RNL language standard, version 0.3

Version 0.3 · revision 2 + later changes (2026-11-08, c82633b)

RNL (Rich Netlist) is written in the comments of a circuit netlist. It describes how the circuit is drawn as a schematic and what its parts mean, while the netlist alone defines the circuit. This document is the standard of version 0.3 of the language.

- The English text is normative. The Chinese version, `rnl-0.3.zh.md`, is a translation; where the two differ, the English text prevails.
- The word list `rnl-words-0.3.json` and its schema `rnl-words.schema.json` are part of this standard (5.3, Annex A).
- The principles of RNL are in `rich-netlist.md` (English, authoritative; Chinese version: `rich-netlist.zh.md`); this standard refers to them as "Principles".
- Versions 0.x make no promise of compatibility with each other (16.1).
- This text revises version 0.2 by the draft of version 0.3 in the rnl-dev research repository, commit d26d049 (2026-10-15); Annex D lists the changes.

Contents

1 Overview 6
1.1 Scope 6
1.2 Structure of this standard 6
1.3 At a glance 6
1.4 Terms and conventions 7
1.5 Core principles 10
1.6 Conformance 10
2 Embedding in the host 12
2.1 Two-stage parsing 12
2.2 Carriers 12
2.3 The entry 13
2.4 Extent of a record 13
2.5 Disabled records and host text 14
2.6 Levels 14
2.7 Package files 15
2.8 Record head 15
3 Lexical structure 17
3.1 Characters 17
3.2 Tokens 17
3.3 Free text 17
3.4 Comments 18
3.5 White space and adjacency 18
3.6 Names 18
3.7 Literals 19
3.8 Points 20
4 Syntax 21
4.1 Grammar 21
4.2 Constraints outside the grammar 22
4.3 Word order 22
4.4 Examples of errors 23
4.5 Standard sentences 24
5 Objects and statements 26
5.1 Objects 26
5.2 Sets 27
5.3 Statements and signatures 29
5.4 Ways of consuming input 32
5.5 Pipes and nesting 32
5.6 Definitions as objects 33
5.7 Handles and names 33
5.8 Keys of facts 34
5.9 Numbers of objects without a name 34
6 Names and scope 37
6.1 Hierarchy of levels 37
6.2 Prefixes 38
6.3 Host references 38
6.4 Terminals 38
6.5 Sub-objects 39
6.6 Naming 39
6.7 Scope of names 40

6.8 Inheritance and hiding ... 40
6.9 Names in definition bodies ... 40
6.10 Placeholders ... 41
6.11 Subscripts ... 41
6.12 Terminal names: DEVICE ... 42
7 Failures and diagnostics ... 45
7.1 Two phases of checking ... 45
7.2 Failed objects ... 45
7.3 Strict and tolerant inputs ... 45
7.4 Recovery ... 45
7.5 Conflicts are not failures ... 46
7.6 Diagnostics ... 46
8 Taking effect ... 48
8.1 A level takes effect after it is read completely ... 48
8.2 Phases ... 48
8.3 Appearance queues ... 48
8.4 Override by key ... 49
8.5 Conflicts ... 49
9 Definitions ... 52
9.1 Common rules ... 52
9.2 Symbols: DEFINE-SYMBOL ... 52
9.3 Styles: DEFINE-STYLE ... 56
9.4 General definitions: DEFINE ... 57
9.5 Uses of general definitions ... 60
10 Selections, rules and generation one by one ... 63
10.1 Selection keywords ... 63
10.2 The condition `wired` ... 66
10.3 Evaluating selections ... 66
10.4 Rules ... 66
10.5 Generation one by one ... 67
11 Placement and appearance ... 69
11.1 Position: PLACE ... 69
11.2 Orientation: ORIENT ... 69
11.3 Symbols and styles: SET-SYMBOL and SET-STYLE ... 70
11.4 Several drawings of one object ... 71
11.5 Groups: GROUP ... 71
11.6 Drawing order and layers: LAYER ... 71
11.7 Coordinates and units ... 72
11.8 Bundles: BUNDLE ... 73
12 Virtual parts ... 74
12.1 Concept ... 74
12.2 Creating virtual parts ... 74
12.3 Positions relative to a terminal ... 75
12.4 Ports ... 75
12.5 Standard virtual parts ... 75
12.6 No-connect marks ... 76
13 Wires and text ... 77
13.1 Wires: WIRE ... 77
13.2 Junctions ... 79
13.3 Text: TEXT ... 80
13.4 Annotations: graphics in the scene ... 82

13.5 (Not used) ... 83
13.6 Text markup and attribute references ... 83
13.7 Drawing whole nets as wires: ROUTE ... 85
14 Layout intent ... 87
14.1 Common rules ... 87
14.2 ALIGN ... 87
14.3 ORDER ... 88
14.4 DISTRIBUTE ... 88
14.5 DISTANCE ... 88
14.6 Signal flow ... 89
14.7 Symmetry ... 89
14.8 Priority: PRIORITY ... 89
14.9 Orientation relation: ORIENT-RELATION ... 91
15 Electrical semantics ... 92
15.1 Applying semantic definitions ... 92
15.2 Current direction: CURRENT ... 92
16 Files, versions, packages and aliases ... 94
16.1 Versions ... 94
16.2 Packages and USE-PACKAGE ... 95
16.3 The standard package ... 98
16.4 Extensions and unknown content ... 101
16.5 Aliases ... 102
17 Modification model ... 105
Annex A Keyword signatures (normative) ... 107
A.1 The record ... 107
A.2 Header records and packages ... 107
A.3 Definitions ... 108
A.4 Symbol content and graphics ... 108
A.5 Placement and appearance ... 109
A.6 Selections ... 109
A.7 Applying definitions ... 110
A.8 Wires ... 110
A.9 Sets, groups and bundles ... 110
A.10 Constraints and priority ... 110
A.11 Electrical semantics ... 111
A.12 Other standard words ... 111
Annex B Standard sentences (informative) ... 112
B.1 Placement and orientation ... 112
B.2 Layout constraints ... 112
B.3 Appearance ... 113
B.4 Symbols and graphics ... 113
B.5 Wires and text ... 113
B.6 Definitions and applications ... 114
B.7 Selections ... 114
B.8 Sets and other keywords ... 116
B.9 Header record and packages ... 116
Annex C Reserved for later versions (informative) ... 117
Annex D Changes from version 0.2 (informative) ... 119

1 Overview

1.1 Scope

RNL (Rich Netlist) is written in the comments of a host netlist. It describes how the circuit that the netlist defines is understood and presented as a schematic: which symbols are used, where parts go, how they are oriented, aligned, wired, labelled and styled, and what groups of parts mean (Principles 1).

Devices, types, parameters, connections and subcircuit interfaces are defined only by the host netlist. RNL refers to them and cannot change them. RNL does not check the electrical meaning that a drawing suggests (Principles 2.2).

This standard defines:

- how RNL is taken from the comments of a host netlist (chapter 2);
- the lexical structure and the syntax of RNL text (chapters 3 and 4);
- what objects the text describes, how names are resolved, how failures are handled and how the descriptions take effect together (chapters 5–8);
- the vocabulary of each kind of description (chapters 9–17);
- the standard word list, published with this text as `rnl-words-0.3.json` (5.3, Annex A).

