiel....31
UL_BUS....31
Konstanten....32
Beispiel....32
UL_CIRCLE....32
Beispiel....32
UL_CLASS....32
Anmerkung....33
Beispiel....33
UL_CONTACT....33
Konstanten....33
Anmerkung....33
Beispiel....34
UL_CONTACTREF....34
Beispiel....34
UL_DEVICE....34
Konstanten....35
Anmerkung....35
Beispiele....35
UL_DEVICESET....36
Konstanten....36
Anmerkung....37
Beispiel....37
UL_ELEMENT....37
Konstanten....38
Anmerkung....38
Beispiele....38
UL_FRAME....39
Konstanten....39

Anmerkung....39
Beispiel....40
UL_GATE....40
Konstanten....40
Anmerkung....40
Beispiel....40
UL_GRID....41
Konstanten....41
Anmerkung....41
Beispiel....41
UL_HOLE....42
Anmerkung....42
Beispiel....42
UL_INSTANCE....42
Konstanten....43
Anmerkung....43
Beispiel....44
UL_JUNCTION....44
Beispiel....44
UL_LABEL....44
Anmerkung....45
Beispiel....45
UL_LAYER....45
Konstanten....46
Beispiel....48
UL_LIBRARY....48
Konstanten....48
Anmerkung....49
Beispiel....49
UL_NET....49
Konstanten....50
Anmerkung....50
Beispiel....50
UL_PACKAGE....50
Konstanten....51
Anmerkung....51
Beispiel....51
UL_PAD....52
Konstanten....52
Anmerkung....53
Beispiel....54
UL_PART....54
Konstanten....54
Anmerkung....54

Beispiel....55
UL_PIN....55
Konstanten....56
Anmerkung....57
Beispiel....57
UL_PINREF....58
Beispiel....58
UL_POLYGON....58
Konstanten....59
Anmerkung....59
Polygon-Strichstärke....59
Teilpolygone....59
Beispiel....60
UL_RECTANGLE....61
Beispiel....61
UL_SCHEMATIC....61
Anmerkung....62
Beispiel....62
UL_SEGMENT....62
Anmerkung....62
Beispiel....62
UL_SHEET....63
Beispiel....63
UL_SIGNAL....64
Konstanten....64
Beispiel....64
UL_SMD....64
Konstanten....65
Anmerkung....65
Beispiel....66
UL_SYMBOL....66
Konstanten....66
Anmerkung....66
Beispiel....67
UL_TEXT....67
Konstanten....67
Anmerkung....67
Beispiel....68
UL_VIA....68
Konstanten....68
Anmerkung....69
Beispiel....69
UL_WIRE....69
Konstanten....70

Wire Style....70
Kreisbögen auf Wire-Ebene....70
Beispiel....71
Definitionen....71
Konstanten-Definitionen....71
Variablen-Definitionen....72
Beispiele....72
Funktions-Definitionen....73
Die spezielle Funktion main()....73
Beispiel....73
Operatoren....74
Bitweise Operatoren....74
Logische Operatoren....75
Vergleichs-Operatoren....75
Evaluation-Operatoren....76
Arithmetische Operatoren....77
String-Operatoren....77
Ausdrücke....78
Arithmetischer Ausdruck....78
Beispiele....78
Zuweisungs-Ausdruck....78
Beispiele....79
String-Ausdruck....79
Beispiele....79
Komma-Ausdruck....79
Beispiel....79
Bedingter Ausdruck....79
Beispiel....79
Funktionsaufruf....79
Beispiel....79
Statements....80
Compound-Statement (Verbundanweisung)....80
Expression-Statement (Ausdrucksanweisung)....80
Control-Statements (Steueranweisungen)....80
break....81
continue....81
do...while....81
Beispiel....81
for....82
Beispiel....82
if...else....82

return....83
switch....83
Beispiel....83
while....84
Beispiel....84
Builtins....84
Builtin-Constants....84
Builtin Variablen....86
Builtin-Functions....86
Character-Funktionen....88
is...()....88
Character-Kategorien....89
Beispiel....89
to...()....89
Datei-Funktionen....90
fileerror()....90
Beispiel....90
fileglob()....91
Hinweis für Windows-Anwender....91
Beispiel....91
Dateinamens-Funktionen....91
Beispiel....92
Datei-Daten-Funktionen....92
Beispiel....92
Datei-Einlese-Funktionen....92
fileread()....93
Beispiel....93
Mathematische Funktionen....93
Fehlermeldungen....94
Absolutwert-, Maximum- und Minimum-Funktion....94
Beispiel....94
Rundungs-Funktionen....95
Beispiel....95
Trigonometrische Funktionen....95
Konstanten....95
Beispiel....96
Exponential-Funktionen....96
Anmerkung....96
Beispiel....96
Sonstige Funktionen....96
exit()....97
Konstanten....97

language()....97
Beispiel....98
lookup()....98
Beispiel....99
palette()....100
Konstanten....100
sort()....100
Einzelnes Array sortieren....101
Einen Satz von Arrays sortieren....101
status()....102
system()....102
Ein-/Ausgabe-Umleitung....102
Ausführung im Hintergrund....103
Beispiel....103
Einheiten-Konvertierung....103
Beispiel....104
Print-Funktionen....104
printf()....104
Format-String....104
Format-Specifier....105
Konvertiertyp-Zeichen....105
Flag-Zeichen....106
Width-Specifier....106
Präzisions-Specifier....106
Default-Präzisionswerte....107
Wie die Präzisionsangabe (.n) die Konvertierung beeinflusst....107
Der binäre Wert 0....107
Beispiel....108
sprintf()....108
Format-String....108
Der binäre Wert 0....108
Beispiel....108
String-Funktionen....108
strchr()....109
Beispiel....109
strjoin()....109
Beispiel....110
strlen()....110
Beispiel....110
strlwr()....110
Beispiel....110
strrchr()....110
Beispiel....111

strrstr()....111
Beispiel....111
strsplit()....111
Beispiel....112
strstr()....112
Beispiel....112
strsub()....112
Beispiel....113
strtod()....113
Beispiel....113
strtol()....113
Beispiel....114
strupr()....114
Beispiel....114
Zeit-Funktionen....114
time()....115
Beispiel....115
timems()....115
Beispiel....115
Zeit-Konvertierungen....115
Beispiel....116
Objekt-Funktionen....116
ingroup()....116
Beispiel....117
Builtin-Statements....117
board()....118
Prüfen, ob ein Board geladen ist....118
Zugriff auf ein Board von einem Schaltplan aus....118
Beispiel....118
deviceset()....118
Prüfen, ob ein Device-Set geladen ist....119
Beispiel....119
library()....119
Prüfen, ob eine Bibliothek geladen ist....120
Beispiel....120
output()....120
Datei-Modi....120
Verschachtelte Output-Statements....121
Beispiel....121
package()....121
Prüfen, ob ein Package geladen ist....121
Beispiel....122

schematic()....122
Prüfen, ob ein Schaltplan geladen ist....122
Zugriff auf einen Schaltplan vom Board aus....122
Zugriff auf die gegenwärtige Seite eines Schaltplans....122
Beispiel....123
sheet()....123
Prüfen, ob eine Schaltplanseite geladen ist....123
Beispiel....123
symbol()....123
Prüfen, ob ein Symbol geladen ist....124
Beispiel....124
Dialoge....124
Vordefinierte Dialoge....125
dlgDirectory()....125
Beispiel....125
dlgFileOpen(), dlgFileSave()....125
Beispiel....126
dlgMessageBox()....126
Beispiel....127
Dialog-Objekte....127
dlgCell....128
Beispiel....129
dlgCheckBox....129
Beispiel....130
dlgComboBox....130
Beispiel....130
dlgDialog....130
Beispiele....131
dlgGridLayout....131
Beispiel....132
dlgGroup....132
Beispiel....132
dlgHBoxLayout....132
Beispiel....133
dlgIntEdit....133
Beispiel....133
dlgLabel....133
Beispiel....134
dlgListBox....134
Beispiel....135
dlgListView....135
Beispiel....136

dlgPushButton....136
Beispiel....137
dlgRadioButton....137
Beispiel....137
dlgRealEdit....138
Beispiel....138
dlgSpacing....138
Beispiel....138
dlgSpinBox....139
Beispiel....139
dlgStretch....139
Beispiel....139
dlgStringEdit....139
Beispiel....140
dlgTabPage....140
Beispiel....140
dlgTabWidget....140
Beispiel....141
dlgTextEdit....141
Beispiel....141
dlgTextView....141
Beispiel....142
dlgVBoxLayout....142
Beispiel....142
Layout-Information....142
Grid-Layout-Kontext....143
Horizontal-Layout-Kontext....143
Vertical-Layout-Kontext....143
Gemischter Layout-Kontext....143
Dialog-Funktionen....143
dlgAccept()....144
Beispiel....144
dlgRedisplay()....144
Beispiel....145
dlgReset()....145
Beispiel....145
dlgReject()....146
Beispiel....146
Escape-Zeichen....146
Ein vollständiges Beispiel....147
Unterstützte HTML-Tags....147

User Language

Die EAGLE-User-Language gestattet den Zugriff auf die EAGLE-Datenstrukturen und kann beliebige Ausgabedateien erzeugen.

Um diese Eigenschaft zu Nutzen, müssen Sie ein User-Language-Programm (ULP) schreiben und anschließend ausführen.

Die folgenden Abschnitte beschreiben die User Language im Detail:

Syntax	Syntax-Regeln
Daten-Typen	Definiert die grundlegenden Datentypen (Data types)
Objekt-Typen	Definiert die EAGLE-Objekte (Objects)
Definitionen	Zeigt, wie man eine Definition schreibt
Operatoren	Liste der gültigen Operatoren (Operators)
Ausdrücke	Zeigt, wie man einen Ausdruck (Expression) schreibt
Statements	Definiert die gültigen Statements
Builtins	Liste der Builtin-Constants, -Functions etc.
Dialoge	zeigt wie man grafische Dialoge in ein ULP integriert

Schreiben eines ULP

Ein User-Language-Programm ist eine reine Textdatei und wird in einer C-ähnlichen Syntax geschrieben. User-Language-Programme verwenden die Extension `.ulp`. Sie können ein ULP mit jedem beliebigen Texteditor schreiben, vorausgesetzt, er fügt keine Steuerzeichen ein, oder Sie können den EAGLE-Texteditor verwenden.

Ein User-Language-Programm besteht aus zwei wesentlichen Bestandteilen: Definitionen und Statements.

Definitionen werden verwendet, um Konstanten, Variablen und Funktionen zu definieren, die wiederum in Statements verwendet werden.

Ein einfaches ULP könnte so aussehen:

```
#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));
  }
```

Der Wert der #usage-Directive zeigt im Control Panel die Beschreibung des Programms an.

Soll das Ergebnis des ULPs ein Befehl sein, der im Editor-Fenster ausgeführt werden soll, kann man die Funktion exit() verwenden um den Befehl an das Editor-Fenster zu schicken.

ULP ausführen

User-Language-Programme werden mit Hilfe des RUN-Befehls von der Kommandozeile eines Editor-Fensters aus ausgeführt.

Ein ULP kann die Information zurückgeben, ob es erfolgreich abgeschlossen wurde oder nicht. Sie können die exit()-Funktion verwenden, um das Programm zu beenden und den Rückgabewert (return value) zu setzen.

Ein "return value" von 0 bedeutet, das ULP wurde normal beendet (erfolgreich), während jeder andere Wert einen unnormalen Programmabbruch anzeigt.

Der Default-Rückgabewert jedes ULP ist 0.

Wird der RUN-Befehl als Teil einer Script-Datei, ausgeführt, dann wird die Script-Datei abgebrochen, wenn das ULP mit einem "return value" ungleich 0 beendet wurde.

Eine spezielle Variante der Funktion exit() kann verwendet werden, um einen Befehl als Ergebnis des ULPs an ein Editor-Fenster zu schicken.

Syntax

Die Grundbausteine eines User-Language-Programms sind:

- Whitespace
- Kommentare
- Direktiven
- Schlüsselwörter (Keywords)
- Identifier
- Konstanten
- Punctuator-Zeichen

Alle unterliegen bestimmten Regeln, die in den entsprechenden Abschnitten beschrieben werden.

Whitespace

Bevor ein User-Language-Programm ausgeführt werden kann, muss es von einer Datei eingelesen werden. Während dieses Vorgangs wird er Inhalt der Datei zerlegt (*parsed*) in Tokens und in *Whitespace*.

Leerzeichen (blanks), Tabulatoren, Newline-Zeichen und Kommentar werden als *Whitespace* behandelt und nicht berücksichtigt.

Die einzige Stelle, an der ASCII-Zeichen, die *Whitespace* repräsentieren, berücksichtigt werden, ist innerhalb von Literal Strings, wie in

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

wo das Leerzeichen zwischen `'o'` und `'W'` ein Teil des Strings bleibt.

Wenn dem abschließenden Newline-Zeichen einer Zeile ein Backslash (`\`), vorausgeht, werden Backslash und Newline nicht berücksichtigt.

```
"Hello \
World"
```

wird als `"Hello World"` interpretiert.

Kommentare

Wenn man ein ULP schreibt, sollte man möglichst erklärenden Text hinzufügen, der einen Eindruck davon vermittelt, was dieses Programm tut. Sie können auch Ihren Namen und, falls verfügbar, Ihre Email-Adresse hinzufügen, damit die Anwender Ihres Programms die Möglichkeit haben, mit Ihnen Kontakt aufzunehmen, wenn Sie Probleme oder Verbesserungsvorschläge haben.

Es gibt zwei Möglichkeiten, Kommentare einzufügen. Die erste verwendet die Syntax

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

bei der alle Zeichen zwischen (und einschließlich) den Zeichen `/*` und `*/` als Kommentar interpretiert wird. Solche Kommentare können über mehrere Zeilen gehen, wie in

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

aber sie lassen sich nicht verschachteln. Das erste `*/` das einem `/*` folgt, beendet den Kommentar.

Die zweite Möglichkeit, einen Kommentar einzufügen, verwendet die Syntax

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

Dabei werden alle Zeichen nach (und einschließlich) dem `//` bis zum Newline-Zeichen (aber nicht einschließlich) am Ende der Zeile als Kommentar interpretiert.

Direktiven

Folgende *Direktiven* sind verfügbar:

`#include`
`#require`
`#usage`

#include

Ein ULP kann Befehle aus einem anderen ULP durch die `#include`-Direktive ausführen. Die Syntax lautet

```
#include "filename"
```

Die Datei `filename` wird zuerst im selben Verzeichnis in dem sich auch die Source-Datei (das ist die Datei mit der `#include`-Directive) befindet, gesucht. Wird sie dort nicht gefunden, wird in den angegebenen ULP-Verzeichnissen gesucht.

Die maximale "include-Tiefe" ist 10.

Jede `#include`-Direktive wird nur **einmal** ausgeführt. So wird sichergestellt, dass keine Mehrfachdefinitionen von Variablen oder Funktionen entstehen, die Fehler verursachen könnten.

Hinweis zur Kompatibilität zwischen den Betriebssystemen

Enthält *filename* eine Pfadangabe, ist es das Beste als Trennzeichen immer den **Forward-Slash** (/)zu verwenden (auch unter Windows!).
Laufwerksbuchstaben unter Windows sollten vermieden werden. Wird das berücksichtigt, läuft das ULP unter allen Betriebssystemen.

#require

Mit der Zeit kann es vorkommen, dass neuere EAGLE-Versionen neue oder veränderte User Language Funktionen implementieren, die Fehlermeldungen verursachen können, wenn ein solches ULP aus einer älteren EAGLE-Version heraus aufgerufen wird. Um dem Benutzer eine konkrete Meldung zu geben, dass dieses ULP mindestens eine bestimmte EAGLE-Version benötigt, kann ein ULP die `#require`-Direktive enthalten. Die Syntax lautet

```
#require version
```

Die *version* muss als Real-Konstante der Form

```
V.RRrr
```

angegeben werden, wobei `V` die Versionsnummer ist, `RR` die Release-Nummer und `rr` die (optionale) Revisions-Nummer (beide mit führenden Nullen aufgefüllt, falls sie kleiner als 10 sind). Falls also zum Beispiel ein ULP mindestens die EAGLE-Version 4.11r06 voraussetzt (welches die Betaversion war die als erste die `#require`-Direktive implementierte), könnte es

```
#require 4.1106
```

benutzen. Entsprechend würde für Version 5.1.2

```
#require 5.0102
```

gelten.

#usage

Jedes User-Language-Programm sollte Informationen über seine Funktion, die Benutzung und evtl. auch über den Autor enthalten.
Die Direktive

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

ist die übliche Methode diese Information verfügbar zu machen.

Wird die `#usage`-Direktive verwendet, wird ihr `Text` (der eine String-Konstante sein muss) im Control Panel verwendet, um die Beschreibung des Programms anzuzeigen.

Für den Fall, dass das ULP diese Information z. B. in einer dlgMessageBox() benötigt, ist dieser `Text` durch die Builtin-Konstante `usage` im ULP verfügbar.

Es wird nur die `#usage`-Direktive des Hauptprogramms (das ist die Datei, die mit dem RUN-Befehl gestartet wurde) berücksichtigt. Deshalb sollten reine include-Dateien auch eine eigene `#usage`-Directive enthalten.

Am besten ist die `#usage`-Direktive an den Anfang der Datei zu stellen, so muss das Control Panel nicht den ganzen Text der Datei durchsuchen, um die Informationen, die angezeigt werden sollen, zu finden.

Soll die Usage-Information in mehreren Sprachen verfügbar gemacht werden, so sind die Texte der verschiedenen Sprachen durch Kommas getrennt anzugeben. Dabei muss jeder Text mit dem zweibuchstabigen Code der jeweiligen Sprache (so wie er auch von der language()-Funktion geliefert wird), gefolgt von einem Doppelpunkt und beliebig vielen Leerzeichen beginnen. Falls für die auf dem aktuellen System verwendete Sprache kein passender Text gefunden wird, so wird der erste angegebene Text verwendet (dieser sollte generell Englisch sein um das Programm einer möglichst großen Zahl von Benutzern zugänglich zu machen).

Beispiel

```
#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"
           "Autor: john@home.org"
```

Schlüsselwörter (Keywords)

Die folgenden *Schlüsselwörter* sind für spezielle Zwecke reserviert und dürfen nicht als normale Identifier-Namen verwendet werden:

`break`
`case`

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

Zusätzlich sind die Namen von Builtins und Objekt-Typen reserviert und dürfen nicht als Identifier-Namen verwendet werden.

Identifier

Ein *Identifier* ist ein Name, der dazu benutzt wird, eine benutzerdefinierte Konstante, Variable oder Funktion einzuführen.

Identifier bestehen aus einer Sequenz von Buchstaben (`a b c`..., `A B C`...), Ziffern (`1 2 3`...) und Unterstreichungszeichen (`_`). Das erste Zeichen eines Identifiers **muss** ein Buchstabe oder ein Unterstreichungszeichen sein.

Identifier sind case-sensitive, das bedeutet, dass

```
int Number, number;
```

zwei unterschiedliche Integer-Variablen definieren würde.

Die maximale Länge eines Identifiers ist 100 Zeichen, von denen alle signifikant sind.

Konstanten

Konstanten sind gleichbleibende Daten, die in ein User-Language-Programm geschrieben werden. Analog zu den verschiedenen Datentypen gibt es auch unterschiedliche Typen von Konstanten.

- Character-Konstanten
- Integer-Konstanten
- Real-Konstanten
- String-Konstanten

Character-Konstanten

Eine *Character-Konstante* besteht aus einem einzelnen Buchstaben oder einer Escape-Sequenz, eingeschlossen in einfachen Hochkommas, wie in

```
'a'
'='
```

```
'\n'
```

Der Typ der Character-Konstante ist `char`.

Integer-Konstanten

Abhängig vom ersten (eventuell auch vom zweiten) Zeichen wird eine *Integer-Konstante* unterschiedlich interpretiert:

erstes	zweites	Konstante interpretiert als
`0`	`1-7`	oktal (Basis 8)
`0`	`x,X`	hexadezimal (Basis 16)
`1-9`		dezimal (Basis 10)

Der Typ einer Integer-Konstante ist `int`.

Beispiele

`16`	dezimal
`020`	oktal
`0x10`	hexadezimal

Real-Konstanten

Eine *Real-Konstante* folgt dem allgemeinen Muster

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

wobei die einzelnen Teile für

- Vorzeichen (optional)
- Dezimal-Integer
- Dezimalpunkt
- Dezimalbruch
- `e` oder `E` und ein Integer-Exponent mit Vorzeichen

stehen.

Sie können entweder Dezimal-Integer-Zahl oder Dezimalbruch weglassen (aber nicht beides). Sie können entweder den Dezimalpunkt oder den Buchstaben `e` oder `E` und den Integer-Exponenten mit Vorzeichen weglassen (aber nicht beides).

Der Typ einer Real-Konstante ist `real`.

Beispiele

Konstante	Wert
`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-Konstanten

Eine *String-Konstante* besteht aus einer Sequenz von Buchstaben oder einer Escape-Sequenz, eingeschlossen in doppelte Anführungszeichen, wie in

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

Der Typ einer String-Konstante ist `string`.

String-Konstanten können jede beliebige Länge haben, vorausgesetzt es steht genügend Speicher zur Verfügung.

String-Konstanten können mit dem einfach aneinandergereiht werden um längere Strings zu bilden:

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

Es ist auch möglich, eine String-Konstante über mehrere Zeilen zu schreiben, indem man das Newline-Zeichen mit Hilfe des Backslash (`\`) "ausblendet":

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

Escape-Sequenzen

Eine *Escape-Sequenz* besteht aus einem Backslash (`\`), gefolgt von einem oder mehreren Sonderzeichen:

Sequenz	Bedeutung

`\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` = bis 3 octal digits
`\xH`	`H` = bis 2 hex digits

Jedes Zeichen nach dem Backslash, das nicht in der Liste aufgeführt ist, wird als dieses Zeichen (ohne Backslash) behandelt.

Escape-Sequenzen können in Character-Konstanten und String-Konstanten verwendet werden.

Beispiele

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

Punctuator-Zeichen

Die *Punctuator-Zeichen*, die in einem User-Language-Programm benutzt werden können, sind

`[]`	Eckige Klammern (Brackets)
`()`	Runde Klammern (Parentheses)
`{}`	Geschweifte Klammern (Braces)

, Komma

; Semikolon

: Doppelpunkt (Colon)

= Gleichheitszeichen

Andere Sonderzeichen werden als Operatoren verwendet.