This standard does not define a layout algorithm or a unique rendering (Principles 1), and it does not define how a host netlist is parsed (Principles 2.1). The standard package `rnl-std` and alias packages such as `rnl-zh` are written in RNL; this standard defines how packages work (16.2, 16.3, 16.5), and their content is published separately.

This text defines version 0.3 of the language. Versions 0.x make no promise of compatibility with each other (16.1).

1.2 Structure of this standard

The standard consists of an abstract model and a text syntax. The abstract model is objects, facts, ways of consuming input and phases of taking effect (chapters 5–8); the text syntax is its written form (chapters 2–4). Tools exchange the text.

Annex A lists the signatures of the keywords; it is generated from the word list and is normative. Annex B gives the standard sentences, Annex C lists what is reserved for later versions, and Annex D lists the changes from version 0.2; all three are informative. Notes and examples are informative. Everything else is normative.

1.3 At a glance

```
* Five-transistor OTA, input stage
* @RNL(type="header") { @VERSION("0.3")}
.model nch nmos level=1
.subckt ota5t inp inn out vdd vss vbias
M1 x    inp   tail vss nch W=2u L=0.5u
M2 out  inn   tail vss nch W=2u L=0.5u
M5 tail vbias vss  vss nch W=4u L=1u
* @RNL {
*   %dp1 = #M1(in_p) and #M2(in_n) treated @AS $DIFFERENTIAL_PAIR;
*   {#M1 #M2} @ALIGN("y"), i.e., same y;
*   #M5 is at @PLACE(20, 6) => it @ORIENT("R0");
*   %g0 = $GROUND is @ATTACH to &vss;
*   net &vss has a @WIRE connecting {#M5:3 %g0:p}
* }
.ends ota5t
.end
```

- `*` starts a SPICE comment. `@RNL {…}` is a record; inside the braces are members separated by `;` (chapter 2). The header record at the start of the file declares the language version (16.1).
- `#M1`, `&vss` and `#M5:3` refer to an instance, a net and a terminal of the host. `%dp1` and `%g0` are RNL names of this level. `$DIFFERENTIAL_PAIR` and `$GROUND` are general definitions (9.4) from the standard package (16.3). `%g0` is the usage made by applying `$GROUND`; the standard package gives it a ground symbol, so it is drawn as a ground (chapter 12).

- `@AS`, `@ALIGN`, `@PLACE`, `@ORIENT`, `@ATTACH` and `@WIRE` are keywords. Parameters are written in brackets attached to the keyword and hold only literals. AS fills the roles of the definition written after it with the objects written before it; ATTACH adds the definition written before it to the objects written after it (9.4, chapter 12).
- `(in_p)` marks a role. `%dp1 =` names the result of a statement. `=>` passes the result of one segment to the next; followed only by a name, it names the result of the segment before it (5.5).
- `and`, `treated`, `, i.e., same y`, `is at`, `it`, `is … to`, `net`, `has a … connecting` are free text and do not change the meaning (3.3). The standard sentences add such words between the tokens so that statements read like sentences (4.5, Annex B). Without them the record reads:

```
%dp1 = #M1(in_p) #M2(in_n) @AS $DIFFERENTIAL_PAIR;
{#M1 #M2} @ALIGN("y");
#M5 @PLACE(20, 6) => @ORIENT("R0");
%g0 = $GROUND @ATTACH &vss;
&vss @WIRE {#M5:3 %g0:p}
```

1.4 Terms and conventions

1.4.1 Requirement words

- **must**, **must not**: a requirement on documents or implementations.
- **should**, **should not**: a recommendation; there may be reasons to do otherwise, and they should be understood first.
- **may**: something allowed.

The meaning of the language itself is stated in the present tense: "the statement fails", "the later one wins".

1.4.2 Failures and diagnostics

- An item (a field, statement, record or file; 7.6) that **fails** has no effect; a statement that fails has a failed object as its value (7.2). Every failure is an **error**, whatever its cause (7.6).
- A **warning** marks an item that is valid but may not be what the writer meant; it takes effect as usual, except where this standard says otherwise, as for a fact that the solver could not decide within its time limit (8.5). A **hint** marks a common override, hiding or resolution that may also be a mistake. The cases are listed in 7.6; where this standard says that something gives a warning or a hint, implementations should report it.
- When **the interpretation fails**, nothing is drawn and an error is reported (1.5 item 3, 7.6).

1.4.3 Notation

- In the text, keywords are written in upper case without `@`: PLACE, SET-SYMBOL. In syntax and examples they are written with `@`.
- Standard words, names and literals are written in code font: `priority`, `"high"`, `instance`, `$GROUND`.
- Section numbers in brackets refer to this standard: (9.4). "Principles 2.4" refers to section 2.4 of the RNL principles, `rich-netlist.md` (Chinese version: `rich-netlist.zh.md`).
- Examples are written in the standard form (4.3) and without free text, except where free text is the subject. An example marked as an error shows a mistake.

1.4.4 Terms

Term	Meaning
alias	Another spelling of a standard word, or of a `$` name and the role, parameter and local names of a definition, declared with ALIAS or loaded with USE-ALIASES (16.5)
alias package	A package file that only declares aliases; loaded with USE-ALIASES (2.7, 16.5)
annotation	A graphic or a text written in the scene rather than attached to an object (13.4)
apply, application	What AS and ATTACH do: fill the roles of a definition with objects. Each application makes a usage (9.4)

Term	Meaning
attach	A TEXT is attached to the point or object it receives: it can read the object's attributes, and it is placed at or beside the object unless a position is given (13.3). Not to be confused with the keyword ATTACH, which applies a general definition (9.4)
body	The statements in the braces of a definition, or of an inline package (9.1, 16.2)
bundle	An object made with BUNDLE: several host instances drawn as one, in their place. Its terminals stand for the terminals of the same name of all members (11.8)
bus	A wire whose net slot receives a set of nets, even a set of one; the netlist has no buses, a bus is only a way of drawing (13.1)
carrier	A whole-line comment of the host that holds RNL: the comment character (`*`, or `//` in the Spectre language mode) in column 1, followed by nothing, by white space only, or by one space and the content (2.2)
commit	What a record does with its statements: every statement of a record becomes part of its level, named or not (5.4)
constraint	A condition on positions or orientations for the solver: ALIGN, ORDER, DISTRIBUTE, DISTANCE, ORIENT-RELATION (chapter 14)
definition	What DEFINE, DEFINE-SYMBOL or DEFINE-STYLE makes, usually named with `$` (chapter 9). What DEFINE makes is a **general definition**; symbols (DEFINE-SYMBOL) and styles (DEFINE-STYLE) are **specific definitions**. This text says "definition" for a general definition wherever roles, parameters, applications or usages make the kind clear, and "general definition" where it must be told apart from symbols and styles
fact	An assertion carried by a statement, such as "M1 is at (10, 20)" (5.1)
failed object	The value of a statement that failed (7.2)
fill	The grounds, supplies and port tags that rules of the standard package add for terminals and nets that are not drawn otherwise (10.5, 16.3)
free text	Text between the tokens of a statement; it does not change the meaning (3.3)
generated set	The set that a generation rule makes; its single members cannot be referred to (10.5)
group	A set marked with GROUP; a hint to the solver to treat its members as one block, which guarantees no geometric relation (11.5)
handle	The thing a fact is about: an instance, net or terminal of the host, or a point, wire, set or other object made by RNL. The facts about one handle take effect together (5.1, 5.7)
header record	The record `@RNL(type="header") {…}`, the first record of a file: version, aliases, terminal names, packages at the root (2.8, 16.1)
hiding	A definition of a lower level, or of a later package, used in place of one with the same name from above (6.8, 16.2)
host, host netlist	The netlist that holds RNL. The host adapter reads it and provides its comments and objects to RNL (2.1)
house style	The overall way of drawing set by a package or an implementation: which symbols, styles and layout definitions are used, whether connections without wires are shown with labels or flight lines, and so on. A package that sets one is a house-style package. It is not a style in the sense of DEFINE-STYLE
item	What a failure applies to: a field of a record head, a statement, a record, a file, or the whole interpretation (7.6)
instance	An element of the host netlist, including subcircuit instances (X lines). An iterated instance such as `XI<3:0>` stands for several instances (6.11)
key	The name of a text on the object it is attached to: a TEXT with `key="wl"` on M1 is `#M1::wl` (13.3)
keyword	`@` followed by an upper-case word, such as `@PLACE`, or a declared alias (16.5); at most one per segment (4.1, 5.3)
label	A TEXT attached to an object; it can refer to the object's attributes, such as its name or value (13.3)
level	A range of text: the file level of the main file (the top level of the circuit), the file level of an included file, or one subcircuit definition; each package is a level too. Levels form a tree by textual containment (2.6, 6.1)