Eckige Klammern

Eckige Klammern (*Brackets*) werden verwendet in Array-Definitionen:

```
int ai[];
```

in Array-Subscripts

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

und in String-Subscripts, um auf die einzelnen Zeichen eines Strings zuzugreifen

```
string s = "Hello world";
char c = s[2];
```

Runde Klammern

Runde Klammern (*Parentheses*) gruppieren Ausdrücke (ändern eventuell die Priorität der Operatoren), isolieren bedingte Ausdrücke und bezeichnen Funktionsaufrufe und Funktionsparameter:

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

Geschweifte Klammern

Geschweifte Klammern (*Braces*) bezeichnen den Beginn und das Ende einer Verbundanweisung (*Compound Statement*)

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

und werden auch verwendet, um die Werte für die Array-Initialisierung zu gruppieren:

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

Komma

Das *Komma* trennt die Elemente einer Funktionsargument-Liste oder die Parameter eines Funktionsaufrufs:

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

Es trennt auch die Wertangaben bei der Array-Initialisierung:

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

und es begrenzt die Elemente einer Variablen-Definition:

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

Semikolon

Der *Semikolon* schließt ein Statement ab, wie in

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

und er begrenzt die Init-, Test- und Inkrement-Ausdrücke eines for Statements:

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

Doppelpunkt

Der *Doppelpunkt* bezeichnet das Ende eines Labels in einem 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");
  }
```

Gleichheitszeichen

Das *Gleichheitszeichen* trennt Variablen-Definitionen von Initialisierungsliste:

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

Es wird auch als Zuweisungsoperator verwendet.

Datentypen

Ein User-Language-Programm kann Variablen unterschiedlicher Typen definieren, die unterschiedliche Arten von EAGLE-Daten repräsentieren.

Die vier grundlegenden Datentypen sind

`char` für Einzelzeichen

`int` für Ganzzahlen

`real` für Gleitkommazahlen

`string` für Textinformation

Neben diesen grundlegenden Datentypen gibt es auch High-level-Objekt-Typen, die die Datenstrukturen repräsentieren, wie sie in den EAGLE-Dateien gespeichert sind.

Der Datentyp `void` wird nur als Return-Typ einer Funktion verwendet. Er zeigt an, dass diese Funktion keinen Wert zurückgibt.

char

Der Datentyp `char` speichert Einzelzeichen, wie die Buchstaben des Alphabets oder kleine Zahlen ohne Vorzeichen.

Eine Variable des Typs `char` belegt 8 bit (1 Byte), und kann jeden Wert im Bereich `0..255` speichern.

Siehe auch Operatoren, Character-Konstanten

int

Der Datentyp `int` speichert Ganzzahlen mit Vorzeichen, wie die Koordinaten eines Objekts.

Eine Variable vom Typ `int` belegt 32 bit (4 Byte), und kann jeden Wert im Bereich `-2147483648..2147483647` speichern.

Siehe auch Integer-Konstanten

real

Der Datentyp `real` speichert Gleitkommazahlen mit Vorzeichen, z.B. den Rasterabstand.

Eine Variable vom Typ `real` belegt 64 bit (8 Byte), und kann jeden Wert im Bereich `±2.2e-308..±1.7e+308` mit einer Genauigkeit von 15 Digits speichern.

Siehe auch Real-Konstanten

string

Der Datentyp `string` speichert Textinformation, z.B. den Namen eines Bauteils oder eines Netzes.

Eine Variable des Typs `string` ist nicht auf eine bestimmte Länge beschränkt, vorausgesetzt, es steht genügend Speicher zur Verfügung.

Variablen des Typs `string` sind ohne explizite Länge definiert. Sie "wachsen" automatisch, soweit erforderlich, während der Programmausführung.

Die Elemente einer `String`-Variablen sind vom Typ `char`, und man kann auf sie individuell zugreifen, indem man `[index]` benutzt.

Das erste Zeichen eines `Strings` hat den Index `0`:

```
string s = "Layout";
printf("Third char is: %c\n", s[2]);
```

Hier würde das Zeichen `'y'` ausgedruckt. Beachten Sie, dass `s[2]` das dritte Zeichen des Strings `s` liefert!

Siehe auch Operatoren, Builtin-Functions, String-Konstanten

Implementierungs-Details

Der Datentyp `string` ist implementiert wie von C her bekannte "Zero-terminated-Strings", (also mit `char[]`). Betrachtet man die folgende Variablen-Definition

```
string s = "abcde";
```

dann ist `s[4]` das Zeichen `'e'`, und `s[5]` ist das Zeichen `'\0'`, oder der Integer-Wert `0x00`. Diese Tatsache kann dazu ausgenutzt werden, das Ende eines Strings ohne die Funktion `strlen()` festzustellen, wie in

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

Es ist auch völlig in Ordnung, einen String "abzuschneiden", indem man den Wert "0" an der gewünschten Stelle einfügt:

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

Als Ergebnis erhält man für den String `s` den Wert `"abc"`. Beachten Sie bitte, dass alles, was auf den Wert "0" folgt, wirklich verschwunden ist, und auch nicht mehr zurückgeholt werden kann, indem der ursprüngliche Wert wieder eingesetzt wird. Das gleiche gilt auch für jede andere Operation, bei der ein Zeichen zu 0 wird, wie etwa --s[3].

Typ-Umwandlung

Der Typ des Ergebnisses eines arithmetischen Ausdrucks, wie z.B. `a + b`, wobei `a` und `b` unterschiedliche arithmetische Typen sind, ist gleich dem "größeren" der beiden Operanden-Typen.

Arithmetische Typen sind `char`, `int` und `real` (in dieser Reihenfolge). Ist zum Beispiel `a` vom Typ `int` und `b` vom Typ `real`, dann ist das Ergebnis `a + b` vom Typ `real`.

Siehe auch Typecast

Typecast

Der Ergebnis-Typ eines arithmetischen Ausdrucks kann explizit in einen anderen

arithmetischen Typ umgewandelt werden, indem man einen *Typecast* darauf anwendet.

Die allgemeine Syntax eines Typecast ist

```
type(expression)
```

wobei `type` `char`, `int` oder `real` ist und `expression` jeder arithmetische Ausdruck sein kann.

Wenn man mit Typecast einen Ausdruck vom Typ `real` in `int` umwandelt, wird der Dezimalbruch des Wertes abgeschnitten.

Siehe auch Typ-Umwandlung

Objekt-Typen

Die EAGLE-Datenstruktur ist in drei Binärdatei-Typen gespeichert:

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

Diese Dateien enthalten Objekte, die hierarchisch gegliedert sind. In einem User-Language-Programm kann man auf die Hierarchiestufen mit Hilfe der entsprechenden Builtin-Zugriffs-Statements zugreifen:

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

Diese Zugriffs-Statements schaffen einen Kontext, innerhalb dessen Sie auf alle Objekte in Bibliotheken, Schaltplänen oder Platinen zugreifen können.

Auf die "Properties" dieser Objekte kann mit Hilfe von *Members* zugegriffen werden.

Es gibt zwei Arten von Members:

- Data members
- Loop members

Data members liefern die Objektdaten unmittelbar. Zum Beispiel in

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

liefert Data member *name* des Board-Objekts *B* den Board-Namen.
Data members können auch andere Objekte zurückgeben, wie in

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

wo Data member *grid* des Boards ein Grid-Objekt zurückliefert, dessen Data member *size* dann Grid-Size (Rastergröße) zurückgibt.

Loop members werden verwendet, um auf Mehrfach-Objekte derselben Art zuzugreifen,

die in einem Objekt einer höheren Hierarchiestufe enthalten sind:

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

Dieses Beispiel verwendet Loop member *elements()* des Boards, um eine Schleife durch alle Board-Elemente zu realisieren. Der Block nach dem `B.elements(E)`-Statement wird der Reihe nach für jedes Element ausgeführt, und das gegenwärtige Element kann innerhalb des Blocks unter dem Namen `E` angesprochen werden.

Loop members behandeln Objekte in alpha-numerisch sortierter Reihenfolge, falls die Objekte einen Namen haben.

Eine Loop-member-Funktion erzeugt eine Variable vom erforderlichen Typ, um die Objekte zu speichern. Sie dürfen jeden gültigen Namen für eine derartige Variable verwenden, so dass das obige Beispiel auch so lauten könnte:

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

Das Ergebnis wäre identisch mit dem vorhergehenden Beispiel. Der Gültigkeitsbereich einer Variablen, die von einer Loop-member-Funktion angelegt wird, ist auf das Statement oder den Block unmittelbar nach dem Loop-Funktionsaufruf beschränkt.

Objekt-Hierarchie einer Bibliothek:

```
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
```

Objekt-Hierarchie eines Schaltplans:

```
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
```

Objekt-Hierarchie einer Platine:

```
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 (Startwinkel, `0.0...359.9`)

`angle2` real (Endwinkel, `0.0...719.9`)

`cap` int (CAP_...)

`layer` int

`radius` int

`width` int

`x1, y1` int (Startpunkt)

`x2, y2` int (Endpunkt)

`xc, yc` int (Mittelpunkt)

Siehe auch UL_WIRE

Konstanten

`CAP_FLAT` flache Kreisbogen-Enden

`CAP_ROUND` runde Kreisbogen-Enden

Anmerkung

Start- und Endwinkel werden im mathematisch positiven Sinne ausgegeben (also gegen den Uhrzeigersinn, "counterclockwise"), wobei gilt `angle1` < `angle2`. Um diese Bedingung einzuhalten kann es sein, dass Start- und Endpunkt eines UL_ARC gegenüber denen des UL_WIRE, von dem der Kreisbogen abstammt, vertauscht sind.

Beispiel

```
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 (linke untere Ecke)
`x2, y2`	int (rechte obere Ecke)

Siehe auch UL_BOARD, UL_DEVICE, UL_PACKAGE, UL_SHEET, UL_SYMBOL

UL_AREA ist ein Pseudo-Objekt, das Informationen über die Fläche liefert, die ein Objekt einnimmt. Für UL_DEVICE, UL_PACKAGE und UL_SYMBOL ist die Fläche definiert als umschließendes Rechteck der Objekt-Definition in der Bibliothek. Deshalb geht der Offset in einem Board nicht in die Fläche ein, obwohl UL_PACKAGE von UL_ELEMENT abgeleitet wird.

Beispiel

```
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=variabel, d.h. überschreiben erlaubt, 1=konstant - siehe Anmerkung)
`defaultvalue`	string (siehe Anmerkung)
`display`	int (`ATTRIBUTE_DISPLAY_FLAG_...`)
`name`	string
`text`	UL_TEXT (siehe Anmerkung)
`value`	string

Siehe auch UL_DEVICE, UL_PART, UL_INSTANCE, UL_ELEMENT

Konstanten

`ATTRIBUTE_DISPLAY_FLAG_OFF`	keine Anzeige
`ATTRIBUTE_DISPLAY_FLAG_VALUE`	Wert wird angezeigt
`ATTRIBUTE_DISPLAY_FLAG_NAME`	Name wird angezeigt

Ein UL_ATTRIBUTE kann dazu benutzt werden, die *Attribute* anzusprechen, die für ein Device in der Bibliothek definiert wurden, bzw. einem Bauteil im Schaltplan oder Board zugewiesen wurden.

Anmerkung

`display` enthält einen bitweise ODER-verknüpften Wert, bestehend aus `ATTRIBUTE_DISPLAY_FLAG_...`, der angibt welche Teile des Attributs dargestellt werden.

In einem UL_ELEMENT-Kontext liefert `constant` nur bei aktiver F/B-Annotation einen tatsächlichen Wert, ansonsten wird 0 geliefert.

Das `defaultvalue`-Member liefert den Wert, wie er in der Bibliothek definiert wurde (falls sich dieser vom tatsächlichen Wert unterscheidet, ansonsten ist es der selbe Wert wie bei `value`). In einem UL_ELEMENT-Kontext liefert `defaultvalue` nur bei aktiver F/B-Annotation einen tatsächlichen Wert, ansonsten wird ein leerer String geliefert.

Das `text`-Member ist nur in einem UL_INSTANCE- oder UL_ELEMENT-Kontext verfügbar und liefert ein UL_TEXT-Objekt welches alle Text-Parameter enthält. Der Wert dieses Text-Objekts ist der Text wie er gemäß dem 'display'-Parameter des UL_ATTRIBUTE angezeigt wird. Wird diese Funktion aus einem anderen Kontext heraus aufgerufen, so sind die Werte des zurückgegebenen UL_TEXT-Objekts undefiniert.

Bei globalen Attributen sind nur `name` und `value` definiert.

Beispiel

```
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

Loop members

`attributes()`	UL_ATTRIBUTE (siehe Anmerkung)
`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

Siehe auch UL_LIBRARY, UL_SCHEMATIC

Anmerkung

Das Loop member `attributes()` geht durch die *globalen* Attribute.

Beispiel

```
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

Siehe auch UL_SHEET

Konstanten

BUS_NAME_LENGTH	max. Länge eines Busnamens (obsolet - ab Version 4 können Bus-Namen beliebig lang sein)

Beispiel

```
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 (Mittelpunkt)

Siehe auch UL_BOARD, UL_PACKAGE, UL_SHEET, UL_SYMBOL

Beispiel

```
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 (siehe Anmerkung)
drill	int
name	string (siehe Anmerkung)
number	int

`width` int

Siehe auch Design Rules, UL_NET, UL_SIGNAL, UL_SCHEMATIC, UL_BOARD

Anmerkung

Das `clearance` Data Member liefert den Mindestabstand zwischen dieser Netzklasse und der Netzklasse mit der angegebenen Nummer. Wird keine Nummer angegeben (und damit auch keine eckigen Klammern), wird der Mindestabstand der Netzklasse selber geliefert. Wird eine Nummer angegeben, so muss diese zwischen 0 und der Nummer dieser Netzklasse liegen.

Wenn `name` einen leeren String liefert, ist die Netzklasse nicht definiert und wird somit auch nicht von einem Signal oder Netz benutzt.

Beispiel

```
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 (Mittelpunkt, siehe Anmerkung)

Siehe auch UL_PACKAGE, UL_PAD, UL_SMD, UL_CONTACTREF, UL_PINREF

Konstanten

`CONTACT_NAME_LENGTH`	max. empfohlene Länge eines "Contact"-Namens (wird nur für formatierte Ausgaben benutzt)

Anmerkung

Das `signal` Data Member liefert den Namen des Signals, an das dieser Contact angeschlossen ist (nur in einem Board-Kontext verfügbar).

Die Koordinaten (`x, y`) des "Contacts" hängen vom Kontext ab aus dem sie aufgerufen

werden:

- Wird "Contact" aus einem UL_LIBRARY-Kontext aufgerufen, sind die Koordinaten dieselben, wie in der Package-Zeichnung
- In allen anderen Fällen gelten die aktuellen Werte in der Board-Datei

Beispiel

```
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

Siehe auch UL_SIGNAL, UL_PINREF

Beispiel

```
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

`prefix`	string (`DEVICE_PREFIX_LENGTH`)
`technologies`	string (siehe Anmerkung)
`value`	string ("On" oder "Off")

Loop members

`attributes()`	UL_ATTRIBUTE (siehe Anmerkung)
`gates()`	UL_GATE

Siehe auch UL_DEVICESET, UL_LIBRARY, UL_PART

Konstanten

`DEVICE_NAME_LENGTH`	max. empfohlene Länge eines Device-Namens (wird nur für formatierte Ausgaben benutzt)
`DEVICE_PREFIX_LENGTH`	max. empfohlene Länge eines Device-Präfix (wird nur für formatierte Ausgaben benutzt)

Alle UL_DEVICE-Member, mit Ausnahme von `name` und `technologies`, liefern dieselben Werte wie die zugehörigen UL_DEVICESET-Member in dem UL_DEVICE definiert wurde. Bitte denken Sie daran: Der `description`-Text darf Newline-Zeichen (`'\n'`) enthalten.

Anmerkung

Der Wert des `technologies`-Member hängt vom Kontext ab aus dem es aufgerufen wurde:

- Wird das Device über UL_DEVICESET hergeleitet, liefert `technologies` einen String, der alles über die Technologien des Devices, durch Leerzeichen getrennt, enthält
- Wird das Device über UL_PART hergeleitet, wird nur die aktuelle Technologie, die von diesem Bauteil benutzt wird, ausgegeben.

Das Loop member `attributes()` erwartet einen zusätzlichen Parameter der angibt, für welche Technology die Attribute geliefert werden sollen (siehe das zweite Beispiel).

Beispiele

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

```
      }
    }
  }
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 (siehe Anmerkung)
library	string
name	string (DEVICE_NAME_LENGTH)
prefix	string (DEVICE_PREFIX_LENGTH)
value	string ("On" oder "Off")

Loop members

devices()	UL_DEVICE
gates()	UL_GATE

Siehe auch UL_DEVICE, UL_LIBRARY, UL_PART

Konstanten

DEVICE_NAME_LENGTH	max. empfohlene Länge des Device-Namen (wird nur bei formatierten Ausgaben benutzt)
DEVICE_PREFIX_LENGTH	max. empfohlene Länge des Prefix (wird nur bei formatierten Ausgaben benutzt)

Anmerkung

Das `description`-Member liefert den vollständigen Beschreibungstext, der mit dem DESCRIPTION-Befehl erzeugt wurde, während das `headline`-Member nur die erste Zeile der Beschreibung ohne HTML-Tags ausgibt. Wenn Sie `description`-Text schreiben, denken Sie daran, dass dieser Newline-Anweisungen (`'\n'`) enthalten darf.

Beispiel

```
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 (siehe Anmerkung)
`column`	string (siehe Anmerkung)
`locked`	int
`mirror`	int
`name`	string (`ELEMENT_NAME_LENGTH`)
`package`	UL_PACKAGE
`row`	string (siehe Anmerkung)
`smashed`	int (siehe Anmerkung)
`spin`	int
`value`	string (`ELEMENT_VALUE_LENGTH`)
`x, y`	int (Ursprung, Aufhängepunkt)

Loop members

`attributes()`	UL_ATTRIBUTE

`texts()`	UL_TEXT (siehe Anmerkung)

Siehe auch UL_BOARD, UL_CONTACTREF

Konstanten

`ELEMENT_NAME_LENGTH`	max. empfohlene Länge eines Element-Namens (wird nur für formatierte Ausgaben benutzt)
`ELEMENT_VALUE_LENGTH`	max. empfohlene Länge eines Element-Values (wird nur für formatierte Ausgaben benutzt)

Anmerkung

Mit dem `attribute[]`-Member kann man ein UL_ELEMENT nach dem Wert eines bestimmten Attributs fragen (siehe das zweite Beispiel). Der zurückgelieferte String ist leer, wenn es kein Attribut mit dem angegebenen Namen gibt, oder wenn dieses Attribut explizit leer ist.

Das `texts()`-Member läuft nur durch die mittels **SMASH** vom Element losgelösten Texte und durch die sichtbaren Texte der Attribute, die diesem Element zugewiesen wurden. Um alle Texte eines Elements zu bearbeiten (zum Beispiel um es zu zeichnen), müssen Sie eine Schleife durch das `texts()`-Member des Elements selbst und durch das `texts()`-Member des zum Element gehörenden Package bilden.

`angle` gibt an um wieviel Grad das Element gegen den Uhrzeigersinn um seinen Aufhängepunkt gedreht ist.

Die `column()`- und `row()`-Members liefern die Spalten- bzw. Zeilenposition innerhalb des Rahmens in der Board-Zeichnung. Falls es in der Zaichnung keinen Rahmen gibt, oder das Element außerhalb des Rahmens liegt, wird ein `'?'` (Fragezeichen) zurückgegeben.

Das `smashed`-Member gibt Auskunft darüber, ob ein Element gesmasht ist. Diese Funktion kann auch verwendet werden um herauszufinden, ob es einen losgelösten Platzhaltertext gibt, indem der Name des Platzhalters in eckigen Klammern angegeben wird, wie in `smashed["VALUE"]`. Dies ist nützlich falls Sie einen solchen Text mit dem MOVE-Befehl etwa durch `MOVE R5>VALUE` selektieren wollen. Gültige Platzhalternamen sind "NAME" und "VALUE", sowie die Namen etwaiger benutzerdefinierter Attribute.
Groß-/Kleinschreibung spielt keine Rolle, und sie dürfen ein vorangestelltes `'>'` Zeichen haben.

Beispiele

```
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

Siehe auch UL_BOARD, UL_PACKAGE, UL_SHEET, UL_SYMBOL

Konstanten

`FRAME_BORDER_BOTTOM`	unterer Rand wird dargestellt
`FRAME_BORDER_RIGHT`	rechter Rand wird dargestellt
`FRAME_BORDER_TOP`	oberer Rand wird dargestellt
`FRAME_BORDER_LEFT`	linker Rand wird dargestellt

Anmerkung

`border` enthält einen bitweise ODER-verknüpften Wert, bestehend aus `FRAME_BORDER_...`, der angibt welche der vier Ränder dargestellt werden.

Die Loop-Members `texts()` und `wires()` gehen durch alle Texte und Linien, aus denen der Frame besteht.

Beispiel

```
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 (Aufhängepunkt, siehe Anmerkung)

Siehe auch UL_DEVICE

Konstanten

GATE_ADDLEVEL_MUST	must
GATE_ADDLEVEL_CAN	can
GATE_ADDLEVEL_NEXT	next
GATE_ADDLEVEL_REQUEST	request
GATE_ADDLEVEL_ALWAYS	always
GATE_NAME_LENGTH	max. empfohlene Länge eines Gate-Namens (wird nur für formatierte Ausgaben benutzt)

Anmerkung

Die Koordinaten des Aufhängepunktes (x, y) sind immer bezogen auf die Lage des Gates im Device, auch wenn das UL_GATE über ein UL_INSTANCE geholt wurde.

Beispiel

```
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_...)

Siehe auch UL_BOARD, UL_LIBRARY, UL_SCHEMATIC, Unit Conversions

Konstanten

GRID_UNIT_MIC	Micron
GRID_UNIT_MM	Millimeter
GRID_UNIT_MIL	Mil
GRID_UNIT_INCH	Inch

Anmerkung

unitdist liefert die Grid-Einheit mit der die tatsächliche Größe des Rasters (die durch distance geliefert wird) definiert wurde, während unit die Grid-Einheit liefert, die für die Anzeige von Werten und die Umrechnung von Benutzereingaben verwendet wird.

Beispiel

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

UL_HOLE

Data members

`diameter[layer]`	int (siehe Anmerkung)
`drill`	int
`drillsymbol`	int
`x, y`	int (Mittelpunkt)

Siehe auch UL_BOARD, UL_PACKAGE

Anmerkung

`diameter[]` ist nur für die Layer `LAYER_TSTOP` und `LAYER_BSTOP` definiert und liefert den Durchmesser der Lötstopmaske im jeweiligen Layer.

`drillsymbol` liefert die Nummer des Bohrsymbols, welches diesem Bohrdurchmesser zugeordnet worden ist (siehe die Liste der definierten Bohrsymbole im Handbuch). Ein Wert von `0` bedeutet, dass diesem Bohrdurchmesser kein Bohrsymbol zugeordnet ist.

Beispiel

```
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` und `270`)
`column`	string (siehe Anmerkung)
`gate`	UL_GATE
`mirror`	int
`name`	string (`INSTANCE_NAME_LENGTH`)
`row`	string (siehe Anmerkung)
`sheet`	int (0=unbenutzt, >0=Seitennummer)

`smashed` int (siehe Anmerkung)

`value` string (`PART_VALUE_LENGTH`)

`x, y` int (Aufhängepunkt)

Loop members

`attributes()` UL_ATTRIBUTE (siehe Anmerkung)

`texts()` UL_TEXT (siehe Anmerkung)

`xrefs()` UL_GATE (siehe Anmerkung)

Siehe auch UL_PART, UL_PINREF

Konstanten

`INSTANCE_NAME_LENGTH`	max. empfohlene Länge eines Instance-Namen (wird nur für formatierte Ausgaben benutzt)
`PART_VALUE_LENGTH`	max. empfohlene Länge eines Bauteil-Values (Instances haben keinen eigenen Value!)

Anmerkung

Das `attributes()`-Member läuft nur durch die Attribute, die explizit dieser Instance zugewiesen wurden (einschließlich *gesmashter* Attribute).

Das `texts()`-Member läuft nur durch die mittels **SMASH** von der Instance losgelösten Texte, und durch die sichtbaren Texte der Attribute, die dieser Instance zugewiesen wurden. Um alle Texte einer Instance zu bearbeiten, müssen Sie eine Schleife durch das `texts()`-Member der Instance selbst und durch das `texts()`-Member des zu dem Gate der Instance gehörenden Symbols bilden. Wurden einer Instance Attribute zugewiesen, so liefert `texts()` deren Texte so, wie sie momentan dargestellt werden.

Die `column()`- und `row()`-Members liefern die Spalten- bzw. Zeilenposition innerhalb des Rahmens auf der Schaltplanseite, auf der diese Instance platziert ist. Falls es auf dieser Seite keinen Rahmen gibt, oder die Instance außerhalb des Rahmens liegt, wird ein `'?'` (Fragezeichen) zurückgegeben. Diese Members können nur in einem UL_SHEET-Kontext verwendet werden.

Das `smashed`-Member gibt Auskunft darüber, ob eine Instance gesmasht ist. Diese Funktion kann auch verwendet werden um herauszufinden, ob es einen losgelösten Platzhaltertext gibt, indem der Name des Platzhalters in eckigen Klammern angegeben wird, wie in `smashed["VALUE"]`. Dies ist nützlich falls Sie einen solchen Text mit dem MOVE-Befehl etwa durch `MOVE R5>VALUE` selektieren wollen. Gültige Platzhalternamen sind "NAME", "VALUE", "Part" und "GATE", sowie die Namen etwaiger benutzerdefinierter Attribute.

Groß-/Kleinschreibung spielt keine Rolle, und sie dürfen ein vorangestelltes `'>'` Zeichen haben.

Das `xrefs()`-Member läuft durch die Gatter des Kontaktspiegels dieser Instance. Diese sind nur dann von Bedeutung, wenn das ULP eine zeichnerische Darstellung des Schaltplans erzeugt (wie etwa eine DXF-Datei).

Beispiel

```
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 (Mittelpunkt)

Siehe auch UL_SEGMENT

Beispiel

```
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 (Aufhängepunkt)

`xref` int (0=normal, 1=Querverweis)

Loop members

`wires()` UL_WIRE (siehe Anmerkung)

Siehe auch UL_SEGMENT

Anmerkung

Falls `xref` ungleich 0 ist, läuft das `wires()` Loop member durch die Wires, aus denen die Umrandung des Querverweis-Labels besteht. Ansonsten ist die Schleife leer.

Die `angle`, `layer`, `mirror` und `spin` Members liefern immer den selben Wert wie diejenigen des UL_TEXT-Objekts, das vom `text` Member geliefert wird. Die `x` und `y` Members des Textes liefern etwas versetzte Werte für Querverweis-Labels `xref` ungleich 0), ansonsten liefern sie die gleichen Werte wie das UL_LABEL.

`xref` hat nur für Netz-Labels eine Bedeutung. Für Bus-Labels liefert es immer 0.

Beispiel

```
sheet(SH) {
  SH.nets(N) {
    N.segments(S) {
      S.labels(L) {
        printf("Label: %d %d '%s'", 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=unbenutzt, 1=benutzt)

`visible` int (0=off, 1=on)

Siehe auch UL_BOARD, UL_LIBRARY, UL_SCHEMATIC

Konstanten

`LAYER_NAME_LENGTH`	max. empfohlene Länge eines Layer-Namens (wird nur für formatierte Ausgaben benutzt)
`LAYER_TOP`	Layer-Nummern
`LAYER_BOTTOM`	
`LAYER_PADS`	
`LAYER_VIAS`	
`LAYER_UNROUTED`	
`LAYER_DIMENSION`	
`LAYER_TPLACE`	
`LAYER_BPLACE`	
`LAYER_TORIGINS`	
`LAYER_BORIGINS`	
`LAYER_TNAMES`	
`LAYER_BNAMES`	
`LAYER_TVALUES`	
`LAYER_BVALUES`	
`LAYER_TSTOP`	
`LAYER_BSTOP`	
`LAYER_TCREAM`	
`LAYER_BCREAM`	

```
LAYER_TFINISH

LAYER_BFINISH

LAYER_TGLUE

LAYER_BGLUE

LAYER_TTEST

LAYER_BTEST

LAYER_TKEEPOUT

LAYER_BKEEPOUT

LAYER_TRESTRICT

LAYER_BRESTRICT

LAYER_VRESTRICT

LAYER_DRILLS

LAYER_HOLES

LAYER_MILLING

LAYER_MEASURES

LAYER_DOCUMENT

LAYER_REFERENCE

LAYER_TDOCU

LAYER_BDOCU

LAYER_NETS

LAYER_BUSSES

LAYER_PINS
```

`LAYER_SYMBOLS`

`LAYER_NAMES`

`LAYER_VALUES`

`LAYER_USER` niedrigste Nummer für benutzerdefinierte Layer (100)

Beispiel

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

UL_LIBRARY

Data members

`description` string (siehe Anmerkung)

`grid` UL_GRID

`headline` string

`name` string (`LIBRARY_NAME_LENGTH`, siehe Anmerkung)

Loop members

`devices()` UL_DEVICE

`devicesets()` UL_DEVICESET

`layers()` UL_LAYER

`packages()` UL_PACKAGE

`symbols()` UL_SYMBOL

Siehe auch UL_BOARD, UL_SCHEMATIC

Konstanten

`LIBRARY_NAME_LENGTH` max. empfohlene Länge eines Bibliotheksnamens (wird nur für formatierte Ausgaben benutzt)

Das `devices()`-Member geht durch alle Package-Varianten und Technologien von UL_DEVICESET in der Bibliothek, so dass alle möglichen Device-Variationen verfügbar werden. Das `devicesets()`-Member geht nur durch die UL_DEVICESETs, die wiederum

nach deren UL_DEVICE-Member abgefragt werden können.