Term	Meaning
local object	A named object in the body of a general definition; each usage has its own, referred to from outside as `%c1::mid` (9.4)
main object	The object that fills the first role of a usage. When it has a coordinate point (the pin of a terminal, the origin of an instance, a point), a drawn usage is drawn at that point and has no position of its own; otherwise the usage has a position of its own (12.1)
member, segment, statement	A member is a part of a set separated by `;` . A segment is a part of a member separated by `=>` (a name alone after `=>` is not a segment, 4.2). A statement is a segment with a keyword, or an object in a segment without one (4.2, 5.3)
name	`%x =` , `$X =` , `=> %x` or `=> $X` gives an object a name; a name makes no object and is bound once in a level (6.6)
package	A set of definitions and rules placed above a level with USE-PACKAGE; written as a package file (2.7) or inline in the body of USE-PACKAGE (16.2). A package that only references other packages is a **package collection**, such as `rnl-std` (16.3)
packages at the root	Packages placed before all levels: the standard package, used without being written, and the packages of USE-PACKAGE in header records (16.3)
pin	The point of a symbol where a terminal is drawn and wires connect (9.2.4)
priority	The rank of a geometric fact: `"disabled"` , `"low"` , `"normal"` or `"high"` . In a conflict, priority is compared first, then where the fact is written (8.5, 14.8)
record	One `@RNL(…) {…}` in the comments. The fields in its brackets form the record head and apply to every statement of the record (2.3, 2.8)
RNL object	What a statement with a keyword or a `{…}` makes: a handle and facts (5.1)
role	The slot that an object fills in a statement, marked in brackets right after the object, as in `#M1(in_p)` . The roles of a definition are named in its role list and written `?name` in its body; the roles of a built-in keyword are its slot names (5.3, 9.4)
rule	A statement that takes the result of a selection. A rule written in a level is evaluated in that level and once more in each level below, each time with the objects of that level only (10.4)
scene	The objects of a level, from the host and from RNL, with the facts that have taken effect (chapter 8)
selection	The result of a SELECT-… keyword: the objects of one kind in its scope that meet its conditions, evaluated when used (chapter 10)
set	An ordered collection of objects, written `{…}` or found by a selection (5.2)
slot	One of the places where a keyword takes objects, such as the `from` and `to` of DISTANCE. Each slot has a name, which can be written as a role (5.2.1, 5.3)
solver	The tool that computes positions and orientations from the RNL and writes them into a generated record (14.8, chapter 17)
standard form	The one way of writing each keyword that tools output: where the keyword stands, which parameters are given by position, whether roles are marked (4.3, 5.3)
standard package	The package `rnl-std` of definitions, symbols, styles and rules that every file uses unless it says otherwise (16.3)
standard sentence	A statement in the standard form with free text added between the tokens so that it reads like a sentence (4.5, Annex B)
standard word	A word defined by the word list: keywords, slot names, parameter names and values, record head fields, kinds, sub-object names, attribute names, style properties, `true` and `false` (5.3)
style	Only the properties defined with DEFINE-STYLE: colours, line style, line width, text size and so on (9.3). Preferences of layout are not styles
subscript	`<…>` or `[…]` right after a host name or a terminal name, such as `&D<3>` or `&D<7:0>` , standing for one or several names (6.11); not the subscripts of text markup (13.6)
sub-object	A part of an object reached with `::` , such as `#M1::position` or `%dp::in_p` (6.5)
symbol	What DEFINE-SYMBOL makes: graphics, pins and texts, drawn for an instance, bundle or usage (9.2)
take effect	After a level has been read completely, its facts become the scene, phase by phase (chapter 8)

Term	Meaning
token	A keyword, reference, point, bracket, brace, `;`, `=>` or `=`; everything else is free text or comments (3.2)
terminal	A connection of an instance, as on its netlist line: `#M1:d`, or `#M1:1` by number. `#:inp` is the port `inp` of the level (6.4)
usage	The object that one application makes. It records the definition and which objects fill its roles; when it has a symbol, it is drawn (9.4, 9.5)
virtual part	A usage that has a symbol: drawn, but not part of the netlist, such as a ground, a supply, a port tag or a junction dot (chapter 12)
word list	The machine-readable list of all standard words and keyword signatures of a language version, `rnl-words-0.3.json` for this version (5.3, Annex A)

1.5 Core principles

The rules of this standard follow these principles. Items 1–6 refine Principles 2.9; item 7 says how the vocabulary is divided.

1. **Everything is a description.** RNL text is a set of descriptions. The interpreter collects all of them before it gives a result. Nothing is executed, and a name can be used before the statement that binds it.
2. **The place of writing is only ordering data.** Where a description is written (its level and its place in the text) matters in three cases only: when descriptions contradict each other, the later one is kept (geometric facts compare priority first; 8.3, 8.4, 8.5); the drawing order (11.6); and the order of the members of a set (5.2). Otherwise the result does not depend on the order of writing. Each segment of a pipe has a place of its own: a later segment is written later than an earlier one (5.5).
3. **Contradictions.**
   - A description that cannot be understood (a syntax error, a name not found, a wrong kind and so on), or that contradicts itself, fails (Principles 2.4, 2.9; chapter 7).
   - Descriptions that can be understood but contradict each other are resolved where this standard gives a definite way, such as override by key (8.4), the appearance queues (8.3) and the resolution of geometric facts (8.5); a hint may be given.
   - Where there is no definite way yet, the whole interpretation fails (7.6). This version has no such case (7.5). A later version may define a way and make them tolerant; files that work now are not affected.
4. **Each description sees the state before it takes effect.** A description reads the state as it was when its phase began; its own effect and the effects of other descriptions in the same phase are not included (8.2).
5. **Objects and names.** Every statement with a keyword and every `{…}` makes a new object, made of a handle and facts. A reference makes no object. A name only refers to an object and is bound once in a level. Overriding applies to facts, not to names (5.1, 6.6).
6. **No limits for the sake of readability.** A form that is hard to read but has a definite meaning is allowed; it is left to writers to restrain themselves. Only forms without a definite meaning are errors.
7. **Keywords and definitions.** What an implementation must understand itself, and what cannot be written with other descriptions, is a keyword. What can be written with existing descriptions in the body of a general definition is a definition, and its name is only a hint (9.5).

1.6 Conformance

Documents. A host netlist conforms to RNL 0.3 when its records, read as chapters 2–4 define and interpreted as chapters 5–17 define, give no errors (7.6), including errors in the content of the packages it uses. Warnings and hints do not affect conformance. A package file conforms in the same way when it is checked on its own (16.2). A file whose header record declares another language version is not governed by this version (16.1).

Implementations are interpreters (an interpreter used only for checking is also called lint), viewers, solvers, editors and other tools that read or write RNL. A conforming implementation:

1. takes RNL from the comments of the host as chapter 2 defines, and reads it as chapters 3 and 4 define;
2. gives every description the effect that chapters 5–17 define, with the signatures of the word list (5.3);

3. reports every error, with its location and cause (7.6); errors cannot be lowered or switched off;
4. should report the warnings and hints that this standard lists (7.6);
5. may add checks of its own only as warnings or hints: it must not make an item fail that this standard lets take effect, or let an item take effect that this standard makes fail;
6. may offer a stricter mode that stops, or rejects the whole input, but not as the default;
7. lets the user see the diagnostics; how they are presented is up to the implementation (7.6);
8. declares the language versions it supports, and treats a file of a version it does not support as 16.1 defines;
9. treats unknown keywords, parameters and values as errors (16.4);
10. uses the standard package at the root as 16.3 defines;
11. when it writes RNL, writes new statements in the standard form and does not reorder the statements that users wrote (4.3); writes the header record and the version when it saves a file (16.1); keeps unknown and failed statements, the host text, ordinary comments and files of unsupported versions as they are (16.4); and changes only the records that chapter 17 allows it to change.
12. when it solves, resolves conflicts as 8.5 defines, writes back every decision it makes, and passes its results to other tools only in a form that a record of the solver can hold (14.8).

This standard does not define how a solver chooses among positions that meet the constraints, nor how a viewer draws what RNL leaves open; the places where it is left to them are stated in the text (for example 9.2.6, 13.1).

Conformance tests check the errors, with their locations and the extent of each failure, and the resulting scene. Warnings may be checked as optional expectations; hints are not checked (7.6).

2 Embedding in the host

2.1 Two-stage parsing

- The host adapter:
  - recognizes the valid ordinary comments of the host and decides which lines are carriers (2.2);
  - determines which level each comment belongs to, and which file and which include each record and each subcircuit come from (6.1);
  - gives source positions and the order of the text;
  - provides the host objects: instances, nets, terminals, subcircuit ports and model types.
- The RNL core looks for records only in these comments. It must not search the whole netlist for `@RNL` without regard to context: the same characters in host statements or host strings are not RNL.
- The interpreter parses the complete host netlist before it processes the RNL (Principles 2.9).

2.2 Carriers

Records are written in whole-line comments of the host; such comment lines are carriers. The host adapter decides line by line which lines are carriers, removes the comment character and the one space after it, and passes the rest to the RNL core (2.1).

- **Whole-line comments only.** End-of-line comments (`$` , `;` , `//` , `!` and so on) are neither read nor written, and block comments are not used.
  - `@RNL` in an end-of-line comment is only host text, and it gives a warning.
  - Note: whether the end of a line is a comment depends on the dialect, the mode and the options: in HSPICE `$` may follow a number directly (`1k$x`), in the `ps` mode of ngspice `$` is an ordinary character, and Eldo can change it with `.COMCHAR` ; in CDL, `$SUB=` and the like are data. Writing at the end of a line would also change the line of a host statement (Principles 2.1). SPICE and Spectre have no block comments (Spectre only inside `reliability{}`).
- **Comment characters of the hosts.**

Host	Comment character
The SPICE family: SPICE3, ngspice, HSPICE, Xyce, LTspice, PSpice, WRspice, Eldo, and CDL netlists	`*`
Sections in the Spectre language mode: `.scs` files, or text after `simulator lang=spectre`	`//` or `*`
Sections in the SPICE language mode of Spectre, including included files that have no `.scs` suffix and do not start with `simulator lang=spectre`	`*`

  - `*` is a comment in both language modes of Spectre. `//` is documented only for the Spectre language mode, and not by Eldo or CDL.
  - The lines of one record may mix `//` and `*` . A tool that writes a record must follow the records already in the file; when there are none, it writes `*` in the SPICE mode and `//` in the Spectre mode.
  - The carriers belong to the outer form, which does not change (16.1). Support for a new host only adds carriers; existing ones are not changed.
- **Carrier lines.** Every line of a record must meet both of these conditions:
  - The comment character is in column 1, with no white space before it.
  - After the comment character there is either nothing, or only white space, which makes an empty comment line; or one ASCII space followed by the content. So `*@RNL` , `*#M1` and `//@RNL` are not carrier lines.

    Note: HSPICE and PSpice require `*` in column 1, and tools treat indented lines differently. Text glued to the comment character means something to other tools: ngspice and WRspice execute `*#` lines as commands, and WRspice also `*@` lines; `*.` starts a CDL directive (such as `*.PININFO`), `*|` starts DSPF data, and `//pragma` is the encryption envelope of Spectre.
  - A comment line that does not meet these conditions (such as `*#M1 …` , `** …` or an indented `* …`) cuts off a record that is not complete yet: the whole record fails (2.4). Outside a record, `@RNL` in such a line is not an entry, and it gives a warning.

  - A line that holds RNL should not end with `\`. Such a line gives a warning, and tools must not write one. The RNL is read as usual and nothing fails; what is at risk is the host netlist. Note: WRspice joins the next line to such a comment, and so may Spectre run with `-E` (through the C preprocessor) and Eldo with `SLASHCONT`; a host statement on the next line would then be swallowed into the comment.
- **Places where RNL is not read.** `@RNL` in these places gives a warning, and tools must not write there:
  - Line 1 of each host file, and other lines that the host takes as titles (such as the line after `.newjob` in WRspice). Line 1 of the main file is its title, but it is not a title when the file is included; it is never read, so that a file reads the same as the main file and as an included file. So the header record can be written on line 2 at the earliest (2.8).
  - Text after the `.end` of the main file. Note: after `.end`, ngspice reads on, Xyce stops, and HSPICE and PSpice may take the text as the next circuit.
  - Regions that the host does not treat as netlist comments: control scripts (ngspice executes `*#` lines inside `.control … .endc`), blocks and files of other languages (`.verilog` in WRspice, `reliability{}` in Spectre, files brought in with `ahdl_include`), encrypted text, parts removed by preprocessing, and `.lib` sections that are not selected. A region that the adapter cannot classify with certainty is treated as not read, with a warning.
- **Places that are read with a warning:**
  - Comment lines between a statement and its continuation lines (`+` lines). Their meaning for RNL is unique, so they are read; but hosts treat them differently (PSpice reports an error, SPICE 3f4 joins the following `+` line to the comment). They give a warning, and tools must not write there.
  - Encrypted regions written in plain text (`.prot … .unprot`, `protect … unprotect`). They are read, with a warning: once the region is encrypted, the records are gone.
- **Recommendations for writing:**
  - RNL does not limit the length of lines (2.4). Generators should break lines at white space, with at most 132 characters on a line: the limit of older versions of PSpice, the strictest among the common hosts (HSPICE allows 1024, Eldo 2000). An adapter that knows the limit of its host follows the host. When a single token does not fit on a line, the tool must report an error and must not write it (strings cannot continue on the next line, 3.7).
  - For the encoding, see 3.1.