Anmerkung

Das `description`-Member liefert den vollständigen Beschreibungstext, der mit dem DESCRIPTION-Befehl erzeugt wurde, während das `headline`-Member nur die erste Zeile der Beschreibung ohne HTML-Tags ausgibt. Wenn Sie den `description`-Text benutzen, denken Sie daran, dass dieser Newline-Anweisungen (`'\n'`) enthalten darf. Die `description` und `headline` Texte stehen nur direkt innerhalb einer Library-Zeichnung zur Verfügung, nicht wenn die Bibliothek aus einem UL_BOARD- oder UL_SCHEMATIC-Kontext heraus angesprochen wird.

Wird die Bibliothek aus einem UL_BOARD- oder UL_SCHEMATIC-Kontext heraus angesprochen, liefert `name` den reinen Bibliotheksnamen (ohne Extension). Ansonsten wird der volle Bibliotheksname ausgegeben.

Beispiel

```
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 (siehe Anmerkung)
`segments()`	UL_SEGMENT (siehe Anmerkung)

Siehe auch UL_SHEET, UL_SCHEMATIC

Konstanten

`NET_NAME_LENG`	max. empfohlene Länge eines Netznamens (wird nur für formatierte

`TH` Ausgaben benutzt)

Anmerkung

Das Loop member `pinrefs()` kann nur benutzt werden, wenn das Net innerhalb eines UL_SCHEMATIC-Kontexts verwendet wird.
Das Loop member `segments()` kann nur benutzt werden, wenn das Net innerhalb eines UL_SHEET-Kontexts verwendet wird.

Die `column()`- und `row()`-Members liefern die Spalten- bzw. Zeilenpositionen innerhalb des Rahmens auf der Schaltplanseite, auf der dieses Netz liegt. Da ein Netz sich über einen bestimmten Bereich erstrecken kann, liefert jede dieser Funktionen zwei durch ein Leerzeichen getrennte Werte zurück. Im Falle von `column()` sind die die am weitesten links bzw. rechts liegende Spalte, die von diesem Netz berührt wird, und bei `row()` ist es die am weitesten oben bzw. unten liegende Zeile. Falls es auf dieser Seite keinen Rahmen gibt, wird `"? ?"` (zwei Fragezeichen) zurückgegeben. Liegt irgend ein Punkt des Netzes außerhalb des Rahmens, so kann jeder der Werte `'?'` (Fragezeichen) sein. Diese Members können nur in einem UL_SHEET-Kontext verwendet werden.

Beispiel

```
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 (siehe Anmerkung)
`wires()`	UL_WIRE

Siehe auch UL_DEVICE, UL_ELEMENT, UL_LIBRARY

Konstanten

`PACKAGE_NAME_LENGTH`	max. empfohlene Länge eines Package-Namens (wird nur für formatierte Ausgaben benutzt)

Anmerkung

Das `description`-Member liefert den vollständigen Beschreibungstext, der mit dem DESCRIPTION-Befehl erzeugt wurde, während das `headline`-Member nur die erste Zeile der Beschreibung ohne HTML-Tags ausgibt. Wenn Sie `description`-Text schreiben, denken Sie daran, dass dieser Newline-Anweisungen (`'\n'`) enthalten darf.

Stammt das UL_PACKAGE aus einem UL_ELEMENT-Kontext, so durchläuft das `texts()`-Member nur die nicht losgelösten Texte dieses Elements.

Beispiel

```
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 (Mittelpunkt, siehe Anmerkung)

Siehe auch UL_PACKAGE, UL_CONTACT, UL_SMD

Konstanten

PAD_FLAG_STOP	Lötstopmaske generieren
PAD_FLAG_THERMALS	Thermals generieren
PAD_FLAG_FIRST	spezielle Form für "erstes Pad" verwenden
PAD_SHAPE_SQUARE	square
PAD_SHAPE_ROUND	round
PAD_SHAPE_OCTAGON	octagon

PAD_SHAPE_LONG	long
PAD_SHAPE_OFFSET	offset
PAD_SHAPE_ANNULUS	annulus (nur in Verbindung mit Supply-Layern)
PAD_SHAPE_THERMAL	thermal (nur in Verbindung mit Supply-Layern)
PAD_NAME_LENGTH	max. empfohlene Länge eines Pad-Namens (identisch mit CONTACT_NAME_LENGTH)

Anmerkung

Die Parameter des Pads hängen vom Kontext ab in dem es angesprochen wird:

- Wird das Pad aus einem UL_LIBRARY-Kontext angesprochen, sind die Koordinaten (`x, y`) und der Winkel (`angle`) dieselben, wie in der Package-Zeichnung
- In allen anderen Fällen gelten die aktuellen Werte vom Board

Durchmesser und Form des Pads hängen vom Layer ab für den es erzeugt werden soll, da diese Werte, abhängig von den Design Rules, unterschiedlich sein können. Wird als Index für das Data Member "diameter" oder "shape" einer der Layer LAYER_TOP..LAYER_BOTTOM, LAYER_TSTOP oder LAYER_BSTOP angegeben, berechnet sich der Wert nach den Vorgaben der Design Rules. Gibt man LAYER_PADS an, wird der in der Bibliothek definierte Wert verwendet.

`drillsymbol` liefert die Nummer des Bohrsymbols, welches diesem Bohrdurchmesser zugeordnet worden ist (siehe die Liste der definierten Bohrsymbole im Handbuch). Ein Wert von `0` bedeutet, dass diesem Bohrdurchmesser kein Bohrsymbol zugeordnet ist.

`angle` gibt an um wieviel Grad das Pad gegen den Uhrzeigersinn um seinen Mittelpunkt gedreht ist.

`elongation` ist nur für die Pad-Formen PAD_SHAPE_LONG und PAD_SHAPE_OFFSET gültig und bestimmt um wieviel Prozent die lange Seite eines solchen Pads länger ist als seine schmale Seite. Für alle anderen Pad-Formen liefert dieses Member den Wert 0.

Der Wert, den `flags` liefert, muss mit den `PAD_FLAG_...` Konstanten maskiert werden um die einzelnen Flag-Einstellungen zu ermitteln, wie zum Beispiel in

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

Falls Ihr ULP lediglich die Objekte darstellen soll, brauchen Sie sich nicht explizit um diese Flags zu kümmern. Die `diameter[]` und `shape[]` Members liefern die richtigen Daten; ist zum Beispiel `PAD_FLAG_STOP` gesetzt, so liefert `diameter[LAYER_TSTOP]` den Wert `0`, was zur Folge haben sollte, dass in diesem Layer nichts gezeichnet wird. Das `flags`

Member ist hauptsächlich für ULPs gedacht, die Script-Dateien erzeugen mit denen Bibliotheksobjekte kreiert werden.

Beispiel

```
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 (siehe Anmerkung)
`device`	UL_DEVICE
`deviceset`	UL_DEVICESET
`name`	string (`PART_NAME_LENGTH`)
`value`	string (`PART_VALUE_LENGTH`)

Loop members

`attributes()`	UL_ATTRIBUTE (siehe Anmerkung)
`instances()`	UL_INSTANCE (siehe Anmerkung)

Siehe auch UL_SCHEMATIC, UL_SHEET

Konstanten

`PART_NAME_LENGTH`	max. empfohlene Länge eines Part-Namens (wird nur für formatierte Ausgaben benutzt)
`PART_VALUE_LENGTH`	max. empfohlene Länge eines Part-Values (wird nur für formatierte Ausgaben benutzt)

Anmerkung

Mit dem `attribute[]`-Member kann man ein UL_PART nach dem Wert eines bestimmten Attributs fragen (siehe das zweite Beispiel). Der zurückgelieferte String ist leer, wenn es kein Attribut mit dem angegebenen Namen gibt, oder wenn dieses Attribut explizit leer ist.

Beim Durchlaufen der `attributes()` eines UL_PART haben nur die `name`, `value`, `defaultvalue` und `constant` Members des resultierenden UL_ATTRIBUTE gültige Werte.

Wenn sich Part in einem UL_SHEET-Kontext befindet, bearbeitet Loop member `instances()` nur solche Instances, die tatsächlich auf dieser Seite benutzt werden. Wenn sich Part in einem UL_SCHEMATIC-Kontext befindet, geht die Schleife durch alle Instances.

Beispiel

```
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` und `270`)
`contact`	UL_CONTACT (siehe Anmerkung)
`direction`	int (`PIN_DIRECTION_...`)
`function`	int (`PIN_FUNCTION_FLAG_...`)
`length`	int (`PIN_LENGTH_...`)
`name`	string (`PIN_NAME_LENGTH`)
`net`	string (siehe Anmerkung)
`swaplevel`	int
`visible`	int (`PIN_VISIBLE_FLAG_...`)
`x, y`	int (Anschlusspunkt)

Loop members

`circles()`	UL_CIRCLE
`texts()`	UL_TEXT

`wires()` UL_WIRE

Siehe auch UL_SYMBOL, UL_PINREF, UL_CONTACTREF

Konstanten

`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`	Passiv
`PIN_DIRECTION_HIZ`	High-Impedance-Output
`PIN_DIRECTION_SUP`	Supply-Pin
`PIN_FUNCTION_FLAG_NONE`	kein Symbol
`PIN_FUNCTION_FLAG_DOT`	Inverter-Symbol
`PIN_FUNCTION_FLAG_CLK`	Taktsymbol
`PIN_LENGTH_POINT`	kein 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. empfohlene Länge eines Pin-Namens (wird nur für formatierte Ausgaben benutzt)
`PIN_VISIBLE_FLAG_OFF`	kein Name sichtbar
`PIN_VISIBLE_FLAG_PAD`	Pad-Name sichtbar
`PIN_VISIBLE_FLAG_PIN`	Pin-Name sichtbar

Anmerkung

Das `contact` Data Member liefert den Contact, der dem Pin durch einen CONNECT-Befehl zugewiesen worden ist. Es kann als boolsche Function verwendet werden um zu prüfen ob dem Pin ein Contact zugewiesen wurde (siehe Beispiel unten).

Die Koordinaten (und der Layer, im Falle eines SMD) des durch das `contact` Data Member gelieferten Contacts hängen vom Kontext ab, in dem es aufgerufen wird:

- falls der Pin von einem UL_PART stammt, welches auf einer Schaltplanseite verwendet wird, und wenn es ein dazugehöriges Element im Board gibt, dann erhält der Contact die Koordinaten die er im Board hat
- in allen anderen Fällen erhält der Contact die Koordinaten wie sie in der Package-Zeichnung definiert sind

Das `name` Data Member liefert den Namen des Pins immer so, wie er in der Bibliothek definiert wurde, einschließlich eines etwaigen `'@'`-Zeichens für Pins mit dem gleichen Namen (siehe PIN-Befehl).
Das `texts` Loop-Member dagegen liefert den Pin-Namen (sofern er sichtbar ist) immer in der Form, wie er im aktuellen Zeichnungstyp dargestellt wird.

Das `net` Data Member liefert den Namen des Netzes, an das der Pin angeschlossen ist (nur in einem UL_SCHEMATIC-Kontext verfügbar).

Beispiel

```
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

Siehe auch UL_SEGMENT, UL_CONTACTREF

Beispiel

```
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 (siehe Anmerkung)

`fillings()` UL_WIRE

`wires()`	UL_WIRE

Siehe auch UL_BOARD, UL_PACKAGE, UL_SHEET, UL_SIGNAL, UL_SYMBOL

Konstanten

`POLYGON_POUR_SOLID`	solid
`POLYGON_POUR_HATCH`	hatch

Anmerkung

Die Loop-Member `contours()` und `fillings()` gehen durch alle Wires, mit denen das Polygon gezeichnet wird, sofern es zu einem Signal gehört und mit dem Befehl RATSNEST freigerechnet wurde. Das Loop-Member `wires()` geht immer durch die Wires, die vom Benutzer gezeichnet wurden. Für nicht freigerechnete Signal-Polygone liefert `contours()` dasselbe Ergebnis wie `wires()`. `Fillings()` hat dann keine Bedeutung.

Wird das `contours()` Loop-Member ohne einem zweiten Parameter aufgerufen, so läuft es durch alle Umriss-Linien, egal ob sie zu einem positiven oder negativen Polygon gehören. Falls Sie daran interessiert sind, die positiven und negativen Umriss-Linien getrennt voneienander zu erhalten, können Sie `contours()` mit einem zusätzlichen Integer-Parameter aufrufen (siehe zweites Beispiel unten). Das Vorzeichen dieses Parameters bestimmt, ob ein positives oder negatives Polygon behandelt wird, und der Wert gibt den Index dieses Polygons an. Falls es kein Polygon mit dem gegebenen Index gibt, wird die Anweisung nicht ausgeführt. Ein weiterer Vorteil dieser Methode ist, dass Sie Anfang und Ende eines bestimmten Polygons nicht selber (durch Vergleich von Koordinaten) bestimmen müssen. Für jeden Index wird die Anweisung für alle Wires dieses Polygons ausgeführt. Mit `0` als zweitem Parameter ist das Verhalten genau so, als wäre kein zweiter Parameter angegeben worden.

Polygon-Strichstärke

Wenn Sie das Loop-Member `fillings()` verwenden um die Füll-Linien des Polygons zu erreichen, stellen Sie sicher, dass die Strichstärke *width* des Polygons nicht null ist (sie sollte etwas über null liegen, bzw. mindestens der Auflösung des Ausgabetreibers mit dem Sie die Zeichnung ausgeben wollen entsprechen). **Zeichnen Sie ein Polygon mit Strichstärke = 0, ergibt sich eine riesige Datenmenge, da das Polygon mit der kleinsten Editor-Auflösung von 1/10000mm berechnet wird.**

Teilpolygone

Ein berechnetes Polygon kann aus verschiedenen getrennten Teilen (*positive*Polygone genannt) bestehen, wobei jedes davon Aussparungen (*negative* Polygone genannt) enthalten kann, die von anderen Objekten, die vom Polygon subtrahiert werden, herrühren. Negative Polygone können wiederum weitere positive Polygone enthalten und so weiter.

Die Wires, die mit `contours()` erreicht werden, beginnen immer in einem positiven

Polygon. Um herauszufinden wo ein Teilpolygon endet und das nächste beginnt, speichern Sie einfach die Koordinate (x1,y1) des ersten Wires und prüfen diese gegenüber (x2,y2) jedes folgenden Wires. Sobald die beiden Werte identisch sind, ist der letzte Wire des Teilpolygons gefunden. Es gilt immer, dass der zweite Punkt (x2,y2) identisch mit dem ersten Punkt (x1,y1) des nächsten Wires in diesem Teilpolygon ist.

Um herauszufinden ob man innerhalb bzw. ausserhalb der Polygons ist, nehmen Sie einen beliebigen Umriss-Wire und stellen sich Sie vor, von dessen Punkt (x1,y1) zum Punkt (x2,y2) zu sehen. Rechts vom Wire ist immer innerhalb des Polygons. Hinweis: Wenn Sie einfach ein Polygon zeichnen wollen, brauchen Sie all diese Details nicht.

Beispiel

```
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 (linke untere Ecke)
`x2, y2`	int (rechte obere Ecke)

Siehe auch UL_BOARD, UL_PACKAGE, UL_SHEET, UL_SYMBOL

`angle` gibt an um wieviel Grad das Rechteck gegen den Uhrzeigersinn um seinen Mittelpunkt gedreht ist. Der Mittelpunkt ergibt sich aus `(x1+x2)/2` und `(y1+y2)/2`.

Beispiel

```
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
`xreflabel`	string

Loop members

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

Siehe auch UL_BOARD, UL_LIBRARY

Anmerkung

Das `xreflabel` Member liefert den Format-String der für die Darstellung von Querverweis-Labels benutzt wird.

Das Loop member `attributes()` geht durch die *globalen* Attribute.

Beispiel

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

UL_SEGMENT

Loop members

`junctions()`	UL_JUNCTION (siehe Anmerkung)
`labels()`	UL_LABEL
`pinrefs()`	UL_PINREF (siehe Anmerkung)
`texts()`	UL_TEXT (veraltet, siehe Anmerkung)
`wires()`	UL_WIRE

Siehe auch UL_BUS, UL_NET

Anmerkung

Die Loop members `junctions()` und `pinrefs()` sind nur für Netzsegmente zugänglich.

Das Loop member `texts()` wurde in früheren EAGLE-Versionen benutzt um durch die Labels eines Segments zu gehen und ist nur noch aus Kompatibilitätsgründen vorhanden. Es liefert den Text von Querverweis-Labels nicht an der richtigen Position. Benutzen Sie das `labels()` Loop member um die Labels eines Segments anzusprechen.

Beispiel

```
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

Siehe auch UL_SCHEMATIC

Beispiel

```
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

Siehe auch UL_BOARD

Konstanten

`SIGNAL_NAME_LENGTH`	max. empfohlene Länge eines Signalnamens (wird nur für formatierte Ausgaben benutzt)

Beispiel

```
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 (siehe Anmerkung)
`name`	string (`SMD_NAME_LENGTH`)
`roundness`	int (siehe Anmerkung)
`signal`	string
`x, y`	int (Mittelpunkt, siehe Anmerkung)

Siehe auch UL_PACKAGE, UL_CONTACT, UL_PAD

Konstanten

SMD_FLAG_STOP	Lötstopmaske generieren
SMD_FLAG_THERMALS	Thermals generieren
SMD_FLAG_CREAM	Lotpastenmaske generieren
SMD_NAME_LENGTH	max. empfohlenen Länge eines Smd-Namens (identisch mit CONTACT_NAME_LENGTH)

Anmerkung

Die Parameter des SMDs hängen vom Kontext ab in dem es angesprochen wird:

- Wird das Smd aus einem UL_LIBRARY-Kontext angesprochen, entsprechen die Werte der Koordinaten (`x, y`), des Winkels (`angle`) und die Angabe für Layer und Roundness denen in der Package-Zeichnung
- in allen anderen Fällen erhalten Sie die aktuellen Werte aus dem Board

Ruft man die Data Member `dx` und `dy` mit einem optionalen Layer-Index auf, werden die Werte für den zugehörigen Layer, entsprechend den Design Rules ausgegeben. Gültige Layer sind LAYER_TOP, LAYER_TSTOP und LAYER_TCREAM für ein Smd im Top-Layer, und LAYER_BOTTOM, LAYER_BSTOP und LAYER_BCREAM für ein Smd im Bottom-Layer.

`angle` gibt an um wieviel Grad das Smd gegen den Uhrzeigersinn um seinen Mittelpunkt gedreht ist.

Der Wert, den `flags` liefert, muss mit den `SMD_FLAG_...` Konstanten maskiert werden um die einzelnen Flag-Einstellungen zu ermitteln, wie zum Beispiel in

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

Falls Ihr ULP lediglich die Objekte darstellen soll, brauchen Sie sich nicht explizit um diese Flags zu kümmern. Die `dx[]` und `dy[]` Members liefern die richtigen Daten; ist zum Beispiel `SMD_FLAG_STOP` gesetzt, so liefert `dx[LAYER_TSTOP]` den Wert `0`, was zur Folge haben sollte, dass in diesem Layer nichts gezeichnet wird. Das `flags` Member ist hauptsächlich für ULPs gedacht, die Script-Dateien erzeugen mit denen Bibliotheksobjekte kreiert werden.

Beispiel

```
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 (siehe Anmerkung)
`wires()`	UL_WIRE

Siehe auch UL_GATE, UL_LIBRARY

Konstanten

`SYMBOL_NAME_LENGTH`	max. empfohlene Länge eines Symbol-Namens (wird nur für formatierte Ausgaben benutzt)

Anmerkung

Stammt das UL_SYMBOL aus einem UL_INSTANCE-Kontext, so durchläuft das `texts()`-Member nur die nicht losgelösten Texte dieser Instance.

Beispiel

```
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 (Aufhängepunkt)

Loop members

`wires()`	UL_WIRE (siehe Anmerkung)

Siehe auch UL_BOARD, UL_PACKAGE, UL_SHEET, UL_SYMBOL

Konstanten

`FONT_VECTOR`	Vector-Font
`FONT_PROPORTIONAL`	Proportional-Font
`FONT_FIXED`	Fixed-Font

Anmerkung

Das Loop-Member `wires()` greift immer auf die individuellen Wires, aus denen der Text im Vektor-Font zusammengesetzt wird, zu. Auch dann, wenn der aktuelle Font nicht `FONT_VECTOR` ist.

Wurde der UL_TEXT aus einem UL_ELEMENT- oder UL_INSTANCE-Kontext angesprochen, so liefern die Members die tatsächlichen Werte, so wie sie in der Board- oder Schaltplan-Zeichnung zu finden sind.

Beispiel

```
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 (Mittelpunkt)

Siehe auch UL_SIGNAL

Konstanten

`VIA_FLAG_STOP`	Lötstopmaske immer generieren
`VIA_SHAPE_SQUARE`	square
`VIA_SHAPE_ROUND`	round
`VIA_SHAPE_OCTAGON`	octagon
`VIA_SHAPE_ANNULUS`	annulus
`VIA_SHAPE_THERMAL`	thermal

Anmerkung

Der Durchmesser und die Form des Vias hängen davon ab für welchen Layer es gezeichnet werden soll, denn es können in den Design Rules unterschiedliche Werte definiert werden. Gibt man einen der Layer LAYER_TOP..LAYER_BOTTOM, LAYER_TSTOP oder LAYER_BSTOP als Index für diameter oder shape an, wird das Via entsprechend den Vorgaben aus den Design Rules berechnet. Wird LAYER_VIAS angegeben, wird der ursprüngliche Wert mit dem das Via definiert wurde, verwendet.

Beachten Sie bitte, dass `diameter` und `shape` auf jeden Fall den Durchmesser bzw. die Form zurückliefern, welche ein Via in dem gegebenen Layer hätte, selbst wenn das konkrete Via diesen Layer gar nicht überdeckt (oder wenn dieser Layer im Layer-Setup überhaupt

nicht benutzt wird).

`start` und `end` liefern den Layer, in dem dieses Via beginnt bzw. endet. Der Wert von `start` ist dabei immer kleiner als der von `end`.

`drillsymbol` liefert die Nummer des Bohrsymbols, welches diesem Bohrdurchmesser zugeordnet worden ist (siehe die Liste der definierten Bohrsymbole im Handbuch). Ein Wert von `0` bedeutet, dass diesem Bohrdurchmesser kein Bohrsymbol zugeordnet ist.

Beispiel

```
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 (Anfangspunkt)
`x2, y2`	int (Endpunkt)

Loop members

`pieces()`	UL_WIRE (siehe Anmerkung)

Siehe auch UL_BOARD, UL_PACKAGE, UL_SEGMENT, UL_SHEET, UL_SIGNAL, UL_SYMBOL, UL_ARC

Konstanten

`CAP_FLAT`	flache Kreisbogen-Enden
`CAP_ROUND`	runde Kreisbogen-Enden

`WIRE_STYLE_CONTINUOUS`	durchgezogen
`WIRE_STYLE_LONGDASH`	lang gestrichelt
`WIRE_STYLE_SHORTDASH`	kurz gestrichelt
`WIRE_STYLE_DASHDOT`	Strich-Punkt-Linie

Wire Style

Bei einem UL_WIRE mit anderem *style* als `WIRE_STYLE_CONTINUOUS`, kann über das Loop-Member `pieces()` auf die individuellen Teile, die z. B. eine gestrichelte Linie darstellen, zugegriffen werden. Wenn `pieces()` für UL_WIRE mit `WIRE_STYLE_CONTINUOUS` aufgerufen wird, erhält man ein Segment, das genau dem original UL_WIRE entspricht. Das Loop-Member `pieces()` kann nicht von UL_WIRE aus aufgerufen werden, wenn dieser selbst schon über `pieces()` aufgerufen wurde (das würde eine unendliche Schleife verursachen).

Kreisbögen auf Wire-Ebene

Kreisbögen sind zunächst einfach nur Wires, mit einigen zusätzlichen Eigenschaften. In erster Näherung werden Kreisbögen genauso behandelt wie Wires, das heisst sie haben einen Anfangs- und Endpunkt, eine Breite und einen Linientyp. Hinzu kommen auf Wire-Ebene die Parameter *cap* und *curve*. *cap* gibt an ob die Kreisbogen-Enden rund oder flach sind, und *curve* bestimmt die "Krümmung" des Kreisbogens. Der gültige Bereich für *curve* ist `-360..+360`, wobei der Wert angibt aus welchem Anteil eines Vollkreises der Kreisbogen besteht. Ein Wert von `90` beispielsweise steht für einen Viertelkreis, während `180` einen Halbkreis ergibt. Der maximale Wert von `360` kann nur theoretisch erreicht werden, da dies bedeuten würde, dass der Kreisbogen aus einem vollen Kreis besteht, der, weil Anfangs- und Endpunkt auf dem Kreis liegen müssen, einen unendlich großen Durchmesser haben müsste. Positive Werte für *curve* bedeuten, dass der Kreisbogen im mathematisch positiven Sinne (also gegen den Uhrzeigersinn) gezeichnet wird. Falls *curve* gleich `0` ist, handelt es sich um eine gerade Linie ("keine Krümmung"), was letztlich einem Wire entspricht.

Der *cap* Parameter ist nur für echte Kreisbögen von Bedeutung und liefert für gerade Wires immer `CAP_ROUND`.

Ob ein UL_WIRE ein Kreisbogen ist oder nicht kann durch Abfragen des boolschen Rückgabewertes des `arc` Data Members herausgefunden werden. Falls dieses `0` liefert, liegt ein gerader Wire vor, ansonsten ein Kreisbogen. Liefert `arc` nicht `0` so darf es weiter dereferenziert werden um die für einen UL_ARC spezifischen Parameter Start- und Endwinkel, Radius und Mittelpunkt zu erfragen. Diese zusätzlichen Parameter sind normalerweise nur von Bedeutung wenn der Kreisbogen gezeichnet oder anderweitig verarbeitet werden soll, und dabei die tatsächliche Form eine Rolle spielt.

Beispiel

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

Definitionen

Konstanten, Variablen und Funktionen müssen definiert werden, bevor sie in einem User-Language-Programm verwendet werden können.

Es gibt drei Arten von Definitionen:

- Konstanten-Definitionen
- Variablen-Definitionen
- Funktions-Definitionen

Der Gültigkeitsbereich einer *Konstanten-* oder *Variablen*-Definition reicht von der Zeile, in der sie definiert wurde, bis zum Ende des gegenwärtigen Blocks, oder bis zum Ende des User-Language-Programms, wenn die Definition ausserhalb aller Blöcke steht.

Der Gültigkeitsbereich einer *Funktions*-Definition reicht von der schließenden geschweiften Klammer (`}`) des Funktionsrumpfes bis zum Ende des User-Language-Programms.

Konstanten-Definitionen

Konstanten werden mit Hilfe des Schlüsselworts `enum` definiert, wie in

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

womit man den drei Konstanten `a`, `b` und `c` die Werte `0`, `1` und `2` zuweisen würde.

Konstanten kann man auch mit bestimmten Werten initialisieren, wie in

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

wo `a` den Wert `0`, `b` den Wert `5` und `c` den Wert `6` erhält.

Variablen-Definitionen

Die allgemeine Syntax einer *Variablen-Definition* ist

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

wobei `type` ein Daten- oder Objekt-Typ ist, `identifier` ist der Name der Variablen, und `initializer` ist ein optionaler Initialisierungswert.

Mehrfach-Variablen-Definitionen desselben `Typs` werden durch Kommas (`,`) getrennt.

Wenn auf `identifier` ein Paar eckiger Klammern (`[]`) folgt, wird ein Array von Variablen des gegebenen `Typs` definiert. Die Größe des Arrays wird zur Laufzeit automatisch bestimmt.

Das optionale Schlüsselwort `numeric` kann mit String-Arrays verwendet werden, um sie alphanumerisch mit der Funktion sort() sortieren zu lassen.

Standardmäßig (wenn kein `Initializer` vorhanden ist), werden Daten-Variablen auf `0` gesetzt (oder `""`, falls es sich um einen String handelt), und Objekt -Variablen werden mit "invalid" initialisiert.

Beispiele

`int i;`	definiert eine int-Variable mit dem Namen `i`
`string s = "Hello";`	definiert eine string-Variable mit dem Namen `s` und initialisiert sie mit `"Hello"`
`real a, b = 1.0, c;`	definiert drei real-Variablen mit den Namen `a`, `b` und `c` und initialisiert `b` mit dem Wert `1.0`
`int n[] = { 1, 2, 3 };`	definiert ein Array of int und initialisiert die ersten drei Elemente mit `1`, `2` und `3`
`numeric string names[];`	definiert ein string-Array das alphanumerisch sortiert werden kann
`UL_WIRE w;`	definiert ein UL_WIRE-Objekt mit dem Namen `w`

Die Members von Elementen eines Arrays von Objekt-Typen können nicht direkt angesprochen werden:

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

Funktions-Definitionen

Sie können Ihre eigenen User-Language-Funktionen schreiben und sie genau so aufrufen wie Builtin-Functions.

Die allgemeine Syntax einer *Funktions-Definition* lautet

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

wobei `type` ein Daten- oder Objekt-Typ ist, `identifier` der Name einer Funktion, `parameters` eine durch Kommas getrennte Liste von Parameter-Definitionen und `statements` eine Reihe von Statements.

Funktionen die keinen Wert zurückgeben, haben den Typ `void`.

Eine Funktion muss definiert werden, **bevor** sie aufgerufen werden kann, und Funktionsaufrufe können nicht rekursiv sein (eine Funktion kann sich nicht selbst aufrufen).

Die Statements im Funktionsrumpf können die Werte der Parameter ändern, das hat aber keinen Einfluss auf die Argumente des Funktionsaufrufs.

Die Ausführung einer Funktion kann mit dem `return`-Statement beendet werden. Ohne `return`-Statement wird der Funktionsrumpf bis zu seiner schließenden geschweiften Klammer (`}`) ausgeführt.

Ein Aufruf der `exit()`-Funktion beendet das gesamte User-Language-Programm.

Die spezielle Funktion `main()`

Wenn Ihr User-Language-Programm eine Funktion namens `main()` enthält, wird diese Funktion explizit als Hauptfunktion aufgerufen. Ihr Rückgabewert ist der Rückgabewert des Programms.

Kommandozeilen-Argumente sind für das Programm über die globalen Builtin-Variablen `argc` und `argv` verfügbar.

Beispiel

```
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));
  }
```

Operatoren

Die folgende Tabelle listet alle User-Language-Operatoren in der Reihenfolge ihrer Priorität auf (*Unary* hat die höchste Priorität, *Comma* die niedrigste):

Unary	`! ~ + - ++ --`
Multiplicative	`* / %`
Additive	`+ -`
Shift	`<< >>`
Relational	`< <= > >=`

Equality	`== !=`
Bitwise AND	`&`
Bitwise XOR	`^`
Bitwise OR	`|`
Logical AND	`&&`
Logical OR	`||`
Conditional	`?:`
Assignment	`= *= /= %= += -= &= ^= |= <<= >>=`
Comma	`,`

Die Assoziativität ist **links nach rechts** für alle Operatoren ausser für *Unary*, *Conditional* und *Assignment*, die **rechts-nach-links**-assoziativ sind.

Die normale Operator-Priorität kann durch den Gebrauch von runden Klammern geändert werden.

Bitweise Operatoren

Bitweise Operatoren kann man nur auf die Datentypen `char` und `int` anwenden.

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

Logische Operatoren

Logische Operatoren arbeiten mit Ausdrücken von jedem Datentyp.

Unary

`!` Logical NOT

Binary

`&&` Logical AND

`||` Logical OR

Die Verwendung eines `String`-Ausdrucks mit einem logischen Operator prüft, ob ein String leer ist.

Die Verwendung eines Objekt-Typs mit einem logischen Operator prüft, ob dieses Objekt gültige Daten enthält.

Vergleichs-Operatoren

Vergleichs-Operatoren können mit Ausdrücken von jedem Datentyp angewendet werden, ausgenommen Objekt-Typen.

`<` Kleiner als

`<=` Kleiner gleich

`>` Größer als

`>=` Größer gleich

`==` Gleich

`!=` Ungleich

Evaluation-Operatoren

Evaluation-Operatoren werden verwendet, um Ausdrücke auszuwerten, die auf einer Bedingung basieren, oder um eine Sequenz von Ausdrücken zu gruppieren und sie als einen Ausdruck auszuwerten.

`?:` Conditional

`,` Komma

Der *Conditional*-Operator wird verwendet, um eine Entscheidung innerhalb eines Ausdrucks zu treffen, wie in

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

was folgender Konstruktion entspricht

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

aber der Vorteil des Conditional-Operators ist, dass er innerhalb des Ausdrucks verwendet werden kann.

Der *Komma*-Operator wird verwendet, um eine Sequenz von Ausdrücken von links nach rechts auszuwerten; Typ und Wert des rechten Operanden werden als Ergebnis verwendet.

Beachten Sie, dass Argumente in einem Funktionsaufruf und Mehrfach-Variablen-Deklarationen ebenfalls Kommas als Trennzeichen verwenden. Dabei handelt es sich aber **nicht** um den Komma-Operator!

Arithmetische Operatoren

Arithmetische Operatoren lassen sich auf die Datentypen `char`, `int` und `real` anwenden (ausser `++`, `--`, `%` und `%=`).

Unary

`+` Unary plus

`-` Unary minus

`++` Pre- oder postincrement

`--` Pre- oder postdecrement

Binary

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

Assignment

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

Siehe auch String-Operatoren

String-Operatoren

String-Operatoren lassen sich mit den Datentypen `char`, `int` und `string` anwenden. Der linke Operand muss immer vom Typ `string` sein.

Binary

+	Concatenation

Assignment

=	Simple assignment
+=	Append to string

Der +-Operator fasst zwei Strings zusammen oder fügt ein Zeichen am Ende eines Strings hinzu und gibt den resultierenden String zurück.

Der +=-Operator fügt einen String oder eine Zeichen an das Ende eines gegebenen Stings

an.

Siehe auch Arithmetische Operatoren

Ausdrücke

Es gibt folgende *Ausdrücke:*

- Arithmetischer Ausdruck
- Zuweisungs-Ausdruck
- String-Ausdruck
- Komma-Ausdruck
- Bedingter Ausdruck
- Funktionsaufruf

Ausdrücke können mit Hilfe von runden Klammern gruppiert werden und dürfen rekursiv aufgerufen werden, was bedeutet, dass ein Ausdruck aus Unterausdrücken bestehen darf.

Arithmetischer Ausdruck

Ein *arithmetischer Ausdruck* ist jede Kombination von numerischen Operanden und arithmetischem Operator oder bitweisem Operator.

Beispiele

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

Zuweisungs-Ausdruck

Ein *Zuweisungs-Ausdruck* besteht aus einer Variablen auf der linken Seite eines Zuweisungsoperators und einem Ausdruck auf der rechten Seite.

Beispiele

```
a = x + 42
b += c
s = "Hello"
```

String-Ausdruck

Ein *String-Ausdruck* ist jede Kombination von string- und char- Operanden und einem String-Operator.

Beispiele

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

Komma-Ausdruck

Ein *Komma-Ausdruck* ist eine Sequenz von Ausdrücken, die mit dem Komma-Operator abgegrenzt werden.

Komma-Ausdrücke werden von links nach rechts ausgewertet, und das Ergebnis eines Komma-Ausdrucks ist der Typ und der Wert des am weitesten rechts stehenden Ausdrucks.

Beispiel

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

Bedingter Ausdruck

Ein *bedingter Ausdruck* verwendet den Conditional-Operator, um eine Entscheidung innerhalb eines Ausdrucks zu treffen.

Beispiel

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

Funktionsaufruf

Ein *Funktionsaufruf* transferiert den Programmfluss zu einer benutzerdefinierten Funktion oder einer Builtin-Function. Die formalen Parameter, die in der Funktions-Definition definiert sind, werden ersetzt durch die Werte der Ausdrücke, die als aktuelle Argumente des Funktionsaufrufs dienen.

Beispiel

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

Statements

Ein *Statement* kann folgendes sein:

- Compound-Statement (Verbundanweisung)
- Control-Statement (Steueranweisung)
- Expression-Statement (Ausdrucksanweisung)
- Builtin-Statement
- Konstanten-Definition
- Variablen-Definition

Statements spezifizieren die Programmausführung. Wenn keine Control-Statements vorhanden sind, werden Statements der Reihe nach in der Reihenfolge ihres Auftretens in der ULP-Datei ausgeführt.

Compound-Statement (Verbundanweisung)

Ein *Compound-Statement* (auch bekannt als *Block*) ist eine Liste (kann auch leer sein) von Statements in geschweiften Klammern (`{}`). Syntaktisch kann ein Block als einzelnes Statement angesehen werden, aber er steuert auch den Gültigkeitsbereich von Identifiern. Ein Identifier, der innerhalb eines Blocks deklariert wird, gilt ab der Stelle, an der er definiert wurde, bis zur schließenden geschweiften Klammer.

Compound-Statements können beliebig verschachtelt werden.

Expression-Statement (Ausdrucksanweisung)

Ein *Expression-Statement* ist jeder beliebige Ausdruck, gefolgt von einem Semikolon.

Ein Expression-Statement wird ausgeführt, indem der Ausdruck ausgewertet wird. Alle Nebeneffekte dieser Auswertung sind vollständig abgearbeitet, bevor das nächste Statement ausgeführt wird. Die meisten Expression-Statements sind Zuweisungen oder Funktionsaufrufe.

Ein Spezialfall ist das *leere Statement*, das nur aus einem Semikolon besteht. Ein leeres Statement tut nichts, aber es ist nützlich in den Fällen, in denen die ULP-Syntax ein Statement erwartet, aber Ihr Programm keines benötigt.

Control-Statements (Steueranweisungen)

Control-Statements werden verwendet, um den Programmfluss zu steuern.

Iteration-Statements sind

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

Selection-Statements sind

`if...else`
`switch`

Jump-Statements sind

`break`
`continue`
`return`

break

Das *break*-Statement hat die allgemeine Syntax

```
break;
```

und bricht sofort das **nächste** einschließende do...while-, for-, switch- oder while-Statement ab. Dies gilt ebenso für *loop members* von Objekt-Typen.

Da all diese Statements gemischt und verschachtelt werden können, stellen Sie bitte sicher,

dass `break` vom korrekten Statement aus ausgeführt wird.

continue

Das *continue*-Statement hat die allgemeine Syntax

```
continue;
```

und transferiert die Steuerung direkt zur Testbedingung des **nächsten** einschließenden do...while-, while-, oder for-Statements oder zum Increment-Ausdruck des **nächsten** einschließenden for-Statements.

Da all diese Statements gemischt und verschachtelt werden können, stellen Sie bitte sicher, dass `continue` das richtige Statement betrifft.

do...while

Das *do...while*-Statement hat die allgemeine Syntax

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

und führt das `statement` aus, bis der `condition`-Ausdruck null wird.

`condition` wird **nach** der ersten Ausführung von `statement` getestet, was bedeutet, dass das Statement wenigstens einmal ausgeführt wird.

Wenn kein `break` oder `return` im `statement` vorkommt, muss das `statement` den Wert der `condition` verändern, oder `condition` selbst muss sich während der Auswertung ändern, um eine Endlosschleife zu vermeiden.

Beispiel

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

for

Das *for*-Statement hat die allgemeine Syntax

```
for ([init]; [test]; [inc])-Statement
```

und führt folgende Schritte aus:

1. Wenn es einen Initialisierungs-Ausdruck `init` gibt, wird er ausgeführt.
2. Wenn es einen `test`-Ausdruck gibt, wird er ausgeführt. Wenn das Ergebnis ungleich null ist (oder wenn es keinen `test`-Ausdruck gibt), wird das `statement` ausgeführt.
3. Wenn es einen `inc`-Ausdruck gibt, wird er ausgeführt.
4. Schließlich wird die Programmsteuerung wieder an Schritt 2 übergeben.

Wenn es kein break oder return im statement gibt, muss der inc-Ausdruck (oder das statement) den Wert des test-Ausdrucks beeinflussen, oder test selbst muss sich während der Auswertung ändern, um eine Endlosschleife zu vermeiden.

Der Initialisierungs-Ausdruck init initialsiert normalerweise einen oder mehrere Schleifenzähler. Er kann auch eine neue Variable als Schleifenzähler definieren. Eine solche Variable ist bis zum Ende des aktiven Blocks gültig.

Beispiel

```
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

Das *if...else*-Statement hat die allgemeine Syntax

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

Der bedingte Ausdruck wird ausgewertet und, wenn der Wert ungleich null ist, wird t_statement ausgeführt. Andernfalls wird f_statement ausgeführt, sofern der else-Teil vorhanden ist.

Der else-Teil bezieht sich immer auf das letzte if ohne else. Wenn Sie etwas anderes wollen, müssen Sie geschweifte Klammern verwenden, um die Statements zu gruppieren, wie in

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

return

Eine Funktion mit einem Return-Typ ungleich void muss mindestens ein *return*-Statement mit der Syntax

```
return expression;
```

enthalten, wobei die Auswertung von expression einen Wert ergeben muss, der kompatibel ist mit dem Return-Typ der Funktion.

Wenn die Funktion vom Typ void ist, kann ein return-Statement ohne expression verwendet werden, um vom Funktionsaufruf zurückzukehren.

switch

Das *switch*-Statement hat die allgemeine Syntax

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

und erlaubt die Übergabe der Steuerung an eines von mehreren `case`-Statements (mit "case" als Label), abhängig vom Wert des Ausdrucks `sw_exp` (der vom Integral-Typ sein muss).

Jedes `case_statement` kann mit einem oder mehreren `case`-Labels versehen sein. Die Auswertung des Ausdrucks `case_exp` jedes `case`-Labels muss einen konstanten Integer-Wert ergeben, der innerhalb des umschließenden `switch`-Statements nur einmal vorkommt.

Es darf höchstens ein `default`-Label vorkommen.

Nach der Auswertung von `sw_exp` werden die `case_exp`-Ausdrücke auf Übereinstimmung geprüft. Wenn eine Übereinstimmung gefunden wurde, wird die Steuerung zum `case_statement` mit dem entsprechenden `case`-Label transferiert.

Wird keine Übereinstimmung gefunden und gibt es ein `default`-Label, dann erhält `def_statement` die Steuerung. Andernfalls wird kein Statement innerhalb der `switch`-Anweisung ausgeführt.

Die Programmausführung wird nicht beeinflusst, wenn `case`- und `default`-Labels auftauchen. Die Steuerung wird einfach an das folgende Statement übergeben.

Um die Programmausführung am Ende einer Gruppe von Statements für ein bestimmtes `case` zu stoppen, verwenden Sie das break-Statement.

Beispiel

```
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

Das *while*-Statement hat die allgemeine Syntax

```
while (condition) statement
```

und führt `statement` so lange aus, bis der `condition`-Ausdruck ungleich null ist.

`condition` wird **vor** der erstmöglichen Ausführung von `statement` getestet, was bedeutet, dass das Statement überhaupt nicht ausgeführt wird, wenn `condition` von Anfang an null ist.

Wenn kein `break` oder `return` im `statement` vorkommt, muss das `statement` den Wert der `condition` verändern, oder `condition` selbst muss sich während der Auswertung ändern, um eine Endlosschleife zu vermeiden.

Beispiel

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

Builtins

Builtins sind *Konstanten*, *Variablen*, *Funktionen* und *Statements*, die zusätzliche Informationen liefern und die Manipulation der Daten erlauben.

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

Builtin-Constants

Builtin-Constants liefern Informationen über Objekt-Parameter, wie die maximale empfohlene Namenslänge, Flags und so weiter.

Viele Objekt-Typen haben ihren eigenen **Konstanten**-Bereich, in dem die Builtin-Constants für das betreffende Objekt aufgelistet sind (siehe z.B. UL_PIN).

Die folgenden Builtin-Constants sind zusätzlich zu denen definiert, die für die einzelnen Objekt-Typen aufgeführt sind:

`EAGLE_VERSION`	EAGLE-Programm-Versionsnummer (int)
`EAGLE_RELEASE`	EAGLE-Programm-Release-Nummer (int)
`EAGLE_SIGNATURE`	ein String der EAGLE-Programmnamen, -Version und -Copyright-Information enthält
`REAL_EPSILON`	die minimale positive real Zahl, so dass `1.0 + REAL_EPSILON != 1.0`
`REAL_MAX`	der größte mögliche real Wert

`REAL_MIN`	der kleinste mögliche (positive!) real Wert die kleinste darstellbare Zahl ist `-REAL_MAX`
`INT_MAX`	der größte mögliche int Wert
`INT_MIN`	der kleinste mögliche int Wert
`PI`	der Wert von "pi" (3.14..., real)
`usage`	ein string der den Text der `#usage`-Direktive enthält

Diese Builtin-Constants enthalten die Directory-Pfade, die im Directories-Dialog definiert wurden, wobei etwaige spezielle Variablen (`$HOME` und `$EAGLEDIR`) durch ihre aktuellen Werte ersetzt wurden. Da jeder Pfad aus mehreren Directories bestehen kann, sind diese Konstanten string-Arrays mit jeweils einem einzelnen Directory in jedem Eintrag. Der erste leere Eintrag bedeutet das Ende des Pfades:

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

Wenn Sie diese Konstanten dazu verwenden, einen vollständigen Dateinamen zu bilden, so müssen Sie ein Directory-Trennzeichen benutzen, wie etwa in

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

Die im Moment durch den USE-Befehl benutzten Bibliotheken:

```
used_libraries[]
```

Builtin Variablen

Builtin-Variablen werden verwendet, um zur Laufzeit Informationen zu erhalten.

`int argc`	Anzahl der Argumente, die an den RUN Befehl übergeben wurden
`string`	Argumente, die an den RUN-Befehl übergeben wurden (`argv[0]` ist der

`argv[]` volle ULP-Datei-Name)

Builtin-Functions

Builtin-Functions werden für spezielle Aufgaben benötigt, z.B. formatierte Strings drucken, Daten-Arrays sortieren o.ä.

Sie können auch eigene Funktionen definieren und sie dazu verwenden, um Ihre User-Language-Programme zu strukturieren.

Builtin-Functions sind in folgende Kategorien eingeteilt:

- Character-Funktionen
- File-Handling-Funktionen
- Mathematische Funktionen
- Verschiedene Funktionen
- Printing-Funktionen
- String-Funktionen
- Zeit-Funktionen
- Objekt-Funktionen

Alphabetische Auflistung aller Builtin-Functions:

- abs()
- acos()
- asin()
- atan()
- ceil()
- cos()
- exit()
- exp()
- 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()
- min()
- palette()
- pow()
- printf()
- round()
- sin()
- sort()
- sprintf()
- sqrt()
- status()
- strchr()
- strjoin()
- strlen()
- strlwr()
- strrchr()
- strrstr()
- strsplit()
- strstr()
- strsub()
- strtod()
- strtol()
- strupr()
- system()
- t2day()
- t2dayofweek()
- t2hour()
- t2minute()
- t2month()
- t2second()
- t2string()
- t2year()
- tan()
- time()
- tolower()
- toupper()
- trunc()

- u2inch()
- u2mic()
- u2mil()
- u2mm()

Character-Funktionen

Mit *Character-Funktionen* manipuliert man einzelne Zeichen.

Die folgenden Character-Funktionen sind verfügbar:

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

is...()

Funktion

Prüfen, ob ein Zeichen in eine bestimmte Kategorie fällt.