2.3 The entry

- A record starts with `@RNL`. The entry is recognized in upper case only.
- All hosts use the same entry, `@RNL`; it is the sub-tag of Principles 2.1.
- `@RNL` may be written in the middle of a comment; the text before it on the same line belongs to the host comment.
- **The space before `@`.** The character right before the `@` of the entry `@RNL` and of a keyword must be an ASCII space (U+0020); a TAB, a line break or the start of a line does not count. On a carrier line, the space that follows the comment character counts as the space before `@`, as in `* @RNL`. An `@` inside a string, a quoted name or a `/* */` comment is not restricted.
  - So a keyword after `{`, `=>`, `=` or `;` also needs a space before it: `#M1 @SET-SYMBOL { @DEFINE-SYMBOL {…}}`, `%x = @DEFINE(…)`, `#M2 @PLACE(30, 20) => @ORIENT("MY")`.
  - Note: `@` is common in host comments, as in e-mail addresses and `@param`; with this rule such text is not taken for RNL. WRspice also executes lines that start with `*@` (2.2).
  - Outside a record, `x@RNL` is not an entry, only host comment. Inside a record, an `@` without a space before it is neither a keyword nor a character of free text: it is a syntax error, and the member it is in fails (7.4).

2.4 Extent of a record

- A record is written `@RNL(…) {…}`. The record head in brackets may be left out; the braces may not, and a single statement also needs them. The bracket of the record head touches `@RNL`, as the parameter brackets of a keyword touch the keyword (3.5): `@RNL (type="header") {…}` is an error, and the record fails as a whole.
- A record runs from `@RNL` to the `}` that matches its first `{`. Strings, `/* */` comments and the contents of parameter brackets are skipped when the matching `}` is looked for.

- **Records on several lines.** A record may be written on several consecutive comment lines (carriers, 2.2). The comment prefix of each line is removed and the contents are joined; a line break is white space (3.5).
  - An empty comment line, with only `*` (or `*` and white space), is white space, and the record continues.
  - When a line that is not a carrier is met before the closing `}` — a host statement, a line such as `.subckt`, `.ends` or `.include`, an entirely blank line, or a comment line that is not written as a carrier (2.2) — or the file ends, the record is incomplete and fails as a whole. So a record never crosses the boundary of a subcircuit or an include. To leave a line empty inside a record, write a line with only `*`.
  - When an entry `@RNL` appears outside strings and comments before the record has ended, the earlier record is incomplete and fails as a whole, and reading starts again at the new entry. A missing `}` thus loses only one record.
  - An incomplete record is an error, and the diagnostic says that the record is incomplete and where it was cut off. One blank line too many makes a whole record fail; this happens easily and must not pass silently (7.6).
  - Every comment line inside a record is RNL. A line of ordinary remarks in between, such as `* the next line is for M2`, is read as free text, and it is an error if it holds characters that cannot be free text. Remarks inside a record are written in `/* */` (3.4).
  - RNL does not limit the length of lines; the limits of the host are a matter for the adapter. A long statement may be broken into several lines at white space; tokens and strings cannot be split (3.5). Recommendations for generators are in 2.2.
- A token on the same line after a record has ended is an error: these tokens fail, and the record itself takes effect as usual. A new `@RNL` is the exception, so one line may hold several records.
- The outer form of records (the comment carriers, the entry `@RNL`, the record head and the matching of the braces) does not depend on the language version, and later versions do not change it (16.1). The fields of the record head are in 2.8.

2.5 Disabled records and host text

- A comment line outside records that holds no `@RNL` is a host comment and is ignored as a whole.
- **Disabled records.** In a comment line, a `/*` that has only white space or comments before it, or that comes right after a record, starts a comment that runs to `*/`. An `@RNL` inside it is not a record.
- A `/*` in the middle of the text of a comment line, as in `models/*.lib`, is only host text.
- **End of a host comment.** Carriers are whole-line comments (2.2): a host comment ends at the end of its line, and nothing inside the line can end it earlier, so the end of a host comment does not conflict with RNL text.

2.6 Levels

- Levels are divided by the text; their hierarchy is in 6.1:
  - The file level of the main file: what is written in the main file outside every `.subckt … .ends`. It is the top level of the circuit.
  - The file level of each included file. The sections of `.lib` are treated the same way.
  - Each `.subckt … .ends`.
- A record belongs to the level it is written in (Principles 2.5).
- **Order across files.** An included file is a level of its own, below the level that includes it (6.1). So where the include line is written does not affect the order, unlike USE-PACKAGE (16.2), and files included side by side do not affect each other. Within one level, the order is the order of the text.
- **scope.** A record can name a target level with `@RNL(scope="name")`. The name is the name of a subcircuit and means the definition of that subcircuit. In the following file, the last record is written in the main file and applies to the subcircuit `ota`:

```
* A record written outside its subcircuit
* @RNL(type="header") { @VERSION("0.3")}
.model nch nmos level=1
.subckt ota inp inn out tail vss
M1 x inp tail vss nch
M2 out inn tail vss nch
```

```
.ends ota
X1 vip vin vo vt 0 ota
* @RNL(scope="ota") { #M1 @PLACE(10, 20)}
.end
```

- Each subcircuit has only one schematic (6.1), so scope names the definition, not an instance.
- scope only goes down: the target must be a subcircuit that this level contains in its text, that is, one defined in this file, or in a file that this level includes directly or through further includes. scope is not written inside a subcircuit: the RNL of a subcircuit covers only itself. A `scope` written in a record inside a subcircuit is an error and is treated as not written (2.8). When the target is not found or is not below this level, it is an error, and the statements of the record, which have no level to be resolved in, all fail (chapter 7).
- All references in the record are resolved in the target level. Its statements are committed to the target level, and the names they bind are published there and shared with the target level's own records (6.7); binding a name that the target level already binds to another object is an error, and binding it again to the same object is allowed (6.6). scope affects only its own record and does not change the host netlist.
- Order: the statements that scope brings into a subcircuit come after its own text; when several places scope into the same subcircuit, the outer ones come later (8.3, 8.5). What is explicitly written outside overrides what is written inside, just as the priority of an outer usage overrides its body (14.8).
- The output of the solver and the changes that users make in an editor must not use scope; they are written into the subcircuit's own text (chapter 17). scope is only for statements that authors choose to write outside.

2.7 Package files

A package can be written as a package file of its own, which USE-PACKAGE finds (16.2). A package file is a pure RNL file; its suggested extension is `.rnl`, and it is written in UTF-8. It is not a netlist and is not loaded with `.include`.

- A package file is **read** like the carriers of a netlist (2.2, 2.4), except that every line is a carrier, and the comment character and the space after it are both empty:
  - an `@` at the start of a line counts as having a space before it (2.3), so `@RNL` and keywords may start in column 1;
  - an empty line is an empty comment line, and a record continues over it;
  - text outside records is remarks and is ignored, like host comments (2.5); `/* */` works as usual;
  - line 1 is read: a package file is not a netlist and has no title line (2.2).
- It holds `@RNL(…) {…}` records, parsed in the order of 16.1. The first record is the header record, which holds VERSION and PACKAGE. For what a package may hold, including references to other packages, see 16.2; alias packages are package files too (16.5).

The full text of a package file follows, without comment prefixes. The third line is outside the records and is a remark:

```
@RNL(type="header") { @VERSION("0.3"); @PACKAGE(name="acme-style", version="1.2")}

ACME house style: NMOS symbol and its rule.
@RNL {
  $NMOS_ACME = @DEFINE-SYMBOL {
    $NMOS;
    ? @TEXT("${name}", key="name") => @PLACE(-2.5, 1.5)
  };
  # @SELECT-INSTANCE(type="nmos") => @SET-SYMBOL $NMOS_ACME
}
```

2.8 Record head

The fields in the brackets of `@RNL(…)` apply to every statement of the record, and all of them are given by name. Field names and the values of `type` are written in the standard spelling; they are case-sensitive and cannot have aliases (3.6, 16.5).

```
* @RNL(type="header") { @VERSION("0.3")}    /* the first record of the file: read as version 0.3
(16.1) */
* @RNL { %dp1 = #M1(in_p) #M2(in_n) @AS $DIFFERENTIAL_PAIR}    /* no type: "authored" */
```

```
* @RNL(type="generated", priority="high", author="rnl-solver 0.3", hash="sig1:9f3a62c1") {
*   #M1 @PLACE(12, 20); #M2 @PLACE(28, 20)
* }
* @RNL(author="rnl-editor 0.1") { #M2 @PLACE(x=30, priority="high")}
```

Field	Meaning	Affects the interpretation
`type`	The kind of record: `"authored"` (default), `"generated"`, `"header"`	yes
`scope`	The target subcircuit (2.6)	yes
`priority`	The default priority of the statements in the record; to AS and ATTACH only `"disabled"` (14.8)	yes
`author`	The author, the tool and its version, the version of a package and so on; any string	no
`hash`	A hash of the input of the solver (chapter 17)	no
`numbering`	The check of the numbers that the record uses (5.9); only with `type="generated"`	yes
`modified`	The time of the last change, an ISO 8601 string	no

- **type**: the values are fixed, and each has a defined behaviour.
  - `"authored"`, the default: written by hand, by an editor or by an import.
  - `"generated"`: the result of the solver. When the solver solves, the level is interpreted without the records with `type="generated"`: they add no objects, names or facts, and do not constrain the result. How else the solver uses them is up to it. Afterwards it replaces them as a whole, whoever their author is. Wires that an editor redraws while a device is dragged are solver results too (14.8, chapter 17). `type` implies no priority: the solver writes `priority="high"` explicitly.
  - `"header"`: the settings of the file (16.1). The header record may be left out. When it is written, it is at the file level, not inside `.subckt`, and has no `scope`; it is the first record of the file, and there is at most one. It holds VERSION (16.1), the aliases of ALIAS and the alias packages of USE-ALIASES (16.5), the terminal names of DEVICE (6.12) and the packages at the root of USE-PACKAGE (16.3); a package file also writes PACKAGE there (2.7, 16.2). The header record itself is read in the standard spelling, without aliases (16.5). A header record written later in the file, or a second one, is an error and is ignored as a whole; the records already read are not revisited. A section of `.lib` that is read counts as a file (2.6): its header record is the first record in the section.
- **Metadata**: `author`, `hash` and `modified` do not affect the interpretation; the drawing is the same when they are deleted. Tools use them to decide how to treat a record (chapter 17). Neither `type` nor `author` affects the order of resolution (8.3).
- **Errors**:
  - An unknown field gives only a warning and is ignored; it does not affect the record or the other fields. This leaves room for later extensions. When the warning is given, it must name the closest field name. Note: as a consequence, a misspelled field that would affect the interpretation (such as `priorty="low"`) gets only a warning, and the record takes effect with priority `"normal"`; fields of that kind added later rely on the version (16.1) to tell old tools that they are out of date.
  - A known field written wrong (a wrong value, `numbering` in a record whose `type` is not `"generated"` or that has `scope`, or the field written twice) is an error, and the field is treated as not written: `priority` falls back to `"normal"`, `type` to `"authored"`, and metadata is dropped. The exception is a `scope` whose target is not found: the statements of the record have no level to be resolved in, and all fail (2.6). A field given by position is an error and is ignored.
- Records have no identifiers. Tools find their own records by `type`, `author` and the keys of the statements (5.8); an object is referred to from another record by its name, or, from the record of the solver, by its number (5.9). Other remarks are written in comments.
- A solver result is not a proposal that has to be adopted: it takes effect directly, and a stale result only gives a hint (chapter 17).

3 Lexical structure

3.1 Characters

- Tokens are made of ASCII characters.
- Non-ASCII characters appear only in strings (including quoted names), free text, comments and aliases, and in unquoted role and parameter names. Outside free text and comments, an unquoted non-ASCII word appears only after `@` and inside brackets: after `@` it must be a declared alias; inside brackets it is an alias, or a role name or parameter name itself (3.6, 16.5).
- **Encoding.** The host adapter decodes the text by the rules of the host; RNL works on characters. A tool that writes RNL must not change the encoding or the line endings of the host file and must not add a byte order mark; it must not write a character that the encoding of the host cannot represent, and gives a warning instead. New files should use UTF-8.
- **Non-ASCII characters in host comments.** Adapters must state the hosts for which non-ASCII characters in comments have been verified; for other hosts, a non-ASCII character gives a hint.

  Note: support differs between hosts. ngspice (from version 42), WRspice (except in the title), Xyce and recent versions of LTspice accept non-ASCII characters in comments; the documentation of HSPICE, PSpice and Eldo does not say; the Spectre manual describes netlists as ASCII files; QSPICE uses Windows-1252. Another reason for not adding a byte order mark: older versions of ngspice join it to the first token of an included file, and WRspice stops reading when line 1 begins with one.

3.2 Tokens

Token	Form
Keyword	`@` followed by an upper-case name, such as `@PLACE`; or a declared alias (16.5)
Reference	A prefix followed by a name, possibly with terminals and sub-objects, such as `#M1`, `#M1:d`, `%dp1::in_p` (chapter 6). A host name or a terminal name may be followed directly by a subscript, such as `&D<3>`, `&D<7:0>` (6.11). In a record of the solver, also `%#` followed by an unsigned integer, a number, such as `%#3` (5.9)
Point	`(x, y)`, not touching a keyword or an object: after white space, a comment, `{`, `;`, `=>` or `=` (3.8)
Brackets	The `(` `)` of parameters and roles, touching the keyword or the object (3.5)
Braces	The `{` `}` of sets
Separators	`;` (separates members), `=>` (pipes and naming), `=` (naming at the start of a member)

Inside brackets (the `()` of parameters, roles and points, and the `[]` of arrays) there are also strings, numbers, `,`, `=`, `_` and names (parameter names, role names, `true`, `false`), and, in the body of a general definition, parameter placeholders `?name` (6.10). Inside the `<>` and `[]` of subscripts there are only unsigned integers, `,` and `:` (6.11).