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);
```

Rückgabewert

Die `is...`-Funktionen liefern einen Wert ungleich null, wenn das Zeichen in die Kategorie fällt, sonst null.

Character-Kategorien

`isalnum`	Buchstaben (`A` bis `Z` oder `a` bis `z`) oder Digits (`0` bis `9`)

`isalpha`	Buchstaben (`A` bis `Z` oder `a` bis `z`)
`iscntrl`	Delete-Zeichen oder normale Steuerzeichen (`0x7F` oder `0x00` bis `0x1F`)
`isdigit`	Digits (`0` bis `9`)
`isgraph`	Druckbare Zeichen (ausser Leerzeichen)
`islower`	Kleinbuchstaben (`a` bis `z`)
`isprint`	Druckbare Zeichen (`0x20` bis `0x7E`)
`ispunct`	Punctuation-Zeichen (`iscntrl` oder `isspace`)
`isspace`	Space, Tab, Carriage Return, New Line, Vertical Tab oder Formfeed (`0x09` bis `0x0D`, `0x20`)
`isupper`	Großbuchstaben (`A` bis `Z`)
`isxdigit`	Hex-Digits (`0` bis `9`, `A` bis `F`, `a` bis `f`)

Beispiel

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

to...()

Funktion

Buchstaben in Groß- oder Kleinbuchstaben umwandeln.

Syntax

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

Rückgabewert

Die `tolower`-Funktion gibt den konvertierten Buchstaben zurück, wenn `c` ein Großbuchstabe ist. Alle anderen Zeichen werden unverändert zurückgegeben. Die `toupper`-Funktion gibt den konvertierten Buchstaben zurück, wenn `c` ein Kleinbuchstabe ist. Alle anderen Zeichen werden unverändert zurückgegeben.

Siehe auch strupr, strlwr

Datei-Funktionen

Datei-Funktionen behandeln Datei-Namen, -Größen und -Zeitstempel.

Folgende Datei-Funktionen sind verfügbar:

- fileerror()
- fileglob()
- filedir()
- fileext()
- filename()
- fileread()
- filesetext()
- filesize()
- filetime()

Weitere Informationen über Ausgaben in eine Datei, finden Sie unter output().

fileerror()

Funktion

Zeigt den Status von I/O-Operationen.

Syntax

```
int fileerror();
```

Rückgabewert

Gibt die `fileerror`-Funktion `0` zurück, ist alles in Ordnung.

Siehe auch output, printf, fileread

`fileerror` prüft den Status beliebiger I/O-Operation, die seit dem letzten Aufruf dieser Funktion ausgeführt wurden und gibt `0` zurück, wenn alles in Ordnung war. Verursachte eine der I/O-Operationen einen Fehler, wird ein Wert ungleich `0` ausgegeben.

Vor der Ausführung von I/O-Operationen sollten Sie mit `fileerror` den Fehlerstatus zurücksetzten. Nach der Ausführung der I/O-Operationen rufen Sie `fileerror`erneut auf, um zu prüfen ob alles in Ordnung war.

Wenn `fileerror` einen Wert ungleich `0` ausgibt (und so einen Fehler anzeigt), wird dem Benutzer eine Fehlermeldung angezeigt.

Beispiel

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

fileglob()

Funktion

Sucht in einem Verzeichnis.

Syntax

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

Rückgabewert

Die Funktion `fileglob` liefert die Anzahl der Einträge, die in `array` kopiert wurden.

Siehe auch dlgFileOpen(), dlgFileSave()

`fileglob` sucht in einem Verzeichnis nach `pattern`.

`pattern` kann `'*'` und `'?'` als Platzhalter enthalten. Endet `pattern` mit einem `'/'`, wird der Inhalt des angegebenen Verzeichnis zurückgegeben.

Namen die im resultierenden `array` mit einem `'/'` enden, sind Verzeichnisnamen.

Das `array` ist alphabetisch sortiert, die Verzeichnisse kommen zuerst.

Die Sondereinträge `'.'` und `'..'` (für das aktuelle und das übergeordnete Verzeichnis) werden nie in `array` geschrieben.

Wenn `pattern` nicht gefunden wird, oder wenn Sie kein Recht haben, das angegebene Verzeichnis zu durchsuchen, ist das `array` leer.

Hinweis für Windows-Anwender

Das Pfad-Trennzeichen in `array` ist immer ein **Forward-Slash** (Schrägstrich). So ist sichergestellt, dass User-Language-Progamme betriebssystemunabhängig arbeiten. In `pattern` wird der **backslash** (`'\'`) auch als Pfad-Trennzeichen behandelt.

Die Sortierreihenfolge unter Windows unterscheidet nicht zwischen Groß- und Kleinschreibung.

Beispiel

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

Dateinamens-Funktionen

Funktion

Datei-Namen in seine Einzelteile aufspalten.

Syntax

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

Rückgabewert

`filedir` liefert das Directory von `file` (einschließlich Laufwerksbuchstaben unter Windows).

`fileext` liefert die Extension von `file`.

`filename` liefert den File-Namen von `file` (einschließlich Extension).

`filesetext` liefert `file` mit Extension auf `newext` gesetzt.

Siehe auch Datei-Daten-Funktionen

Beispiel

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

Datei-Daten-Funktionen

Funktion

Holt den Timestamp und die Größe einer Datei.

Syntax

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

Rückgabewert

`filesize` liefert die Größe (in Byte) der Datei.

`filetime` liefert den Timestamp der Datei in einem Format, das mit den Zeit-Funktionen benutzt wird.

Siehe auch time, Dateinamens-Funktionen

Beispiel

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

Datei-Einlese-Funktionen

Datei-Einlese-Funktionen werden verwendet um Daten von Dateien einzulesen.

Folgendes Datei-Einlesen ist möglich:

- fileread()

Siehe output() für Informationen zum Thema 'In eine Datei schreiben'.

fileread()

Funktion

Liest Daten aus einer Datei aus.

Syntax

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

Rückgabewert

`fileread` liefert die Anzahl der Objekte, die aus einer Datei ausgelesen wurden.

Die tatsächliche Bedeutung des Rückgabewerts hängt vom `dest`-Typ ab.

Siehe auch lookup, strsplit, fileerror

Wenn `dest` ein Character-Array ist, werden Binär-Daten aus der Datei ausgelesen. Der Rückgabewert entspricht dann der Anzahl der Bytes, die in das Character-Array eingelesen wurden (das entspricht der Dateigröße).

Wenn `dest` ein String-Array ist, wird die Datei als Textdatei gelesen (eine Zeile pro Array-Member). Der Rückgabewert zeigt die Anzahl der Zeilen, die in das Array eingelesen wurden. Newline-Zeichen werden nicht berücksichtigt.

Wenn `dest` ein String ist, wird die ganze Datei in diesen String eingelesen. Der Rückgabewert ist die Länge des Strings (die nicht unbedingt der Dateigröße entsprechen muss, wenn das Betriebssystem Textdateien mit "cr/lf" anstatt "newline" am Zeilenende speichert).

Beispiel

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

Mathematische Funktionen

Mathematische Funktionen werden dazu verwendet, matematische Operationen auszuführen.

Die folgenden mathematischen Funktionen sind verfügbar:

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

- trunc()
- tan()

Fehlermeldungen

Wenn die Argumente eines mathematischen Funktionsaufrufs zu einem Fehler führen, zeigen die Fehlermeldungen die aktuellen Werte der Argumente. Deshalb führen die Statements

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

zur Fehlermeldung

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

Absolutwert-, Maximum- und Minimum-Funktion

Funktion

Absolutwert-, Maximum- und Minimum-Funktion.

Syntax

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

Rückgabewert

`abs` liefert den absoluten Wert von `x`.
`max` liefert das Maximum von `x` und `y`.
`min` liefert das Minimum von `x` und `y`.

Der Return-Typ dieser Funktionen ist identisch mit dem größeren Typ der Argumente. `type` muss `char`, `int` oder `real` sein.

Beispiel

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

Rundungs-Funktionen

Funktion

Rundungs-Funktionen.

Syntax

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

Rückgabewert

`ceil` liefert den kleinsten Integer-Wert nicht kleiner als `x`.

`floor` liefert den größten Integer-Wert nicht größer als `x`.
`frac` liefert den Dezimalbruch von `x`.
`round` liefert `x` gerundet auf den nächsten Integer-Wert.
`trunc` liefert den ganzzahligen Teil von `x`.

Beispiel

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

Trigonometrische Funktionen

Funktion

Trigonometrische Funktionen.

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);
```

Rückgabewert

`acos` liefert den arc-cosinus von `x`.
`asin` liefert den arc-sinus von `x`.
`atan` liefert den arc-tangens von `x`.
`cos` liefert den cosinus von `x`.
`sin` liefert den sinus von `x`.
`tan` liefert den tangens von `x`.

Konstanten

`PI` der Wert von "pi" (3.14...)

Beispiel

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

Exponential-Funktionen

Funktion

Exponential-Funktionen.

Syntax

```
real exp(real x);
real log(real x);
real log10(real x);
```

```
real pow(real x, real y);
real sqrt(real x);
```

Rückgabewert

`exp` liefert *e* hoch `x`.
`log` liefert den natürlichen Logarithmus von `x`.
`log10` liefert den Zehnerlogarithmus von `x`.
`pow` liefert den Wert von `x` hoch `y`.
`sqrt` liefert die Quadratwurzel von `x`.

Anmerkung

Die "n-te" Wurzel kann mit Hilfe der `pow`-Funktion und einem negativen Exponenten berechnet werden.

Beispiel

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

Sonstige Funktionen

Sonstige Funktionen werden für weitere Aufgaben benötigt.

Die folgenden sonstigen Funktionen sind verfügbar:

- exit()
- language()
- lookup()
- palette()
- sort()
- status()
- system()
- Einheiten-Konvertierung

exit()

Funktion

Beendet ein User-Language-Programm.

Syntax

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

Siehe auch RUN

Die `exit`-Funktion beendet die Ausführung des User-Language-Programms.
Wird `result` (integer) angegeben, wird es als Rückgabewert des Programms benutzt.
Wird ein `command`-String angegeben, wird dieser Befehl genauso ausgeführt, als wäre über die Kommandozeile direkt nach dem RUN-Befehl eingegeben worden. In diesem Fall wird

der Rückgabewert des ULPs auf EXIT_SUCCESS gesetzt.

Konstanten

EXIT_SUCCESS Rückgabewert für erfolgreiche Programmausführung (Wert 0)

EXIT_FAILURE Rückgabewert für fehlerhafte Programmausführung (Wert -1)

language()

Funktion

Liefert den Sprachcode des verwendeten Systems.

Syntax

```
string language();
```

Rückgabewert

language liefert einen String bestehend aus zwei Kleinbuchstaben, der die auf dem aktuellen System verwendete Sprache angibt. Falls sich diese Einstellung nicht ermitteln lässt, wird ein leerer String zurückgegeben.

Die language-Funktion kann dazu benutzt werden, in einem ULP unterschiedliche Texte zu verwenden, je nachdem welche Sprache das aktuelle System verwendet.

In dem folgenden Beispiel sind alle im ULP verwendeten Strings im Array I18N[] aufgelistet, beginnend mit einem String der die verschiedenen Sprachcodes enthält die dieses ULP unterstützt. Beachten Sie die vtab-Zeichen, die dazu benutzt werden, die einzelnen Teile jedes Strings zu trennen (diese sind wichtig für die lookup-Funktion) und die Benutzung der Kommas um die Strings zu trennen. Die eigentliche Arbeit wird in der Funktion tr() gemacht, welche die übersetzte Version des übergebenen Strings zurückliefert. Sollte der ursprüngliche String im I18N-Array nicht gefunden werden, oder es keine Übersetzung für die aktuelle Sprache geben, so wird der ursprüngliche String unübersetzt verwendet.

Die erste im I18N-Array definierte Sprache muss diejenige sein, in der die im restlichen ULP verwendeten Strings geschrieben sind, und sollte generell Englisch sein um das Programm einer möglichst großen Zahl von Benutzern zugänglich zu machen.

Beispiel

```
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()

Funktion

Sucht Daten in einem 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]);
```

Rückgabewert

`lookup` liefert den Wert des Feldes, das durch `field_index` oder `field_name` markiert wird.
Existiert dieses Feld nicht oder wird kein passender String für `key` gefunden, kommt ein leerer String zurück.

Siehe auch fileread, strsplit

Ein `array` das mit `lookup()` benutzt werden kann, besteht aus Text-Strings, wobei jeder String einen Daten-Record darstellt.

Jeder Daten-Record enthält eine beliebige Anzahl von Feldern, die durch das Zeichen `separator` (default ist `'\t'`, der Tabulator) getrennt sind. Das erste Feld in einem Record wird als `key` benutzt und hat die Nummer `0`.

Alle Records müssen eindeutige `key`-Felder haben. Keines der `key`-Felder darf leer sein - ansonsten ist nicht definiert welcher Record gefunden wird.

Enthält der erste String in `array` einen "Header"-Record (der Record, in dem der Inhalt der Felder beschrieben wird), bestimmt `lookup` mit einem String `field_name` automatisch

den Index des Feldes. Das erlaubt es, die `lookup`-Funktion zu benutzen, ohne genau zu wissen, welcher Feld-Index die gewünschten Daten enthält.
Es bleibt dem Benutzer überlassen, sicherzustellen, dass der erste Record tatsächlich Header-Informationen enthält.

Ist der `key`-Parameter beim Aufruf von `lookup()` ein leerer String, wird der erste String von `array` verwendet. Das erlaubt dem Programm zu bestimmen, ob ein Header-Record mit den gewünschten Feld-Namen existiert.

Enthält ein Feld das `separator`-Zeichen, muss es in Anführungszeichen eingeschlossen werden (wie in `"abc;def"`, wobei hier das Semikolon (`';'`) das Trennzeichen ist). Das gilt auch für Felder, die Anführungszeichen (`"`) enthalten, wobei die Anführungszeichen im Feld verdoppelt werden müssen (wie hier: `"abc;""def"";ghi"` ergibt also `abc;"def";ghi`).
Es wird empfohlen den "tab"-Separator (default) zu verwenden, der diese Probleme nicht kennt (kein Feld kann einen Tabulator enthalten).

Hier folgt eine Beispiel-Daten-Datei (zur besseren Lesbarkeit wurde der Separator `';'` verwendet):

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

Beispiel

```
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()

Funktion

Liefert Farbpaletten-Information.

Syntax

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

Rückgabewert

Die `palette`-Funktion liefert einen ARGB-Wert als Integer-Zahl der Form 0xaarrggbb, oder den Typ der momentan verwendeten Palette (abhängig vom Wert von `index`).

Die `palette`-Funktion liefert den ARGB-Wert der Farbe mit dem gegebenen `index` (welcher im Bereich 0..PALETTE_ENTRIES-1 liegen kann). Falls `type` nicht angegeben ist (oder den Wert `-1` hat) wird die Palette verwendet, die dem aktuellen Editor-Fenster zugewiesen ist. Ansonsten gibt `type` an, welche Palette verwendet werden soll (PALETTE_BLACK, PALETTE_WHITE oder PALETTE_COLORED).

Der spezielle Wert `-1` für `index` bewirkt, dass die Funktion den Typ der momentan vom Editor-Fenster verwendeten Palette liefert.

Falls `index` oder `type` ausserhalb des gültigen Wertebereichs liegen wird eine Fehlermeldung ausgegeben und das ULP abgebrochen.

Konstanten

`PALETTE_TYPES`	die Anzahl der Palette-Typen (3)
`PALETTE_BLACK`	die Palette für schwarzen Hintergrund (0)
`PALETTE_WHITE`	die Palette für weißen Hintergrund (1)
`PALETTE_COLORED`	die Palette für farbigen Hintergrund (2)
`PALETTE_ENTRIES`	die Anzahl der Farben pro Palette (64)

sort()

Funktion

Sortiert ein Array oder einen Satz von Arrays.

Syntax

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

Die `sort`-Funktion sortiert entweder direkt ein `array1`, oder sie sortiert einen Satz von Arrays (beginnend mit `array2`), wobei `array1` ein **int**-Array ist, das als Pointer-Array verwendet wird.

In jedem Fall definiert das Argument `number` die Zahl der Einträge im Array oder in den Arrays.

Einzelnes Array sortieren

Wenn die `sort`-Funktion mit einem einzelnen Arrray aufgerufen wird, wird dieses Array direkt sortiert, wie im folgenden Beispiel:

```
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]);
```

Einen Satz von Arrays sortieren

Wenn die `sort`-Funktion mit mehr als einem Array aufgerufen wird, muss das erste Array ein **int**-Array sein, während alle anderen Arrays von jedem Typ sein können. Sie enthalten die zu sortierenden Daten. Das folgende Beispiel zeigt, wie das erste Array als Pointer verwendet wird:

```
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]]);
  }
```

Die Idee dahinter ist, dass an ein Netz mehrere Pins angeschlossen sein können. In einer Netzliste wollen Sie unter Umständen die Netznamen sortieren und innerhalb eines Netzes die Bauteilnamen, und so weiter.

Beachten Sie die Verwendung des Schlüsselworts `numeric` in den String-Arrays. Das führt dazu, dass die String-Sortierung einen numerischen Teil am Ende des Namens berücksichtigt (IC1, IC2,... IC9, IC10 anstelle von IC1, IC10, IC2,...IC9).

Wenn man einen Satz von Arrays sortiert, muss das erste (Index-)Array vom Typ `int` sein und braucht nicht initialisiert zu werden. Jeder vor dem Aufruf der `sort`-Funktion vorhandene Inhalt wird mit den resultierenden Indexwerten überschrieben.

status()

Funktion

Zeigt eine Statusmeldung in der Statuszeile an.

Syntax

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

Siehe auch dlgMessageBox()

Die `status`-Funktion zeigt die angegebene `message` in der Statuszeile des Editor-Fensters an, in dem das ULP läuft.

system()

Funktion

Führt ein externes Programm aus.

Syntax

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

Rückgabewert

Die `system`-Funktion liefert den "Exit Status" des Befehls zurück. Dieser ist normalerweise `0` wenn alles gut gegangen ist, und ungleich `0` wenn ein Fehler aufgetreten ist.

Die `system`-Funktion führt das im `command` angegebene externe Programm aus und wartet, bis dieses beendet ist.

Zur Sicherheit werden Sie vor der Ausführung des Befehls gefragt, ob Sie dieses zulassen möchten, damit nicht ein "böses" ULP unerwünschte externe Befehle ausführen kann. Wird dieser Dialog abgebrochen, so liefert der `system()` Aufruf `-1` zurück. Wird der Dialog bestätigt, so werden alle künftigen `system()` Aufrufe in der laufenden EAGLE-Sitzung mit genau der gleichen Befehlszeile ohne weiteren Bestätigungsdialog ausgeführt.

Ein-/Ausgabe-Umleitung

Soll ein externes Programm seine Eingaben von einer bestimmten Datei lesen (bzw. seine Ausgaben in eine bestimmte Datei schreiben), so muß die Ein- bzw. Ausgabe umgeleitet werden.

Unter **Linux** und **Mac OS X** geschieht dies einfach durch Anhängen von `'<'` bzw. `'>'` an die Befehlszeile, jeweils gefolgt von der gewünschten Datei, wie in

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

womit `program` gestartet wird und es von `infile` liest und nach `outfile` schreibt.

Unter **Windows** muß explizit ein Kommando-Prozessor gestartet werden um dies zu ermöglichen, wie in

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

(auf DOS-basierten Windows-Systemen verwenden Sie `command.com` anstatt `cmd.exe`).

Ausführung im Hintergrund

Die `system`-Funktion wartet bis das angegebene Programm beendet ist. Dies ist sinnvoll für Programme, die nur wenige Sekunden laufen, oder die Aufmerksamkeit des Benutzers komplett an sich ziehen.


Läuft ein externes Programm längere Zeit, und Sie wollen, dass der `system()`-Aufruf sofort zurückkehrt, ohne zu warten, bis das Programm beendet ist, so können Sie unter Linux und Mac OS X einfach ein `'&'` an die Befehlszeile anfügen, wie in

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

Unter Windows muß explizit ein Kommando-Prozessor gestartet werden um dies zu ermöglichen, wie in

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

(auf DOS-basierten Windows-Systemen verwenden Sie `command.com` anstatt `cmd.exe`).

Beispiel

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

Hiermit würde ein Simulationsprogramm aufgerufen und diesem eine Datei übergeben werden, die das ULP gerade erzeugt hat. Beachten Sie bitte, dass `simulate` hier nur ein Beispiel und nicht Bestandteil des EAGLE-Paketes ist!

Einheiten-Konvertierung

Funktion

Konvertiert interne Einheiten.

Syntax

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

Rückgabewert

`u2inch` liefert den Wert von `n` in *Inch*.
`u2mic` liefert den Wert von `n` in *Micron* (1/1000mm).
`u2mil` liefert den Wert von `n` in *Mil* (1/1000inch).
`u2mm` liefert den Wert von `n` in *Millimeter*.

Siehe auch UL_GRID

EAGLE speichert alle Koordinaten und Größen als `int`-Werte mit einer Auflösung von 1/10000mm (0.1μ). Die oben angegebenen Einheiten-Konvertier-Funktionen können dazu verwendet werden, die internen Einheiten in die gewünschten Maßeinheiten umzuwandeln.

Beispiel

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

Print-Funktionen

Print-Funktionen dienen zur Ausgabe formatierter Strings.

Die folgenden Print-Funktionen sind verfügbar:

- printf()
- sprintf()

printf()

Funktion

Schreibt formatiere Ausgaben in eine Datei.

Syntax

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

Rückgabewert

Die `printf`-Funktion liefert die Zahl der Zeichen, die in die vom letzten output-Statement geöffnete Datei geschrieben wurden.

Wenn ein Fehler auftritt, liefert `printf -1`.

Siehe auch sprintf, output, fileerror

Format-String

Der Format-String steuert, wie die Argumente konvertiert, formatiert und ausgegeben werden. Es müssen genau so viele Argumente vorhanden sein, wie für das Format erforderlich sind. Die Zahl und der Typ der Argumente werden für jedes Format geprüft, und wenn sie nicht den Anforderungen entsprechen, wird eine Fehlermeldung ausgegeben.

Der Format-String enthält zwei Objekt-Typen - *einfache Zeichen* und *Format-Specifier*:

- Einfache Zeichen werden direkt ausgegeben
- Format-Specifier holen Argumente von der Argument-Liste und formatieren sie

Format-Specifier

Ein Format-Specifier hat folgende Form:

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

Jede Format-Spezifizierung beginnt mit dem Prozentzeichen (`%`). Nach dem `%` kommt folgendes, in dieser Reihenfolge:

- optional eine Folge von Flag-Zeichen, `[flags]`
- optional ein Breiten-Specifier, `[width]`
- optional ein Präzisions-Specifier, `[.prec]`
- das Konvertiertyp-Zeichen, `type`

Konvertiertyp-Zeichen

`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 von der Form `[-]dddd.dddd`

`e` **signed real** value von der Form `[-]d.ddde[±]ddd`

`E` wie `e`, aber mit **E** für Exponent

`g` **signed real** value entweder wie `e` oder `f`, abhängig vom gegebenen Wert und Präzision

`G` wie `g`, aber mit **E** für Exponent, wenn `e`-Format verwendet wird

`c` einzelnes Zeichen

`s` Character-String

`%` das `%`-Zeichen wird ausgegeben

Flag-Zeichen

Die folgenden Flag-Zeichen können in jeder Kombination und Reihenfolge auftreten.

`"-` das formatierte Argument wird innerhalb des Feldes linksbündig ausgegeben;
`"` normalerweise ist die Ausgabe rechtsbündig

`"+` ein positiver Wert mit Vorzeichen wird mit Pluszeichen (`+`) ausgegeben;
`"` normalerweise werden nur negative Werte mit Vorzeichen ausgegeben

`"` ein positiver Wert mit Vorzeichen wird mit Leerzeichen am Anfang ausgegeben; wenn
`"` `"+"` und `" "` angegeben sind, überschreibt `"+"` die Angabe `" "`

Width-Specifier

Der Width-Specifier setzt die minimale Feldbreite für einen Ausgabewert.

Die Breite wird entweder direkt mit einem Dezimalstellen-String oder indirekt mit einem Stern (`*`) gesetzt. Wenn Sie `*` verwenden, legt das nächste Argument im Aufruf (das vom Typ `int` sein muss) die minimale Feldbreite fest.

Auf keinen Fall führt ein nicht existierendes oder zu ein kleines Feld dazu, dass ein Wert abgeschnitten wird. Wenn das Ergebnis der Konvertierung breiter ist als das Feld, wird das Feld einfach so vergrößert, dass das Ergebnis platz hat.

n	Mindestens *n* Zeichen werden ausgegeben. Wenn der Ausgabewert weniger als *n* Zeichen hat, wird er mit Leerzeichen aufgefüllt (rechts wenn das `"-"`-Flag gesetzt ist, sonst links).
`0`*n*	Mindestens *n* Zeichen werden ausgegeben. Wenn der Ausgabewert weniger als *n* Zeichen hat, wird links mit Nullen aufgefüllt.
`*`	Die Argument-Liste liefert den Width-Specifier, der dem eigentlichen (zu formatierenden) Argument vorausgehen muss.

Präzisions-Specifier

Ein Präzisions-Specifier beginnt immer mit einem Punkt (`.`), um ihn von einem vorangehenden Width-Specifier zu trennen. Dann wird, wie bei "Width", die Präzision entweder direkt mit einem Dezimalstellen-String oder indirekt mit einem Stern (`*`) angegeben. Wenn Sie `*` verwenden, legt das nächste Argument im Aufruf (das vom Typ `int` sein muss) die Präzision fest.

keiner	Präzision auf Standardwert gesetzt.
`.0`	Für `int`-Typen, Präzision wird auf Default gesetzt; für `real`-Typen, kein Dezimalpunkt wird ausgegeben.
`.`*n*	*n* Zeichen oder *n* Dezimalstellen werden ausgegeben. Wenn der Ausgabewert mehr als *n* Zeichen hat, kann er abgeschnitten oder gerundet werden (abhängig vom Typ-Zeichen).
`*`	Die Argument-Liste liefert den Präzisions-Specifier, der dem eigentlichen (zu formatierenden) Argument vorausgehen muss.

Default-Präzisionswerte

`douxX`	1
`eEf`	6

`gG`	alle signifikanten Stellen
`c`	keine Auswirkung
`s`	gesamten String ausgeben

Wie die Präzisionsangabe (`.n`) die Konvertierung beeinflusst

`douxX`	*.n* spezifiziert dass mindestens *n* Zeichen ausgegeben werden. Wenn das Eingangs-Argument weniger als *n* Stellen hat, wird der Ausgangswert links mit Nullen aufgefüllt. Wenn das Eingangs-Argument mehr als *n* Stellen hat, wir die Ausgabe **nicht** abgeschnitten.
`eEf`	*.n* spezifiziert dass *n* Zeichen nach dem Dezimalpunkt ausgegeben werden, und die letzte ausgegebene Stelle wird gerundet.
`gG`	*.n* spezifiziert dass höchstens *n* signifikante Stellen ausgegeben werden.
`c`	*.n* hat keinen Einfluss auf die Ausgabe.
`s`	*.n* spezifiziert dass nicht mehr als *n* Zeichen gedruckt werden.

Der binäre Wert 0

Im Gegensatz zu sprintf kann die `printf`-Funktion den binären Wert 0 (0x00) ausgeben.

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

Beispiel

```
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()

Funktion

Schreibt eine formatierte Ausgabe in einen String.

Syntax

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

Rückgabewert

Die `sprintf`-Funktion liefert die Zahl der Zeichen, die in den `result`-String geschrieben wurden.

Im Falle eines Fehlers liefert `sprintf` den Wert `-1`.

Siehe auch printf

Format-String

Siehe printf.

Der binäre Wert 0

Bitte beachten Sie, dass `sprintf` den binären Wert 0 (0x00) nicht verarbeiten kann. Wenn der Ergebnis-String 0x00 enthält, werden die folgenden Zeichen ignoriert. Verwenden Sie printf um binäre Daten auszugeben.

Beispiel

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

String-Funktionen

String-Funktionen werden dazu verwendet, Character-Strings zu manipulieren.

Die folgenden String-Funktionen sind verfügbar:

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

strchr()

Funktion

Durchsucht einen String nach dem ersten Vorkommen eines gegebenen Zeichens.

Syntax

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

Rückgabewert

Die `strchr`-Funktion liefert den Integer-Offset des Zeichen im String oder `-1`, wenn das Zeichen nicht vorkommt.

Siehe auch strrchr, strstr

Falls `index` angegeben wird, beginnt die Suche an dieser Position. Negative Werte werden vom Ende des Strings her gezählt.

Beispiel

```
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()

Funktion

Erzeugt aus einem String-Array einen einzelnen String.

Syntax

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

Rückgabewert

Die `strjoin`-Funktion liefert die kombinierten Einträge von `array`.

Siehe auch strsplit, lookup, fileread

`strjoin` fügt alle Einträge aus `array`, getrennt durch den angegebenen `separator` zusammen, und liefert den Ergebnis-String.

Wenn `separator` ein Newline-Zeichen (`"\n"`) ist, wird der Ergebnis-String mit einem Newline-Zeichen abgeschlossen. So erhält man eine Textdatei mit N Zeilen (jede davon ist mit einem Newline-Zeichen abgeschlossen). Die Datei wird mit den Funktionen fileread() eingelesen und mit split in ein Array mit N Strings aufgeteilt und zu dem ursprünglichen String, der aus der Datei eingelesen wurde, hinzugefügt.

Beispiel

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

strlen()

Funktion

Berechnet die Länge eines Strings.

Syntax

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

Rückgabewert

Die `strlen`-Funktion liefert die Zahl der Zeichen im String.

Beispiel

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

strlwr()

Funktion

Wandelt Großbuchstaben in einem String in Kleinbuchstaben um.

Syntax

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

Rückgabewert

Die `strlwr`-Funktion liefert den modifizierten String. Der Original-String (als Parameter übergeben) wird nicht geändert.

Siehe auch strupr, tolower

Beispiel

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

strrchr()

Funktion

Durchsucht einen String nach dem letzten Vorkommen eines gegebenen Zeichens.

Syntax

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

Rückgabewert

Die `strrchr`-Funktion liefert den Integer-Offset des Zeichens im String oder `-1`, wenn das Zeichen nicht vorkommt.

Siehe auch strchr, strrstr

Falls `index` angegeben wird, beginnt die Suche an dieser Position. Negative Werte werden vom Ende des Strings her gezählt.

Beispiel

```
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()

Funktion

Durchsucht einen String nach dem letzten Vorkommen eines gegebenen Substrings.

Syntax

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

Rückgabewert

Die `strrstr`-Funktion liefert den Integer-Offset des ersten Zeichens von s2 in s1, oder `-1`, wenn der Substring nicht vorkommt.

Siehe auch strstr, strrchr

Falls `index` angegeben wird, beginnt die Suche an dieser Position. Negative Werte werden vom Ende des Strings her gezählt.

Beispiel

```
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()

Funktion

Teilt einen String in einzelne Felder.

Syntax

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

Rückgabewert

Die `strsplit`-Funktion liefert die Anzahl der Einträge die nach `array` kopiert wurden.

Siehe auch strjoin, lookup, fileread

`strsplit` teilt den String `s` am angegebenen `separator` und speichert die so erzeugten Felder in `array`.

Wenn `separator` ein Newline-Zeichen ist (`"\n"`), wird das letzte Feld einfach ignoriert, sofern es leer ist. So erhält man eine Textdatei, die aus N Zeilen besteht (jede durch Newline beendet). Diese wird durch die Funktion fileread() eingelesen und in ein Array von N Strings aufgeteilt. Mit jedem anderen `separator` ist ein leeres Feld am Ende des Strings gültig. So entstehen aus `"a:b:c:"` 4 Felder, das letzte davon ist leer.

Beispiel

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

strstr()

Funktion

Durchsucht einen String nach dem ersten Vorkommen eines gegebenen Substrings.

Syntax

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

Rückgabewert

Die `strstr`-Funktion liefert den Integer-Offset des ersten Zeichens von s2 in s1, oder `-1`, wenn der Substring nicht vorkommt.

Siehe auch strrstr, strchr

Falls `index` angegeben wird, beginnt die Suche an dieser Position. Negative Werte werden vom Ende des Strings her gezählt.

Beispiel

```
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()

Funktion

Extrahiert einen Substring aus einem String.

Syntax

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

Rückgabewert

Die `strsub`-Funktion liefert den Substring, der durch `start` und `length` definiert ist.

Der Wert für `length` muss positiv sein, andernfalls wird ein leerer String zurückgegeben. Wenn `length` nicht angegeben ist, wird der Reststring (beginnend bei `start`) zurückgegeben.

Wenn `start` auf eine Position ausserhalb des Strings deutet, wird ein leerer String zurückgegeben.

Beispiel

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

strtod()

Funktion

Konvertiert einen String in einen Real-Wert.

Syntax

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

Rückgabewert

Die `strtod`-Funktion liefert die numerische Repräsentation eines gegebenen Strings als `real`-Wert. Die Konvertierung wird beim ersten Zeichen beendet, das nicht dem Format einer Real-Konstanten entspricht. Wenn ein Fehler während er Konvertierung auftaucht, wir der Wert `0.0` zurückgegeben.

Siehe auch strtol

Beispiel

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

strtol()

Funktion

Konvertiert einen String in einen Integer-Wert.

Syntax

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

Rückgabewert

Die `strtol`-Funktion liefert die numerische Representation eines gegebenen Strings als `int`-Wert. Die Konvertierung wird beim ersten Zeichen beendet, das nicht dem Format einer Integer-Konstanten entspricht. Wenn ein Fehler während der Konvertierung auftaucht, wir der Wert `0` zurückgegeben.

Siehe auch strtod

Beispiel

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

strupr()

Funktion

Konvertiert Kleinbuchstaben in einem String in Großbuchstaben.

Syntax

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

Rückgabewert

Die `strupr`-Funktion liefert den modifizierten String. Der Original-String (als Parameter übergeben) wird nicht geändert.

Siehe auch strlwr, toupper

Beispiel

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

Zeit-Funktionen

Zeit-Funktionen werden dazu verwendet, die Zeit- und Datums- Informationen zu erhalten und weiterzuverarbeiten.

Die folgenden Zeit-Funktionen sind verfügbar:

- t2day()
- t2dayofweek()
- t2hour()
- t2minute()
- t2month()
- t2second()
- t2string()
- t2year()
- time()
- timems()

time()

Funktion

Holt die gegenwärtige Systemzeit.

Syntax

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

Rückgabewert

Die `time`-Funktion liefert die gegenwärtige Systemzeit als Zahl von Sekunden, die seit einem systemabhängigen Referenzzeitpunkt vergangen sind.

Siehe auch Zeit-Konvertierungen, filetime

Beispiel

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

timems()

Funktion

Liefert die Zeit in Millisekunden seit dem Start des ULPs.

Syntax

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

Rückgabewert

Die `timems`-Funktion liefert die Zeit in Millisekunden seit dem Start des ULPs.

Nach 86400000 Millisekunden (d.h. alle 24 Stunden) beginnt der Wert wieder bei 0.

Siehe auch Zeit-Konvertierungen, filetime, timems()

Beispiel

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

Zeit-Konvertierungen

Funktion

Zeit-Wert in Tag, Monat, Jahr etc. konvertieren.

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);
```

Rückgabewert

`t2day` liefert den Tag des Monats (`1..31`)
`t2dayofweek` liefert den Tag der Woche (`0=sunday..6`)
`t2hour` liefert die Stunde (`0..23`)
`t2minute` liefert die Minute (`0..59`)
`t2month` liefert den Monat (`0..11`)
`t2second` liefert die Sekunde (`0..59`)
`t2year` liefert das Jahr (einschließlich Jahrhundert!)
`t2string` liefert einen formatierten String, der Datum und Zeit enthält

Siehe auch time

Beispiel

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

Objekt-Funktionen

Objekt-Funktionen werden dazu verwendet, allgemeine Informationen von Objekten zu erfragen.

Die folgenden Objekt-Funktionen sind verfügbar:

- ingroup()

ingroup()

Funktion

Prüft ob ein Objekt in der Gruppe liegt.

Syntax

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

Rückgabewert

Die `ingroup`-Funktion liefert einen Wert ungleich 0 wenn das gegebene Objekt in der Gruppe liegt.

Siehe auch GROUP-Befehl

Wurde im Editor eine Gruppe definiert, so kann die `ingroup()`-Funktion benutzt werden um zu prüfen, ob ein bestimmtes Objekt Bestandteil der Gruppe ist.

Objekte mit einem einzelnen Aufhängepunkt, die in der aktuellen Zeichnung gezielt selektiert werden können (wie etwa UL_TEXT, UL_VIA, UL_CIRCLE etc.), liefern beim Aufruf von `ingroup()` einen Wert ungleich 0 wenn dieser Aufhängepunkt innerhalb der Gruppe liegt.

Ein UL_WIRE liefert 0, 1, 2 oder 3, je nachdem, ob keiner, der erste, der zweite oder beide Endpunkte in der Gruppe liegen.

Ein UL_RECTANGLE bzw. UL_FRAME liefert einen Wert ungleich 0 wenn einer oder mehrere seiner Eckpunkte in der Gruppe liegen.

Objekte ohne Aufhängepunkt (wie etwa UL_NET, UL_SEGMENT, UL_SIGNAL etc.) liefern einen Wert ungleich 0 wenn eines oder mehrere der Objekte, die sie enthalten, in der Gruppe liegen.

UL_CONTACTREF und UL_PINREF haben zwar selber keinen Aufhängepunkt, liefern aber einen Wert ungleich 0 wenn der referenzierte UL_CONTACT bzw. UL_PIN innerhalb der Gruppe liegt.

Beispiel

```
output("group.txt") {
  board(B) {
```

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

Builtin-Statements

Builtin-Statements werden im allgemeinen dazu verwendet, einen Kontext zu eröffnen, der den Zugriff auf Datenstrukturen und Dateien erlaubt.

Die allgemeine Syntax von Builtin-Statements ist

```
name(parameters) statement
```

wobei `name` der Name des Builtin-Statement ist, `parameters` steht für einen oder mehrere Parameter, und `statement` ist der Code, der innerhalb des vom Builtin-Statement geöffneten Kontexts ausgeführt wird.

Beachten Sie, dass es sich bei `statement` um eine Compound-Statement handeln kann, wie in

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

Die folgenden Builtin-Statements sind verfügbar:

- board()
- deviceset()
- library()
- output()
- package()
- schematic()
- sheet()
- symbol()

board()

Funktion

Öffnet einen Board-Kontext.

Syntax

```
board(identifier) statement
```

Siehe auch schematic, library

Das `board`-Statement öffnet einen Board-Kontext wenn das gegenwärtige Editor-Fenster ein Board enthält. Eine Variable vom Typ UL_BOARD wird angelegt und erhält den Namen, den `identifier` angibt.

Sobald der Board-Kontext erfolgreich geöffnet wurde und eine Board-Variable angelegt ist,

wird `statement` ausgeführt. Innerhalb des Gültigkeitsbereichs von `statement` kann man auf die Board-Variable zugreifen, um weitere Daten aus dem Board zu erhalten.

Wenn das gegenwärtige Editor-Fenster kein Board enthält, wird eine Fehlermeldung ausgegeben, und das ULP wird beendet.

Prüfen, ob ein Board geladen ist

Mit dem `board`-Statement ohne Angabe eines Arguments können Sie prüfen, ob das gegenwärtige Editor-Fenster ein Board enthält. In diesem Fall verhält sich `board` wie eine Integer-Konstante, die den Wert `1` zurückgibt, sofern ein Board geladen ist. Andernfalls wird der Wert `0` zurückgegeben.

Zugriff auf ein Board von einem Schaltplan aus

Wenn das gegenwärtige Editor-Fenster einen Schaltplan enthält, können Sie trotzdem auf das zugehörige Board zugreifen, indem Sie dem `board`-Statement den Präfix `project` voranstellen, wie in

```
project.board(B) { ... }
```

Das öffnet einen Board-Kontext, unabhängig davon, ob das gegenwärtige Editor-Fenster ein Board oder einen Schaltplan enthält. Allerdings muss es auf dem Desktop ein Fenster geben, das dieses Board enthält!

Beispiel

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

deviceset()

Funktion

Öffnet einen Device-Set-Kontext.

Syntax

```
deviceset(identifier) statement
```

Siehe auch package, symbol, library

Das `deviceset`-Statement öffnet einen Device-Set-Kontext wenn das gegenwärtige Editor-Fenster ein Device-Set enthält. Eine Variable vom Typ UL_DEVICESET wird angelegt und erhält den Namen, den `identifier` angibt.

Sobald der Device-Set-Kontext erfolgreich geöffnet wurde und eine Device-Set-Variable angelegt ist, wird `statement` ausgeführt. Innerhalb des Gültigkeitsbereichs von `statement` kann man auf die Device-Set-Variable zugreifen, um weitere Daten aus dem Device-Set zu erhalten.

Wenn das gegenwärtige Editor-Fenster kein Device-Set enthält, wird eine Fehlermeldung ausgegeben, und das ULP wird beendet.

Prüfen, ob ein Device-Set geladen ist

Mit dem `deviceset`-Statement ohne Angabe eines Arguments können Sie prüfen, ob das gegenwärtige Editor-Fenster ein Device-Set enthält. In diesem Fall verhält sich `deviceset` wie eine Integer-Konstante, die den Wert `1` zurückgibt, sofern ein Device-Set geladen ist. Andernfalls wird der Wert `0` zurückgegeben.

Beispiel

```
if (deviceset)
   deviceset(D) {
     D.gates(G)
       printf("Gate: %s\n", G.name);
     }
```

library()

Funktion

Öffnet einen Library-Kontext.

Syntax

```
library(identifier) statement
```

Siehe auch board, schematic, deviceset, package, symbol

Das `library`-Statement öffnet einen Library-Kontext wenn das gegenwärtige Editor-Fenster eine Library enthält. Eine Variable vom Typ UL_LIBRARY wird angelegt und erhält den Namen, den `identifier` angibt.

Sobald der Library-Kontext erfolgreich geöffnet wurde und eine Board-Variable angelegt ist, wird `statement` ausgeführt. Innerhalb des Gültigkeitsbereichs von `statement` kann man auf die Library-Variable zugreifen, um weitere Daten aus der Bibliothek zu erhalten.

Wenn das gegenwärtige Editor-Fenster keine Bibliothek enthält, wird eine Fehlermeldung ausgegeben, und das ULP wird beendet.

Prüfen, ob eine Bibliothek geladen ist

Mit dem `library`-Statement ohne Angabe eines Arguments können Sie prüfen, ob das gegenwärtige Editor-Fenster eine Bibliothek enthält. In diesem Fall verhält sich `library` wie eine Integer-Konstante, die den Wert `1` zurückgibt, sofern eine Bibliothek geladen ist. Andernfalls wird der Wert `0` zurückgegeben.

Beispiel

```
if (library)
   library(L) {
     L.devices(D)
       printf("Device: %s\n", D.name);
```

```
}
```

output()

Funktion

Öffnet eine Ausgabe-Datei für nachfolgende printf()-Aufrufe.

Syntax

```
output(string filename[, string mode]) statement
```

Siehe auch printf, fileerror

Das `output`-Statement öffnet eine Datei mit dem Namen `filename` und dem Parameter `mode` für die Ausgabe mit nachfolgenden printf()-Aufrufen. Sobald die Datei erfolgreich geöffnet wurde, wird `statement` ausgeführt, und danach wird die Datei geschlossen.

Wenn die Datei nicht geöffnet werden kann, wird eine Fehlermeldung ausgegeben, und das ULP wird beendet.

Standardmäßig wird die erzeugte Datei in das **Projekt** Verzeichnis geschrieben.

Datei-Modi

Der `mode`-Parameter definiert, wie die Datei geöffnet werden soll. Wenn kein `mode`-Parameter angegeben ist, gilt der Standardwert `"wt"`.

`a`	an existierende Datei anhängen oder neue Datei anlegen, falls die Datei nicht existiert
`w`	neue Datei anlegen (existierende überschreiben)
`t`	Datei im Textmodus öffnen
`b`	Datei im Binärmodus öffnen
`D`	Datei am Ende der EAGLE-Sitzung löschen (funktioniert nur zusammen mit `w`)
`F`	diesen Dateinamen erzwingen (normalerweise werden *.brd, *.sch und *.lbr abgewiesen)

Mode-Parameter können in beliebiger Kombination und Reihenfolge angegeben werden. Allerdings ist nur der letzte aus `a` und `w` bzw. `t` und `b` signifikant. Die Angabe `"abtw"` würde zum Beispiel eine Text-Datei öffnen (entsprechend `"wt"`).

Verschachtelte Output-Statements

`output`-Statements können verschachtelt werden, solange genügend Datei-Handles verfügbar sind - vorausgesetzt, es greifen nicht mehrere aktive `output`-Statements auf dieselbe Datei zu.

Beispiel

```
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()

Funktion

Öffnet einen Package-Kontext.

Syntax

```
package(identifier) statement
```

Siehe auch library, deviceset, symbol

Das `package`-Statement öffnet einen Package-Kontext wenn das gegenwärtige Editor-Fenster ein Package enthält. Eine Variable vom Typ UL_PACKAGE wird angelegt und erhält den Namen, den `identifier` angibt.

Sobald der Package-Kontext erfolgreich geöffnet wurde und eine Package-Variable angelegt ist, wird `statement` ausgeführt. Innerhalb des Gültigkeitsbereichs von `statement` kann man auf die Package-Variable zugreifen, um weitere Daten aus dem Package zu erhalten.

Wenn das gegenwärtige Editor-Fenster kein Package enthält, wird eine Fehlermeldung ausgegeben, und das ULP wird beendet.

Prüfen, ob ein Package geladen ist

Mit dem `package`-Statement ohne Angabe eines Arguments können Sie prüfen, ob das gegenwärtige Editor-Fenster ein Package enthält. In diesem Fall verhält sich `package` wie eine Integer-Konstante, die den Wert `1` zurückgibt, sofern ein Package geladen ist. Andernfalls wird der Wert `0` zurückgegeben.

Beispiel

```
if (package)
   package(P) {
     P.contacts(C)
       printf("Contact: %s\n", C.name);
     }
```

schematic()

Funktion

Öffnet einen Schaltplan-Kontext.

Syntax

```
schematic(identifier) statement
```

Siehe auch board, library, sheet

Das `schematic`-Statement öffnet einen Schaltplan-Kontext wenn das gegenwärtige Editor-Fenster einen Schaltplan enthält. Eine Variable vom Typ UL_SCHEMATIC wird angelegt und erhält den Namen, den `identifier` angibt.

Sobald der Schaltplan-Kontext erfolgreich geöffnet wurde und eine UL_SCHEMATIC-Variable angelegt ist, wird `statement` ausgeführt. Innerhalb des Gültigkeitsbereichs von `statement` kann man auf die UL_SCHEMATIC-Variable zugreifen, um weitere Daten aus dem Schaltplan zu erhalten.

Wenn das gegenwärtige Editor-Fenster keinen Schaltplan enthält, wird eine Fehlermeldung ausgegeben, und das ULP wird beendet.

Prüfen, ob ein Schaltplan geladen ist

Mit dem `schematic`-Statement ohne Angabe eines Arguments können Sie prüfen, ob das gegenwärtige Editor-Fenster einen Schaltplan enthält. In diesem Fall verhält sich `schematic` wie eine Integer-Konstante, die den Wert `1` zurückgibt, sofern ein Schaltplan geladen ist. Andernfalls wird der Wert `0` zurückgegeben.

Zugriff auf einen Schaltplan vom Board aus

Wenn das gegenwärtige Editor-Fenster ein Board enthält, können Sie trotzdem auf den zugehörigen Schaltplan zugreifen, indem Sie dem `schematic`-Statement den Präfix `project` voranstellen, wie in

```
project.schematic(S) { ... }
```

Das öffnet einen UL_SCHEMATIC-Kontext, unabhängig davon, ob das gegenwärtige Editor-Fenster ein Board oder einen Schaltplan enthält. Allerdings muss es auf dem Desktop ein Fenster geben, das diesen Schaltplan enthält!

Zugriff auf die gegenwärtige Seite eines Schaltplans

Verwenden Sie das `sheet`-Statement, um direkt auf die gegenwärtig geladene Schaltplanseite zuzugreifen.

Beispiel

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

sheet()

Funktion

Öffnet einen UL_SHEET-Kontext.

Syntax

```
sheet(identifier) statement
```

Siehe auch schematic

Das `sheet`-Statement öffnet einen UL_SHEET-Kontext, wenn das gegenwärtige Editor-Fenster eine Schaltplanseite enthält. Eine Variable vom Typ UL_SHEET wird angelegt und erhält den Namen, den `identifier` angibt.

Sobald der UL_SHEET-Kontext erfolgreich geöffnet wurde und eine UL_SHEET-Variable angelegt ist, wird `statement` ausgeführt. Innerhalb des Gültigkeitsbereichs von `statement` kann man auf die UL_SHEET-Variable zugreifen, um weitere Daten aus der Seite zu erhalten.

Wenn das gegenwärtige Editor-Fenster keine Schaltplanseite enthält, wird eine Fehlermeldung ausgegeben, und das ULP wird beendet.

Prüfen, ob eine Schaltplanseite geladen ist

Mit dem `sheet`-Statement ohne Angabe eines Arguments können Sie prüfen, ob das gegenwärtige Editor-Fenster eine Schaltplanseite enthält. In diesem Fall verhält sich `sheet` wie eine Integer-Konstante, die den Wert `1` zurückgibt, sofern eine Schaltplanseite geladen ist. Andernfalls wird der Wert `0` zurückgegeben.

Beispiel

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

symbol()

Funktion

Öffnet einen Symbol-Kontext.