3.3 Free text

- Outside brackets, any character can be free text except those listed below. This includes all non-ASCII characters (such as the Chinese punctuation `：`, `、`, `；`, `（`, `）`) and `-`, `:`, `'`, `!`, `+`, `/`, `*` and `|`. The exceptions are:
  - `@`, `#`, `&`, `%`, `$`, `?`, `"`, `;`, `=`, `{`, `}`, `(`, `)`: outside brackets they start or belong to tokens (3.2). `?` starts a placeholder (6.10); in a symbol body, a lone `?` refers to the object drawn with the symbol (13.3);
  - `[`, `]`, `<`, `>`: the brackets of arrays and subscripts (3.7, 6.11);
  - `/*`: it starts a comment (3.4).
- Rules of position keep the meaning unique:
  - A `:` or `::` directly after a reference and followed by a name or a quote is a terminal or a sub-object, as usual (6.4, 6.5). When no name follows, the `:` is free text, as in `Input pair: %dp1 = …` and `#M1: the left one`.
  - A `<` or `[` directly after a name is a subscript (6.11).
  - In the name of a keyword, a `-` belongs to the name only when a letter follows it (keywords: 3.6). `@GROUP- the pair` is GROUP followed by free text.
- Free text may be written between any two tokens outside brackets, and at the start of a member.

- **Free text is part of the statement.** A reference written in it is not an explanation but an object of the statement: `Unlike #M2, #M1 @PLACE(x=0)` puts M2 at x = 0 too. Text meant only as an explanation is written in a comment (3.4), or as a member of its own, separated from the next member by `;`: a member with only free text is an empty member and is ignored (4.1).
- The standard does not check free text, and free text does not affect the meaning (Principles 2.8): `do not move #M2 @PLACE(x=30)` still moves M2. An implementation may give a warning of its own for words such as negations (7.6).
- In the same way, in a mistaken `#M1.d` the `.d` is free text, and the reference is M1 itself, not its drain (a warning is given, see below).
- There is no free text inside brackets.
- **Warnings.** The following mistakes could pass silently; they give a warning (7.6):
  - A word in free text that is the name of a host instance of this level, a visible RNL name (without its prefix) or the name of a keyword, such as `M3` in `#M1 #M2 M3 @ALIGN("y")` or `WIRE` in `WIRE {…}`: a `#`, `%`, `$` or `@` is probably missing. Net names are often ordinary words (such as in and out) and are not checked.
  - A reference directly followed by `.` and a name, such as `#M1.d`: terminals are written with `:`.
  - A reference directly followed by `-`, `!`, `'`, `+`, `/`, `*`, `\`, `^`, `~` or `|`, such as `&vdd!` or `#X1-a`: the name probably contains this character and needs quotes, as in `&"vdd!"` (3.6); without them, the reference silently means `vdd` or `X1`.
  - `|` in free text: a pipe is written `=>` (4.2 item 3).
  - `:` or `::` directly after a reference and followed by a non-ASCII word, such as `%c1::中点`: a name with non-ASCII characters after `::` needs quotes, as in `%c1::"中点"` (3.6); without them, the `::` and the word are free text, and the reference means `%c1`.
- Most other mistakes of this sort are errors already: there is no free text in brackets, so `@ORIENT(MY)` is an error; `#M1 @PLACE 1 2` does not match the signature; `#M1 (in_p)` is an error, since a bracket after white space starts a point and `(in_p)` is not a point (3.5); and the `>` in `R1 -> R2` cannot be free text.
- Writing guides for authors are generated from the word list (5.3). Records written by solvers and editors must not contain free text.

3.4 Comments

`/* … */` may be written wherever white space is allowed, including inside brackets. Comments do not nest, and any character may be written in them. An explanation that contains references, or characters such as `@`, `#` and `;`, is written in a comment (3.3).

3.5 White space and adjacency

- White space is the space, the TAB and the line break. Tokens cannot be split across lines, and strings cannot span lines.
- White space and comments are not allowed: between a prefix and its name, or between `%#` and its number; on either side of `:` and `::`; between a name and its subscript (6.11); between a keyword and its parameter brackets, and between `@RNL` and the bracket of its record head (2.4); between an object and its role brackets.
- A `(` after white space or a comment starts a point (3.8). So `#M1 @PLACE (10, 20)` is PLACE followed by a point, and PLACE has no parameters: an error. `#M1 (in_p)` is M1 followed by a point that is not written correctly: an error as well. The diagnostic suggests removing the white space.
- A `<` or `[` directly after a name, quoted or not, always starts a subscript (6.11). Points are written `(x, y)`, which does not conflict with subscripts; to write a point after a name, separate them with white space, as in `&D (1, 2)`. White space and comments may be written inside a subscript, as in `&D<1, 2:3>` (6.11).
- A `.` directly after a keyword is free text, such as the full stop in `{#M1 #M2} @GROUP.`; `@BUNDLE.LOCK` is BUNDLE followed by the free text `.LOCK` (16.4).

3.6 Names

- **Unquoted names** are one or more ASCII letters, digits and underscores, taken by longest match.

- **Host names** (after `#` and `&`) and terminal names follow the rules of the host dialect (Principles 2.1): which names are valid, whether case matters and whether a name may start with a digit, as in `&0`. SPICE is not case-sensitive; Spectre is. A name with other characters is written in quotes, such as `#"M1<0>"`, `&"vdd!"`, `#XU1:"in+"`; the text in quotes is written as in strings (3.7). A name may be followed directly by a subscript: `#M1<0>` is `#"M1<0>"`, and `&D<7:0>` is a list of nets (6.11). A `<` or `[` inside quotes is part of the name and is not parsed.
- **RNL names** (the names after `%`, `$` and `?`, role names, and the parameter names of general definitions):
  - They are case-sensitive: `$DIFFERENTIAL_PAIR` and `$differential_pair` are two names.
  - After `%`, `$` and `?` a name may start with a digit, as in `%1st`. In the places without a prefix (`(role)` and `parameter=`), an unquoted name consists of ASCII letters, digits, underscores and any non-ASCII characters, and does not start with an ASCII digit, so that it does not look like a number: `#M1(正输入)`, `@AS(间距=3) $X`. Both places are inside brackets, where there is no free text (3.3), so no other word is taken into the name.
  - They may be quoted, with the same meaning: `$"a"` is `$a`; quotes only escape. Names with other characters, and non-ASCII names, are written in quotes, such as `%"差分对"`, `%"input pair"`, `?"正输入"`, `%dp::"正输入"`. In role and parameter-name places, non-ASCII names may be unquoted (the item above), but a space or other ASCII punctuation still needs quotes, as in `#M1("input pair")`. In role and parameter-name places, a quoted name is always the name itself, never an alias. This does not apply to `$` names and names after `::`, whose non-ASCII aliases are written in quotes anyway (16.5).
  - They are compared character by character as written, without Unicode normalization. An empty name is an error.
  - They take no subscript (6.11): a `<` or `[` directly after them is an error, as in `%p<1, 2>`. A point is written `(1, 2)`, separated from the name by white space.
- **Aliases** (16.5) are declared in the header record or come from alias packages. Aliases of keywords and standard words may contain non-ASCII letters; they are written without quotes only after `@` and inside brackets. An alias of a `$` name is written like a `$` name, in quotes when it is non-ASCII, as in `$"接地补位集"`. Unquoted names after `#`, `&`, `%`, `$`, `?` and `::` are ASCII only, as above, so `#M1和#M2` is `#M1`, the free text `和` and `#M2`.
- **Keywords** are upper case only, with words joined by `-`, such as `@AS` and `@SET-SYMBOL`; upper case also makes them easy to find in free text. `@at` is an unknown keyword, unless the header record declares it as an alias (16.5).
- **Standard words** are case-sensitive and written as the standard spells them: parameter names, the field names of the record head, sub-object names, the names of selection conditions and enumeration values (such as `"x"`, `"low"`, `"MY"`). Values that name things of the host (names in selection conditions, model names, net names) follow the rules of the host. Other spellings can be accepted through aliases (16.5).
- **`true` and `false`** are reserved and lower case only. They cannot be role names or parameter names.