Syntax

```
symbol(identifier) statement
```

Siehe auch library, deviceset, package

Das `symbol`-Statement öffnet einen Symbol-Kontext wenn das gegenwärtige Editor-Fenster ein Symbol enthält. Eine Variable vom Typ UL_SYMBOL wird angelegt und erhält den Namen, den `identifier` angibt.

Sobald der Symbol-Kontext erfolgreich geöffnet wurde und eine Symbol-Variable angelegt ist, wird `statement` ausgeführt. Innerhalb des Gültigkeitsbereichs von `statement` kann man auf die Symbol-Variable zugreifen, um weitere Daten aus dem Symbol zu erhalten.

Wenn das gegenwärtige Editor-Fenster kein Symbol enthält, wird eine Fehlermeldung

ausgegeben, und das ULP wird beendet.

Prüfen, ob ein Symbol geladen ist

Mit dem `symbol`-Statement ohne Angabe eines Arguments können Sie prüfen, ob das gegenwärtige Editor-Fenster ein Symbol enthält. In diesem Fall verhält sich `symbol` wie eine Integer-Konstante, die den Wert `1` zurückgibt, sofern ein Symbol geladen ist. Andernfalls wird der Wert `0` zurückgegeben.

Beispiel

```
if (symbol)
   symbol(S) {
     S.pins(P)
       printf("Pin: %s\n", P.name);
     }
```

Dialoge

User-Language-Dialoge ermöglichen es, ein eigenes Frontend für ein User-Language-Progamm zu definieren.

In den folgenden Abschnitten werden die User-Language-Dialoge detailliert beschrieben:

Vordefinierte Dialoge	beschreibt vordefinierte Standard-Dialoge
Dialog-Objekte	beschreibt die Objekte aus denen ein Dialog bestehen kann
Layout-Information	erklärt wie man die Position von Objekten in einem Dialog bestimmt
Dialog-Funktionen	beschreibt besondere Funktionen, die mit Dialogen verwendet werden können
Ein vollständiges Beispiel	zeigt ein vollständiges ULP mit einem Dialog zur Daten-Eingabe

Vordefinierte Dialoge

Vordefinierte Dialoge sind die typischen Dialoge, die häufig zur Dateiauswahl oder bei Fehlermeldungen verwendet werden.

Es gibt folgende vordefinierte Dialoge:

- dlgDirectory()
- dlgFileOpen()
- dlgFileSave()
- dlgMessageBox()

Siehe Dialog-Objekte für Informationen über das Definieren eigener, komplexer Benutzer-Dialoge.

dlgDirectory()

Funktion

Zeigt den Verzeichnis-Dialog.

Syntax

```
string dlgDirectory(string Title[, string Start])
```

Rückgabewert

Die `dlgDirectory`-Funktion liefert den vollen Pfadnamen des gewählten Verzeichnisses.
Hat der Benutzer den Dialog abgebrochen, ist das Resultat ein leerer String.

Siehe auch dlgFileOpen

Die `dlgDirectory`-Funktion zeigt einen Verzeichnis-Dialog in dem der Benutzer ein Verzeichnis selektieren kann.

`Title` zeigt den Titel des Dialogs.

Wenn `Start` nicht leer ist, wird diese Angabe als Startpunkt für `dlgDirectory` verwendet.

Beispiel

```
string dirName;
dirName = dlgDirectory("Select a directory", "");
```

dlgFileOpen(), dlgFileSave()

Funktion

Zeigt einen Datei-Dialog.

Syntax

```
string dlgFileOpen(string Title[, string Start[, string
Filter]])
string dlgFileSave(string Title[, string Start[, string
Filter]])
```

Rückgabewert

Die Funktionen `dlgFileOpen` und `dlgFileSave` liefern die volle Pfadangabe der gewählten Datei.
Bricht der Benutzer den Dialog ab, ist das Ergebnis ein leerer String.

Siehe auch dlgDirectory

Die Funktionen `dlgFileOpen` und `dlgFileSave` zeigen einen Datei-Dialog, aus dem der Benutzer eine Datei selektieren kann.

`Title` wird als Titel des Dialogs verwendet.

Ist `Start` nicht leer, wird diese Angabe als Startpunkt für den Dialog verwendet. Ansonsten wird das aktuelle Verzeichnis verwendet.

Nur Dateien, die der Angabe von `Filter` entsprechen, werden angezeigt. Wird kein

`Filter` angegeben, werden alle Dateien angezeigt.

`Filter` kann entweder ein einfacher Pattern sein (wie in `"*.brd"`), eine Liste von Patterns (wie in `"*.bmp *.jpg"`) oder kann sogar beschreibenden Text enthalten, wie in `"Bitmap-Dateien (*.bmp)"`. Falls die "Dateityp" Combo-Box des Datei-Dialogs mehrere Einträge haben soll, müssen diese durch zwei Semikolons voneinander getrennt werden, wie in `"Bitmap-Dateien (*.bmp);;Andere Bilddateien (*.jpg *.png)"`.

Beispiel

```
string fileName;
fileName = dlgFileOpen("Select a file", "", "*.brd");
```

dlgMessageBox()

Funktion

Zeigt eine Message-Box.

Syntax

`int dlgMessageBox(string Message[, button_list])`

Rückgabewert

Die `dlgMessageBox`-Funktion liefert den Index der Schaltfläche, die der Benutzer selektiert hat.
Die erste Schaltfläche in `button_list` hat den Index `0`.

Siehe auch status()

Die `dlgMessageBox`-Funktion zeigt die angegebene `Message` in einem modalen Dialog-Fenster und wartet darauf, dass der Benutzer eine der Schaltflächen, die über `button_list` definiert wurden, selektiert.

Falls `Message` HTML-Tags enthält, so müssen die Zeichen '<', '>' und '&', damit sie als solche angezeigt werden, als "<", ">" bzw. "&" angegeben werden.

`button_list` ist eine optionale Liste durch Komma getrennter Strings, die einen Satz von Schaltflächen, die unten im Dialog-Fenster angezeigt werden, definiert.
Es können maximal drei Schaltflächen definiert werden. Wird keine `button_list` angegeben, erscheint automatisch `"OK"`.

Die erste Schaltfläche in `button_list` wird die Default-Schaltfläche (sie wird gedrückt, wenn der Benutzer ENTER drückt), und der letzte Eintrag in der Liste wird der "Cancel-Button", der gewählt wird, wenn der Benutzer Esc drückt oder das Dialog-Fenster einfach schließt. Sie können eine andere Schaltfläche als Default-Button definieren, indem Sie den String mit einem `'+'` beginnen. Wollen Sie eine andere Schaltfläche als Cancel-Button definieren, stellen Sie dem String ein `'-'` voran. Um einen Schaltflächen-Text mit einem `'+'` oder `'-'` zu beginnen, muss das Zeichen mit einem Escape-Zeichen markiert werden.

Enthält der Text ein `'&'`, wird das folgende Zeichen zum Hotkey. Wenn der Benutzer die entsprechende Taste drückt, wird diese Schaltfläche gewählt. Um das Zeichen `'&'` im

Schaltflächen-Text zu verwenden, muss es mit einem Escape-Zeichen markiert werden.

Dem Dialog-Fenster kann ein Icon mitgegeben werden, indem das erste Zeichen in `Message` auf

- `';'` - für eine *Information*
- `'!'` - für eine *Warnung*
- `':'` - für einen *Fehler*

gesetzt wird. Soll die `Message` jedoch mit einem dieser Zeichen beginnen, so muss dieses mit einem Escape-Zeichen markiert werden.

Unter **Mac OS X** führt nur das Zeichen `':'` tatsächlich zur Darstellung eines Icons. Alle anderen werden ignoriert.

Beispiel

```
if (dlgMessageBox("Are you sure?", "&Yes", "&No") == 0) {
   // let's do it!
   }
```

Dialog-Objekte

Ein User-Language-Dialog kann aus folgenden *Dialog-Objekten* bestehen (die einzelnen Begriffe wurden in diesem Fall nicht ins Deutsche übersetzt, da sonst der Zusammenhang zu den ULP-Objekten verloren ginge):

dlgCell	ein Grid-Cell-Kontext
dlgCheckBox	eine Checkbox
dlgComboBox	ein Combo-Box-Auswahl-Feld
dlgDialog	die Grundlage eines jeden Dialogs
dlgGridLayout	ein Grid-basierter-Layout-Kontext
dlgGroup	ein Group-Feld
dlgHBoxLayout	ein Horizontal-Box-Layout-Kontext
dlgIntEdit	ein Integer-Eingabe-Feld
dlgLabel	ein Text-Label
dlgListBox	eine List-Box
dlgListView	eine List-View
dlgPushButton	ein Push-Button

dlgRadioButton	ein Radio-Button
dlgRealEdit	ein Real-Eingabe-Feld
dlgSpacing	ein Layout-Spacing-Objekt
dlgSpinBox	ein Spin-Box-Auswahl-Feld
dlgStretch	ein Layout-Stretch-Objekt
dlgStringEdit	ein String-Eingabe-Feld
dlgTabPage	eine Tab-Page
dlgTabWidget	ein Tab-Page-Container
dlgTextEdit	ein Text-Eingabe-Feld
dlgTextView	ein Text-Viewer-Feld
dlgVBoxLayout	ein Vertical-Box-Layout-Kontext

dlgCell

Funktion

Definiert die Position einer Cell (Zelle) in einem Grid-Layout-Kontext.

Syntax

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

Siehe auch dlgGridLayout, dlgHBoxLayout, dlgVBoxLayout, Layout-Information, Ein vollständiges Beispiel

Das `dlgCell`-Statement definiert die Lage einer Cell in einem Grid-Layout-Kontext.

Der Index für Reihe (row) und Spalte (column) beginnt mit 0, so das die obere linke Cell den Index (0, 0) hat.

Mit zwei Parametern wird das Dialog-Objekt, das in `statement` angegeben wurde, in einer Cell an der Stelle `row` und `column` plaziert. Mit vier Parametern erstreckt sich das Objekt über alle Cells von `row/column` bis zu `row2/column2`.

Standardmäßig enthält `dlgCell` ein dlgHBoxLayout. Enthält eine Cell mehr als ein Dialog-Objekt, werden diese nebeneinander horizontal angeordnet.

Beispiel

```
string Text;
dlgGridLayout {
  dlgCell(0, 0) dlgLabel("Cell 0,0");
  dlgCell(1, 2, 4, 7) dlgTextEdit(Text);
  }
```

dlgCheckBox

Funktion

Definiert eine Checkbox.

Syntax

`dlgCheckBox(string Text, int &Checked) [ statement ]`

Siehe auch dlgRadioButton, dlgGroup, Layout-Information, Ein vollständiges Beispiel

Das `dlgCheckBox`-Statement definiert eine Checkbox mit dem angegebenen `Text`.

Wenn `Text` ein `'&'` enthält, wird das folgende Zeichen als Hotkey markiert. Wenn der Benutzer `Alt+hotkey` drückt, wird die Checkbox selektiert/deselektiert. Um ein `'&'`-Zeichen im Text zu verwenden, muss er mit einem Escape-Zeichen markiert werden.

`dlgCheckBox` wird hauptsächlich in dlgGroup benutzt, kann aber auch anders verwendet werden.
Alle Check-Boxen innerhalb eines gemeinsamen Dialogs müssen **unterschiedliche** `Checked`-Variablen haben!

Wenn ein Benutzer eine `dlgCheckBox` wählt, wird die entsprechende `Checked`-Variable auf `1` gesetzt, andernfalls ist sie auf `0` gesetzt. Der ursprüngliche Wert von `Checked` definiert, ob eine Checkbox anfänglich selektiert ist oder nicht. Wenn `Checked` ungleich `0` ist, ist die Checkbox defaultmäßig selektiert.

Das optionale `statement` wird jedesmal ausgeführt, wenn Sie die `dlgCheckBox` selektieren/deselektieren.

Beispiel

```
int mirror = 0;
int rotate = 1;
int flip   = 0;
dlgGroup("Orientation") {
  dlgCheckBox("&Mirror", mirror);
  dlgCheckBox("&Rotate", rotate);
  dlgCheckBox("&Flip", flip);
  }
```

dlgComboBox

Funktion

Definiert ein Combo-Box-Auswahl-Feld.

Syntax

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

Siehe auch dlgListBox, dlgLabel, Layout-Information, Ein vollständiges Beispiel

Das `dlgComboBox`-Statement definiert ein Combo-Box-Auswahlfeld mit dem Inhalt von `array`.

`Selected` reflektiert den Index des selektieren Combo-Box-Eintrags. Der erste Eintrag hat den Index `0`.

Jedes Element von `array` legt den Inhalt eines Eintrags in der Combo-Box fest. Keiner der Strings in `array` darf leer sein (sollte ein leerer String existieren, werden alle folgenden, inklusive des leeren, ignoriert).

Das optionale `statement` wird jedesmal ausgeführt, wenn die Auswahl in der `dlgComboBox` verändert wird.
Bevor `statement` ausgeführt wird, werden alle Variablen , die in den Dialog-Objekten verwendet werden neu eingelesen und jede Veränderung innerhalb von `statement` wird im Dialog angezeigt.

Ist der Ausgangswert von `Selected` ausserhalb des Bereichs der Indices von `array`, wird dieser auf `0` gesetzt.

Beispiel

```
string Colors[] = { "red", "green", "blue", "yellow" };
int Selected = 2; // initially selects "blue"
dlgComboBox(Colors, Selected) dlgMessageBox("You have selected " +
Colors[Selected]);
```

dlgDialog

Funktion

Führt einen User-Language-Dialog aus.

Syntax

```
int dlgDialog(string Title) block ;
```

Rückgabewert

Die `dlgDialog`-Funktion liefert einen Integer-Wert, dem durch den Aufruf der `dlgAccept()`-Funktion eine benutzerspezifische Bedeutung zugeordnet werden kann.
Wird der Dialog einfach geschlossen, ist der Rückgabewert `0`.

Siehe auch dlgGridLayout, dlgHBoxLayout, dlgVBoxLayout, dlgAccept, dlgReset, dlgReject, Ein vollständiges Beispiel

Die `dlgDialog`-Funktion, die durch `block` definiert wird. Das ist das einzige Dialog-Objekt das tatsächlich eine User-Language-Builtin-Funktion ist. Sie kann überall wo ein Funktionsaufruf erlaubt ist, verwendet werden.

`block` enthält normalerweise andere Dialog-Objekte. Man kann aber auch andere User-

Language-Statements verwenden, zum Beispiel, um bedingungsabhängig dem Dialog Objekte hinzuzufügen (siehe das zweite der folgenden Beispiele).

Standardmäßig enthält `dlgDialog` ein dlgVBoxLayout, so dass man sich bei einem einfachen Dialog um das Layout kein Gedanken machen muss.

Ein `dlgDialog` sollte an einer Stelle den Aufruf der `dlgAccept()` -Funktion enthalten, um dem Benutzer zu erlauben, den Dialog zu schließen und dessen Inhalt zu akzeptieren.

Wenn Sie nur eine einfache Message-Box oder einen einfachen Dialog brauchen, können Sie statt dessen auch einen der Vordefinierten Dialoge verwenden.

Beispiele

```
int Result = dlgDialog("Hello") {
  dlgLabel("Hello world");
  dlgPushButton("+OK") dlgAccept();
  };
int haveButton = 1;
dlgDialog("Test") {
  dlgLabel("Start");
  if (haveButton)
     dlgPushButton("Here") dlgAccept();
  };
```

dlgGridLayout

Funktion

Öffnet einen Grid-Layout-Kontext.

Syntax

`dlgGridLayout` *`statement`*

Siehe auch dlgCell, dlgHBoxLayout, dlgVBoxLayout, Layout-Information, Ein vollständiges Beispiel

Das `dlgGridLayout`-Statement öffnet einen Grid-Layout-Kontext.

Das einzige Dialog-Objekt, das direkt in `statement` verwendet werden kann, ist dlgCell, das die Position eines Dialog-Objekts im Grid-Layout festlegt.

Die Indices für row und column beginnen mit 0, so dass die obere linke Cell den Index (0, 0) hat.
Die Anzahl der Reihen und Spalten wird automatisch an die Position von Dialog-Objekten, die innerhalb des Grid-Layout-Kontexts definiert werden, angepasst. Die Anzahl der Reihen und Spalten muss nicht explizit definiert werden.

Beispiel

```
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

Funktion

Definiert ein Group-Feld.

Syntax

```
dlgGroup(string Title) statement
```

Siehe auch dlgCheckBox, dlgRadioButton, Layout-Information, Ein vollständiges Beispiel

Das `dlgGroup`-Statement definiert eine Gruppe mit dem gegebenen `Title`.

Standardmäßig enthält `dlgGroup` ein dlgVBoxLayout, so braucht man sich bei einer einfachen Group keine Gedanken zum Layout machen.

`dlgGroup` wird hauptsächlich für einen Satz von Radio-Buttons oder Check-Boxes verwendet, kann aber auch jedes andere beliebige Objekt in `statement` enthalten. Radio-Buttons in einer `dlgGroup` werden mit `0` beginnend numeriert.

Beispiel

```
int align = 1;
dlgGroup("Alignment") {
  dlgRadioButton("&Top", align);
  dlgRadioButton("&Center", align);
  dlgRadioButton("&Bottom", align);
  }
```

dlgHBoxLayout

Funktion

Öffnet einen Horizontal-Box-Layout-Kontext.

Syntax

```
dlgHBoxLayout statement
```

Siehe auch dlgGridLayout, dlgVBoxLayout, Layout-Information, Ein vollständige Beispiel

Das `dlgHBoxLayout`-Statement öffnet einen Horizontal-Box-Layout-Kontext für das angegebene `statement`.

Beispiel

```
dlgHBoxLayout {
  dlgLabel("Box 1");
  dlgLabel("Box 2");
  dlgLabel("Box 3");
  }
```

dlgIntEdit

Funktion

Definiert ein Integer-Eingabe-Feld.

Syntax

```
dlgIntEdit(int &Value, int Min, int Max)
```

Siehe auch dlgRealEdit, dlgStringEdit, dlgLabel, Layout-Information, Ein vollständiges Beispiel

Das `dlgIntEdit`-Statement definiert ein Integer-Eingabe-Feld mit einem in `Value` angegebenen Wert.

Ist `Value` ursprünglich ausserhalb des Bereichs `Min` und `Max`, wird er auf diesen Bereich limitiert.

Beispiel

```
int Value = 42;
dlgHBoxLayout {
  dlgLabel("Enter a &Number between 0 and 99");
  dlgIntEdit(Value, 0, 99);
  }
```

dlgLabel

Funktion

Definiert ein Text-Label.

Syntax

```
dlgLabel(string Text [, int Update])
```

Siehe auch Layout-Information, Ein vollständiges Beispiel, dlgRedisplay()

Das `dlgLabel`-Statement definiert ein Label mit dem angegebenen `Text`.

`Text` kann entweder ein fester String wie `"Hello"` sein, oder eine String-Variable.

Falls `Text` HTML-Tags enthält, so müssen die Zeichen '<', '>' und '&', damit sie als solche angezeigt werden, als "<", ">" bzw. "&" angegeben werden.

Wenn der `Update`-Parameter nicht `0` ist und `Text` eine String-Variable, kann deren Inhalt im `statement` z. B. eines dlgPushButton modifiziert werden, wodurch das Label automatisch aktualisiert wird. Das ist natürlich nur sinnvoll wenn `Text` eine eindeutig bestimmte String-Variable ist (und beispielsweise keine Loop-Variable eines `for`-Statements).

Enthält `Text` ein `'&'`-Zeichen, und kann das Objekt, das auf das Label folgt, den Keyboard-Fokus bekommen, wird das folgende Zeichen zum Hot-Key. Drückt der Benutzer `Alt+hotkey`, wird das Objekt, das direkt nach `dlgLabel` definiert wurde, aktiv. Um ein `'&'`-Zeichen direkt im Text zu verwenden, muss man es mit einem Escape-Zeichen markieren.

Beispiel

```
string OS = "Windows";
dlgHBoxLayout {
  dlgLabel(OS, 1);
  dlgPushButton("&Change OS") { OS = "Linux"; }
  }
```

dlgListBox

Funktion

Definiert ein List-Box-Auswahl-Feld.

Syntax

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

Siehe auch dlgComboBox, dlgListView, dlgLabel, Layout-Information, Ein vollständiges Beispiel

Das `dlgListBox`-Statement definiert ein List-Box-Auswahl-Feld mit dem Inhalt von `array`.

`Selected` gibt den Index des selektierten List-Box-Eintrags wieder. Der erste Eintrag hat den Index `0`.

Jedes Element von `array` legt den Inhalt einer Zeile in der List-Box fest. Keiner der Strings in `array` darf leer sein (sollte ein leerer String existieren, werden alle folgenden, inklusive des leeren, ignoriert).

Das optionale `statement` wird immer dann ausgeführt, wenn der Benutzer einen Doppelklick auf einen Eintrag der `dlgListBox` ausführt.
Bevor `statement` ausgeführt wird, werden alle Variablen, die von Dialog-Objekten benutzt werden, aktualisiert. Alle Änderungen, die in `statement` gemacht wurden, wirken sich auf den Dialog aus, sobald das Statement zurückgegeben wird.

Ist der Ausgangswert von `Selected` ausserhalb des Index-Bereichs von `array`, wird kein Eintrag selektiert.

Beispiel

```
string Colors[] = { "red", "green", "blue", "yellow" };
int Selected = 2; // initially selects "blue"
dlgListBox(Colors, Selected) dlgMessageBox("You have selected " +
Colors[Selected]);
```

dlgListView

Funktion

Definiert ein mehrspaltiges List-View-Auswahl-Feld.

Syntax

`dlgListView(string Headers, string array[], int &Selected[, int &Sort]) [ statement ]`

Siehe auch dlgListBox, dlgLabel, Layout-Information, Ein vollständiges Beispiel

Das `dlgListView`-Statement definiert ein mehrspaltiges List-View-Auswahl-Feld mit dem Inhalt, der in `array` angegeben ist.

`Headers` definiert die durch Tabulatoren getrennte Liste der Spalten-Überschriften.

`Selected` gibt den Index des selektierten List-View-Eintrags von `array` wieder (die Reihenfolge in der die Einträge tatsächlich angezeigt werden, kann unterschiedlich sein, da der Inhalt einer `dlgListView` in den verschiedenen Spalten sortiert werden kann). Der erste Eintrag hat den Index `0`.
Wenn kein spezieller Eintrag selektiert werden soll, wählt man für `Selected` den Wert `-1`.

`Sort` gibt an, nach welcher Spalte der List-View sortiert werden soll. Die linke Spalte hat die Nummer `1`. Das Vorzeichen dieses Parameters legt die Richtung der Sortierung fest (positive Werte sortieren in aufsteigender Reihenfolge). Falls `Sort` den Wert `0` hat, oder außerhalb der gültigen Anzahl von Spalten liegt, wird nicht sortiert. Der Rückgabewert von `Sort` spiegelt die vom Benutzer durch Anklicken der Spalten-Header gewählte Sortierspalte und -richtung wieder. Standardmäßig wird nach der ersten Spalte, in aufsteigender Richtung sortiert.

Jedes Element von `array` legt den Inhalt einer Zeile in der List-View fest und muss durch Tabulatoren getrennte Werte enthalten. Sind weniger Werte eines Elements in `array` definiert als im `Headers`-String vorgegeben, bleiben die restlichen Felder leer. Sind mehr Werte eines Element in `array` angegeben als im `Headers`-String, werden die überzählgen stillschweigend ignoriert. Keiner der Strings in `array` darf leer sein (sollte ein leerer String vorhanden sein, werden alle nachfolgenden, inklusive dem Leerstring ignoriert).

Enthält ein Listeneintrag Zeilenumbrüche (`'\n'`), so wird er entsprechend mehrzeilig dargestellt.

Das optionale `statement` wird ausgeführt, wann immer der Benutzer auf einen Eintrag in `dlgListView` doppelklickt.
Bevor `statement` ausgeführt wird, werden alle Variablen, die mit den Dialog-Objekten benutzt wurden, aktualisiert. Alle Änderungen, die in `statement` gemacht wurden, wirken sich auf den Dialog aus, sobald das Statement zurückgegeben wird.

Ist der Ausgangswert von `Selected` ausserhalb des Index-Bereichs von `array`, wird kein Eintrag selektiert.

Ist `Headers` ein leerer String, wird das erste Element von `array` als Header-String benutzt. Folglich ist der Index des ersten Eintrags dann `1`.

Der Inhalt von `dlgListView` kann in einer beliebigen Spalte sortiert werden, indem man auf dessen Spalten-Header klickt. Die Spalten-Reihenfolge kann man durch Anklicken&Ziehen des Spalten-Headers verändern. Beachten Sie, dass keine dieser Änderunen eine Auswirkung auf den Inhalt von `array` hat. Soll der Inhalt alphanumerisch sortiert werden, kann ein `numeric string[]`-Array verwendet werden.

Beispiel

```
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

Funktion

Definiert einen Push-Button.

Syntax

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

Siehe auch Layout-Information, Dialog-Funktionen, Ein vollständige Beispiel

Das `dlgPushButton`-Statement definiert einen Push-Button mit dem angegebenen `Text`.

Enthält `Text` ein `'&'`-Zeichen, wird das folgende Zeichen zum Hot-Key. Wenn der Benutzer dann `Alt+hotkey` drückt, wird dieser Button selektiert. Soll ein `'&'`-Zeichen im Text verwendet werden, muss es mit einem Escape-Zeichen markiert werden.

Beginnt `Text` mit einem `'+'`-Zeichen, wird dieser Button der Default-Button. Dieser wird betätigt, wenn der Benutzer ENTER drückt.
Wenn `Text` mit einem `'-'`-Zeichen beginnt, wird dieser Button der Cancel-Button. Dieser wird gewählt wenn der Benutzer den Dialog schließt.
Achtung: Stellen Sie sicher, dass das `statement` eines so markierten Buttons einen Aufruf von dlgReject() enthält! Ansonsten ist es dem Benutzer nicht möglich den Dialog überhaupt zu schließen!
Um ein `'+'` oder `'-'`-Zeichen als erstes Zeichen des Textes zu verwenden, muss es mit einem Escape-Zeichen markiert werden.

Wenn der Benutzer einen `dlgPushButton` selektiert, wird das angegebene `statement` ausgeführt.
Bevor `statement` ausgeführt wird, werden alle Variablen, die mit den Dialog-Objekten benutzt wurden, aktualisiert. Alle Änderungen, die in `statement` gemacht wurden, wirken sich auf den Dialog aus, sobald das Statement zurückgegeben wird.

Beispiel

```
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

Funktion

Definiert einen Radio-Button.

Syntax

```
dlgRadioButton(string Text, int &Selected) [ statement ]
```

Siehe auch dlgCheckBox, dlgGroup, Layout-Information, Ein vollständiges Beispiel

Das `dlgRadioButton`-Statement definiert einen Radio-Button mit dem angegebenen `Text`.

Enthält `Text` ein `'&'`-Zeichen, wird das folgende Zeichen zum Hot-Key. Wenn der Benutzer dann `Alt+hotkey` drückt, wird dieser Button selektiert. Soll ein `'&'`-Zeichen im Text verwendet werden, muss es mit einem Escape-Zeichen markiert werden.

`dlgRadioButton` kann nur innerhalb einer dlgGroup verwendet werden.
Alle Radio-Buttons innerhalb derselben Group müssen **dieselbe** `Selected`-Variable haben!

Wenn der Benutzer einen `dlgRadioButton` selektiert, wird der Index dieses Buttons innerhalb der `dlgGroup` in der `Selected`-Variablen gespeichert.
Der Ausgangswert von `Selected` definiert, welcher Radio-button per default selektiert ist. Liegt `Selected` ausserhalb des gültigen Bereichs dieser Group, ist kein Radio-Button selektiert. Um die richtige Radio-Button-Selektion zu erhalten, muss `Selected` schon **vor** der Definition des ersten `dlgRadioButton` festgelegt werden, und darf nicht verändert werden, während man weitere Radio-Buttons hinzufügt. Ansonsten ist es ungewiss welcher Radio-Button (wenn überhaupt einer) selektiert wird.

Das optionale `statement` wird ausgeführt, wenn ein `dlgRadioButton` selektiet wird.

Beispiel

```
int align = 1;
dlgGroup("Alignment") {
  dlgRadioButton("&Top", align);
  dlgRadioButton("&Center", align);
  dlgRadioButton("&Bottom", align);
  }
```

dlgRealEdit

Funktion

Definiert ein Real-Eingabe-Feld.

Syntax

```
dlgRealEdit(real &Value, real Min, real Max)
```

Siehe auch dlgIntEdit, dlgStringEdit, dlgLabel, Layout-Information, Ein vollständiges Beispiel

Das `dlgRealEdit`-Statement definiert ein Real-Eingabe-Feld mit dem angegebenen

`Value` (Wert).

Wenn `Value` ursprünglich ausserhalb des Bereiches von `Min` und `Max` liegt, wird dieser auf diese Werte begrenzt.

Beispiel

```
real Value = 1.4142;
dlgHBoxLayout {
  dlgLabel("Enter a &Number between 0 and 99");
  dlgRealEdit(Value, 0.0, 99.0);
  }
```

dlgSpacing

Funktion

Definiert zusätzlichen Abstand in einem Box-Layout-Kontext.

Syntax

```
dlgSpacing(int Size)
```

Siehe auch dlgHBoxLayout, dlgVBoxLayout, dlgStretch, Layout-Information, Ein vollständiges Beispiel

Das `dlgSpacing`-Statement definiert zusätzlichen Abstand in einem Vertical- bzw. Horizontal-Box-Layout-Kontext.

`Size` definiert die Anzahl der Pixel des zusätzlichen Abstands.

Beispiel

```
dlgVBoxLayout {
  dlgLabel("Label 1");
  dlgSpacing(40);
  dlgLabel("Label 2");
  }
```

dlgSpinBox

Funktion

Definiert ein Spin-Box-Auswahl-Feld.

Syntax

```
dlgSpinBox(int &Value, int Min, int Max)
```

Siehe auch dlgIntEdit, dlgLabel, Layout-Information, Ein vollständiges Beispiel

Das `dlgSpinBox`-Statement definiert ein Spin-Box-Auswahl-Feld mit dem angegebenen `Value`.

Wenn `Value` ursprünglich ausserhalb des Bereiches von `Min` und `Max` liegt, wird dieser auf diese Werte begrenzt.

Beispiel

```
int Value = 42;
dlgHBoxLayout {
  dlgLabel("&Select value");
  dlgSpinBox(Value, 0, 99);
  }
```

dlgStretch

Funktion

Definiert einen leeren, dehnbaren Abstand in einem Box-Layout-Kontext.

Syntax

```
dlgStretch(int Factor)
```

Siehe auch dlgHBoxLayout, dlgVBoxLayout, dlgSpacing, Layout-Information, Ein vollständiges Beispiel

Das `dlgStretch`-Statement definiert einen leeren dehnbaren Abstand in einem Vertical- oder einem Horizontal-Box-Layout-Kontext.

`Factor` definiert den Dehnungsfaktor des Abstands.

Beispiel

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

dlgStringEdit

Funktion

Definiert ein String-Eingabe-Feld.

Syntax

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

Siehe auch dlgRealEdit, dlgIntEdit, dlgTextEdit, dlgLabel, Layout-Information, Ein vollständiges Beispiel

Das `dlgStringEdit`-Statement definiert ein Text-Eingabe-Feld mit dem angegebenen `Text`.

Beispiel

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

dlgTabPage

Funktion

Definiert eine Tab-Page.

Syntax

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

Siehe auch dlgTabWidget, Layout-Information, Ein vollständiges Beispiel

Das `dlgTabPage`-Statement definiert eine Tab-Page mit dem angegebenen `Title`, die `statement` enthält.

Enthält `Title` ein `'&'`-Zeichen, wird das folgende Zeichen zum Hot-Key. Drückt der Benutzer `Alt+hotkey`, wird diese Tab-Page geöffnet. Soll im Text ein `'&'`-Zeichen verwendet werden, muss es mit einem Escape-Zeichen markiert werden.

Tab-Pages können nur innerhalb eines dlgTabWidget verwendet werden.

Standardmäßig enthält `dlgTabPage` ein dlgVBoxLayout, so dass man sich bei einer einfachen Tab-Page nicht um das Layout kümmern muss.

Beispiel

```
dlgTabWidget {
  dlgTabPage("Tab &1") {
    dlgLabel("This is page 1");
    }
  dlgTabPage("Tab &2") {
    dlgLabel("This is page 2");
    }
  }
```

dlgTabWidget

Funktion

Definiert einen Container für Tab-Pages.

Syntax

```
dlgTabWidget statement
```

Siehe auch dlgTabPage, Layout-Information, Ein vollständiges Beispiel

Das `dlgTabWidget`-Statement definiert einen Platzhalter für einen Satz von Tab-Pages.

`statement` ist eine Liste eines oder mehrerer dlgTabPage-Objekte. Es dürfen keine anderen Dialog-Objekte in dieser Liste enthalten sein.

Beispiel

```
dlgTabWidget {
  dlgTabPage("Tab &1") {
    dlgLabel("This is page 1");
    }
  dlgTabPage("Tab &2") {
```

```
    dlgLabel("This is page 2");
    }
  }
```

dlgTextEdit

Funktion

Definiert ein mehrzeiliges Text-Eingabe-Feld.

Syntax

```
dlgTextEdit(string &Text)
```

Siehe auch dlgStringEdit, dlgTextView, dlgLabel, Layout-Information, Ein vollständiges Beispiel

Das `dlgTextEdit`-Statement definiert ein mehrzeiliges text-Eingabe-Feld mit dem angegebenen `Text`.

Die einzelnen Zeilen in `Text` müssen mit einem Newline-Zeichen (`'\n'`) getrennt werden. Beliebige Leerzeichen am Ende der `Text`-Zeilen werden gelöscht. Leere Zeilen am Endes des Textes werden vollständig entfernt.

Beispiel

```
string Text = "This is some text.\nLine 2\nLine 3";
dlgVBoxLayout {
  dlgLabel("&Edit the text");
  dlgTextEdit(Text);
  }
```

dlgTextView

Funktion

Definiert ein mehrzeiliges Text-Viewer-Feld.

Syntax

```
dlgTextView(string Text)
dlgTextView(string Text, string &Link) statement
```

Siehe auch dlgTextEdit, dlgLabel, Layout-Information, Ein vollständiges Beispiel

Das `dlgTextView`-Statement definiert ein mehrzeiliges Text-Viewer-Feld mit dem angegebenen `Text`.

Der `Text` darf HTML-Tags enthalten.

Falls `Link` angegeben wird und der `Text` Hyperlinks enthält, wird `statement` ausgeführt wenn der Benutzer auf einen Hyperlink klickt, wobei der Wert von `Link` auf das gesetzt wird, was im `<a href=...>`-Tag als Wert für *href* angegeben wurde.

Beispiel

```
string Text = "This is some text.\nLine 2\nLine 3";
```

```
dlgVBoxLayout {
  dlgLabel("&View the text");
  dlgTextView(Text);
  }
```

dlgVBoxLayout

Funktion

Öffnet einen Vertical-Box-Layout-Kontext.

Syntax

`dlgVBoxLayout` *`statement`*

Siehe auch dlgGridLayout, dlgHBoxLayout, Layout-Information, Ein vollständiges Beispiel

Das `dlgVBoxLayout`-Statement öffnet einen Vertical-Box-Layout-Kontext für das angegebene `statement`.

Standardmäßig enthält dlgDialog ein `dlgVBoxLayout`, so dass man sich bei einfachen Dialogen keine Gedanken zum Layout machen muss.

Beispiel

```
dlgVBoxLayout {
  dlgLabel("Box 1");
  dlgLabel("Box 2");
  dlgLabel("Box 3");
  }
```

Layout-Information

Alle Objekte eines User-Language-Dialogs werden in einem *Layout-Kontext* verwendet.

Es gibt verschiedene Layout-Kontexte, wie grid, horizontal oder vertical.

Grid-Layout-Kontext

Objekte in einem Grid-Layout-Kontext müssen die Raster-Koordinaten der Zelle (Cell) oder der Zellen angeben, in der/denen sie plaziert werden sollen. Um einen Text in Reihe (row) 5, Spalte (column) 2 zu plazieren, schreiben Sie

```
dlgGridLayout {
  dlgCell(5, 2) dlgLabel("Text");
  }
```

Soll das Objekt über mehr als eine Zelle gehen, müssen Sie die Koordinaten der Start-Zelle und der End-Zelle angeben. Um eine Group zu plazieren, die sich von Reihe 1, Spalte 2 über Reihe 3, Spalte 5 erstreckt, schreiben Sie

```
dlgGridLayout {
  dlgCell(1, 2, 3, 5) dlgGroup("Title") {
    //...
    }
  }
```

Horizontal-Layout-Kontext

Objekte in einem Horizontal-Layout-Kontext werden von links nach rechts plaziert.

Die Sonder-Objekte dlgStretch und dlgSpacing können verwendet werden, um die Verteilung der Abstände zu verfeinern.

Um zwei Buttons zu definieren, die sich bis an den rechten Rand des Dialogs erstrecken, schreiben Sie

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

Vertical-Layout-Kontext

Objekte in einem Vertical-Layout-Kontext folgen denselben Vorschriften wie die in einem Horizontal-Layout-Kontext mit dem Unterschied, dass sie von oben nach unten angeordnet werden.

Gemischter Layout-Kontext

Vertical-, Horizontal- und Grid-Layout-Kontexte können gemischt werden, um die gewünschte Dialog-Struktur zu ezeugen. Siehe Ein vollständiges Beispiel.

Dialog-Funktionen

Folgende Funktionen können mit User-Language-Dialogen verwendet werden:

dlgAccept()	schließt den Dialog und akzeptiert dessen Inhalt
dlgRedisplay()	aktualisiert den Dialog nachdem beliebige Werte verändert wurden
dlgReset()	setzt alle Dialog-Objekte auf den Ursprungswert zurück
dlgReject()	schließt den Dialog und verwirft dessen Inhalt

dlgAccept()

Funktion

Schließt den Dialog und akzeptiert dessen Inhalt.

Syntax

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

Siehe auch dlgReject, dlgDialog, Ein vollständiges Beispiel

Die `dlgAccept`-Funktion schließt dlgDialog, und kehrt zurück nachdem das aktuelle Statement beendet wurde.

Jede Änderung, die der Benutzer im Dialog macht, wird übernommen und an die Variablen, die bei der Definition der Dialog-Objekte angegeben wurden, kopiert.

Die optionale Angabe von `Result` ist der Wert der vom Dialog geliefert wird. Das sollte typischerweise ein positiver Integer-Wert sein. Wird kein Wert angegeben, ist der Standardwert gleich `1`.

Bitte beachten Sie, dass `dlgAccept()` wieder in die normale Programm- Routine zurückkehrt, so wie in dieser Sequenz:

```
dlgPushButton("OK") {
  dlgAccept();
  dlgMessageBox("Accepting!");
  }
```

Das Statement nach `dlgAccept()` wird noch ausgeführt!

Beispiel

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

dlgRedisplay()

Funktion

Aktualisiert den Dialog-Inhalt nachdem Werte verändert wurden.

Syntax

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

Siehe auch dlgReset, dlgDialog, Ein vollständiges Beispiel

Die `dlgRedisplay`-Funktion wird aufgerufen, um den dlgDialog, nach dem Verändern von Variablen, die in den Dialog-Objekten definiert wurden, zu aktualisieren.

Sie brauchen nur `dlgRedisplay()` aufrufen, wenn der Dialog während der Ausführung des Programmcodes aktualisiert werden soll. Im folgenden Beispiel wird der Status auf "Running..." gesetzt und `dlgRedisplay()` muss aufgerufen werden, um die Änderungen für die Ausführung des Programms wirksam zu machen. Nach dem Ändern des Status auf "Finished.", braucht man `dlgRedisplay()` nicht mehr aufrufen, da alle Dialog-Objekte nach dem Verlassen des Statements aktualisiert werden.

Beispiel

```
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()

Funktion

Setzt alle Dialog-Objekte auf ihren ursprünglichen Wert.

Syntax

```
void dlgReset(void);
```

Siehe auch dlgReject, dlgDialog, Ein vollständiges Beispiel

Die `dlgReset`-Funktion kopiert die ursprünglichen Werte in alle Dialog-Objekte des aktuellen dlgDialog zurück.

Alle Änderungen, die der Benutzer im Dialog machte, werden verworfen.

Ein Aufruf von `dlgReject()` impliziert einen Aufruf von `dlgReset()`.

Beispiel

```
int Number = 1;
int Result = dlgDialog("Test") {
               dlgIntEdit(Number);
               dlgPushButton("+OK")    dlgAccept(42);
               dlgPushButton("Cancel") dlgReject();
               dlgPushButton("Reset")  dlgReset();
               };
```

dlgReject()

Funktion

Schließt den Dialog und verwirft seinen Inhalt.

Syntax

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

Siehe auch dlgAccept, dlgReset, dlgDialog, Ein vollständiges Beispiel

Die `dlgReject`-Funktion veranlasst, dass dlgDialog geschlossen wird und nach dem Beenden der aktuellen Statement-Sequenz zurückkehrt.

Jede Änderung, die der Benutzer im Dialog machte, wird verworfen. Die Variablen, die während der Definition der Dialog-Objekte übergeben wurden, werden auf Ihren ursprünglichen Wert zurückgesetzt.

Der optionale Wert für `Result` wird vom Dialog zurückgegeben. Typischerweise ist das `0` oder ein negativer Integer-Wert. Wird kein Wert angegeben, ist er standardmäßig `0`.

Beachten Sie, dass `dlgReject()` wieder in die normale Programm-Routine zurückkehrt,

wie in dieser Sequenz:

```
dlgPushButton("Cancel") {
  dlgReject();
  dlgMessageBox("Rejecting!");
  }
```

Das Statement nach `dlgReject()` wird auch noch ausgeführt!

Der Aufruf von `dlgReject()` impliziert den Aufruf von `dlgReset()`.

Beispiel

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

Escape-Zeichen

Einige Zeichen haben in Schaltflächen- oder Label-Texten eine besondere Bedeutung, so dass sie mit *Escape-Zeichen* markiert werden müssen, wenn sie tatsächlich im Text erscheinen sollen.

Dazu müssen Sie dem Zeichen einen *Backslash* voranstellen, so wie in

```
dlgLabel("Miller \\& Co.");
```

Das Ergebnis wird im Dialog so aussehen: "Miller & Co."

Beachten Sie, dass hier in Wirklichkeit **zwei** Backslash-Zeichen verwendet wurden, da diese Zeile erst durch den User-Language-Parser geht, der den ersten Backslash abzieht.

Ein vollständiges Beispiel

Hier folgt ein vollständiges Beispiel eines User-Language-Dialogs:

```
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();
    }
  };
```

Unterstützte HTML-Tags

EAGLE unterstützt eine Teilmenge von Tags (Steuerzeichen), die zum Formatieren von HTML-Seiten verwendet werden. Damit kann man Texte von einigen Objekten im User-Language-Dialog, in der `#usage`-Directive oder in der Description von Bibliotheks-Objekten formatieren.

Text wird zu HTML-Text, wenn die erste Zeile ein Tag enthält. Wenn das nicht der Fall ist und Sie den Text formatieren wollen, schließen Sie den ganzen Text in das `<html>...</html>` Tag ein.

Die folgende Tabelle listet alle unterstützten HTML-Tags mit ihren verfügbaren Attributen auf:

Tag	**Beschreibung**
<html>...</html>	Ein HTML-Dokument. Es versteht folgende Attribute • `bgcolor` - Die Hintergrundfarbe, z. B. `bgcolor="yellow"` or `bgcolor="#0000FF"`. • `background` - Das Hintergrundbild, zum Beispiel `background="granit.xpm"`. • `text` - Die default Textfarbe, z. B. `text="red"`. • `link` - Die Farbe eines Links, z. B. `link="green"`.
...	Eine Haupt-Überschrift.
...	Eine untergeordnete Überschrift.
...	Eine weiter untergeordnete Überschrift.

...	Ein links-bündiger Abschnitt. Bestimmen Sie die Anordnung mit dem `align` Attribut. Mögliche Werte sind `left`, `right` und `center`.
...	Ein zentrierter Abschnitt.
<blockquote>...</blockquote>	Ein eingerückter Abschnitt, sinnvoll für Zitate.
...	Eine ungeordnete Liste. Sie können auch ein type-Argument angeben um einen Bullet-Style zu definieren. Default ist `type=disc`, andere Typen sind `circle` und `square`.
...	Eine geordnete Liste. Sie können auch ein type-Argument angeben um die Art der Nummerierung zu definieren. Default ist `type="1"`, andere Typen sind `"a"` und `"A"`.
...	Ein Punkt in einer Liste. Dieses Tag kann nur innerhalb eines `ol` oder `ul` Kontext verwendet werden.
<pre>...</pre>	Für größere Mengen von Code. Leerzeichen im Inhalt bleiben erhalten. Für kleinere Mengen Code, benutzen Sie den Inline-style `code`.
<a>...	Ein Anker oder Link. Folgende Attribute sind erlaubt • `href` - Das Referenz-Ziel wie in `<a href="target.qml">...</a>`. Sie dürfen auch einen zusätzlichen Anker innerhalb des angegebenen Ziel-Dokuments angeben, z. B. `<a href="target.qml#123">...</a>`. • `name` - Der Anker-Name, wie in `<a name="target">...</a>`.
...	Emphasized (kursiv)(genauso wie `<i>...</i>`).
...	Stark (genauso wie `<b>...</b>`).
...	Kursiver Text.
...	Fetter Text.
<u>...</u>	Unterstrichener Text.

<big>...</big>	Eine größere Texthöhe.
...	Eine kleinere Texthöhe.
<code>...</code>	Kennzeichnet Code. (wie auch `<tt>...</tt>`. Für größere Mengen an Code, verwenden Sie das Block-Tag `pre`.
<tt>...</tt>	Typewriter Schriftart.
...	Zur Bestimmung von Texthöhe, Schrift-Familie und Textfarbe. Das Tag versteht folgende Attribute: • `color` - Die Textfarbe, z. B. `color="red"` oder `color="#FF0000"`. • `size` - Die logische Größe der Schrift. Logische Größen von 1 bis 7 werden unterstützt. Der Wert darf entweder absolut, z. B. `size=3`, oder relativ, wie `size=-2` sein. Im letzten Fall werden die Größen einfach addiert. • `face` - Die Schriftart-Familie, z. B. `face=times`.
<img...>	Ein Bild. Dieses Tag versteht die folgenden Attribute: • `src` - Den Namen des Bildes, z. B. `<img src="image.xpm">`. Unterstützte Bildformate sind: ".bmp" (Windows Bitmap Dateien) ".pbm" (Portable Bitmap Dateien) ".pgm" (Portable Grayscale Bitmap Dateien) ".png" (Portable Network Graphics Dateien) ".ppm" (Portable Pixelmap Dateien) ".xbm" (X Bitmap Dateien) ".xpm" (X Pixmap Dateien) • `width` - Die Breite des Bildes. Passt das Bild nicht in die angegebene Größe, wird es automatisch skaliert. • `height` - Die Höhe des Bildes. • `align` - Bestimmt wo das Bild plaziert wird. Defaultmäßig wird ein Bild "inline" plaziert, genauso wie ein Buchstabe. Legen Sie `left` oder `right` fest, um das Bild an der entsprechenden Stelle zu plazieren.
<hr>	Eine waagrechte Linie.
 	Ein Zeilenumbruch.
<nobr>...</nobr>	Kein Zeilenumbruch.Erhält "Word Wrap".
<table>...</table>	Eine Tabellen-Definition. Die Standardtabelle ist ohne Rahmen. Geben Sie das boolsche Attribut `border` an um einen Rahmen zu

	erhalten. Andere Attribute sind: • `bgcolor` - Die Hintergrundfarbe. • `width` - Die Tabellenbreite. Wird entweder in Pixel oder in Prozent der Spaltenbreite angegeben, z. B. `width=80%`. • `border` - Die Breite des Tabellenrandes. Default ist 0 (= kein Rand). • `cellspacing` - Zusätzlicher Leerraum um die Tabellenzelle. Default ist 2. • `cellpadding` - Zusätzlicher Leerraum um den Inhalt einer Tabellenzelle. Default ist 1.
<tr>...</tr>	Eine Tabellen-Reihe. Kann nur mit `table` verwendet werden. Versteht die Attribute: • `bgcolor` - Die Hintergrundfarbe.
<td>...</td>	Eine Zelle in einer Tabelle. Kann nur innerhalb `tr` verwendet werden.Versteht die Attribute: • `bgcolor` - Die Hintergrundfarbe. • `width` - Die Zellenbreite. Wird entweder in Pixel oder in Prozent der gesamten Tabellenbreite angegeben, z. B. `width=50%`. • `colspan` - Legt fest wieviele Spalten diese Zelle belegt. Default ist 1. • `rowspan` - Legt fest wieviele Reihen diese Zelle belegt. Default ist 1. • `align` - Positionierung, mögliche Angaben sind `left`, `right` und `center`. Default ist links-bündig.
<th>...</th>	Eine "Header"-Zelle in der Tabelle. Wie `td` aber als default mit zentrierter Ausrichtung und fetter Schriftart.
<author>...</author>	Markiert den Autor des Texts.
<dl>...</dl>	Eine Definitions-Liste.
<dt>...</dt>	Ein Definitions-Tag. Kann nur innerhalb `dl` verwendet werden.
<dd>...</dd>	Definitions-Daten. Kann nur innerhalb `dl` verwendet werden.

Tag	**Bedeutung**
&lt;	<
&gt;	>

&amp;	&
	Leerzeichen ohne Umbruch
&auml;	ä
&ouml;	ö
&uuml;	ü
&Auml;	Ä
&Ouml;	Ö
&Uuml;	Ü
&szlig;	ß
&copy;	©
&deg;	°
&micro;	μ
&plusmn;	±