3.7 Literals

- **Numbers** are decimal. They may have a sign and leading zeros, may leave out the zero before the decimal point or the digits after it, and may have an exponent: `0`, `+1`, `01`, `.5`, `1.`, `-2.5E-4`. `01` is 1, not octal. There are no units or engineering suffixes (`2u` and `10k` are errors), no hexadecimal numbers, no `NaN` and no `Infinity`. The number of digits is not limited. When an implementation cannot hold a value, the statement fails in that implementation, with an error that says it is a limit of the implementation; the value is not changed silently. Coordinates are compared with a tolerance (11.7).
- **Strings** are written in double quotes and cannot span lines. The characters between the quotes are taken as they are; only `"` itself is written `""`, as in `"say ""hi"""`. There are no backslash escapes: `\` is an ordinary character, and there is no `\n` or `\uXXXX`; non-ASCII characters are written directly. A line break cannot be written in a string; a line break in a text is written with the markup `\\` (13.6). TEXT has backslash commands of its own (13.6); they are how TEXT reads its text, not escapes of the string. Strings in the record head are written the same way; this belongs to the outer form and does not change (16.1).
- **Arrays** are written `[value, value, …]`. Their elements are literals, and arrays may be nested.
- **Truth values** are `true` and `false`.
- There are no other literals. Enumerations are written as strings, such as `axis="x"`; there is no `null`, no map and no constructed value.
- There are no expressions: operators are errors.

3.8 Points

- A point is written `(x, y)`, where x and y are numbers or `_` (in the body of a general definition, also a parameter placeholder, 6.10). `_` leaves that axis free, to be decided by constraints or the solver.
- Writing a point creates a point object (5.1).
- There is no arithmetic on points; relative positions are written as constraints (chapter 14).
- The `(` of a point has white space or a comment before it, or comes right after `{`, `;`, `=>` or `=`. Brackets that touch a keyword or an object hold its parameters or its role (3.5). A `<` directly after a name is a subscript in any case (6.11).

4 Syntax

4.1 Grammar

The grammar of a record is given in EBNF. Each production is written `name = expression ;`; `|` separates alternatives, `[ … ]` is optional, `{ … }` is repeated zero or more times, `( … )` groups, text in double quotes is written literally, and `(* … *)` is a comment.

```
record         = "@RNL" [ parameters ] set ;          (* the bracket touches "@RNL" *)
set            = "{" [ member ] { ";" [ member ] } "}" ;
member         = [ front-naming ] segment { naming } { "=>" segment { naming } } ;
front-naming   = ( "%" rnl-name | "$" rnl-name ) "=" ;
naming         = "=>" ( "%" rnl-name | "$" rnl-name ) ;
segment        = argument { argument } [ call { argument } ]
               | call { argument } ;
call           = keyword [ parameters ] ;          (* the bracket touches the keyword *)
keyword        = "@" keyword-name ;                (* upper case, or a declared alias *)
argument       = object [ "(" role-label ")" ] ;   (* the bracket touches the object *)
object         = reference | placeholder | point | set ;
reference      = "#" [ host-name [ subscript ] ] path
               | "&" host-name [ subscript ] path
               | "%" rnl-name path
               | "%#" integer path               (* a number, in a record of the solver, 5.9 *)
               | "$" rnl-name ;
placeholder    = "?" rnl-name path
               | "?" ;                            (* alone, in a symbol body *)
path           = { ":" terminal-name [ subscript ] | "::" rnl-name } ;
subscript      = "<" subscript-item { "," subscript-item } ">"
               | "[" subscript-item { "," subscript-item } "]" ;
subscript-item = integer [ ":" integer [ ":" integer ] ] ;   (* unsigned, 6.11 *)
host-name      = name | string ;
rnl-name       = name | string ;
terminal-name  = name | string ;
role-label     = bracket-name | string ;
point          = "(" coordinate "," coordinate ")" ;   (* after white space, a comment, "{", ";", "=>" or
"=", 3.8 *)
coordinate     = number | "_" | parameter-placeholder ;
parameters     = "(" [ value { "," value } [ "," named-value { "," named-value } ]
                     | named-value { "," named-value } ] ")" ;
named-value    = parameter-name "=" value ;
parameter-name = bracket-name | string ;
value          = number | string | "true" | "false"
               | "[" [ value { "," value } ] "]"
               | parameter-placeholder ;
parameter-placeholder = "?" rnl-name ;
```

- `name`, `bracket-name`, `keyword-name`, `string`, `number` and `integer` are written as chapter 3 defines. `name` is an unquoted name (3.6). `bracket-name` is an unquoted name in the brackets of a role or before the `=` of a parameter: ASCII letters, digits, `_` and any non-ASCII characters, not starting with an ASCII digit (3.6). `keyword-name` is made of letters, digits and `_`, with words joined by `-`; a `-` belongs to it only when a letter follows (3.3). A standard keyword is upper case (3.6); an alias is made of letters of any script, digits and `_`, without `-` (16.5). `string` and `number` are in 3.7; `integer` is an unsigned decimal integer.
- `rnl-name`, `host-name`, `terminal-name`, `role-label` and `parameter-name` may also be quoted, with the same meaning, except that in role and parameter-name places a quoted name is never an alias (3.6). Keywords, parameter names, slot names in role brackets, sub-object names, `true` and `false` may be written as declared aliases; unquoted non-ASCII aliases are used only after `@` and inside brackets, and an alias after `::` is quoted (16.5).
- A subscript follows a host name or a terminal name directly, with no white space between them; white space and comments may be written inside a subscript (3.5, 6.11). Which names take a subscript: 4.2 item 10.
- The parameter brackets touch the keyword, and the role brackets touch the object; a `(` after white space or a comment starts a point (3.5).
- `=>` is a pipe or a naming, depending on what follows it (4.2 item 3).

- A placeholder with a name is written only in the body of a general definition (6.10). In an object place it is a role and may have a path; in a place that takes a literal (a parameter value, an array element, a point coordinate) it is a parameter of the definition and has no path (6.10, 9.4). The lone `?` is written only in a symbol body (13.3).
- Free text and comments may be written between any two tokens outside brackets and at the start of a member (3.3, 3.4); white space and comments may also be written inside brackets. The grammar does not show them.
- An empty member (as in `;;`) is ignored; the `;` after the last member may be left out.

4.2 Constraints outside the grammar

1. A segment is not empty.
2. A segment has at most one keyword. `@RNL` appears only at the start of a record.
3. A `=>` that is followed by a `%` or `$` name, after which comes `=>`, `;`, `}` or the end of the member, is a naming. Any other `=>` starts the next segment; that segment has exactly one keyword, and a name written there is an object of the segment. The first segment of a pipe may have no keyword; then it has exactly one object. So to name a result and continue the pipe, write `=> %g0 => @PLACE(1, 2)`. In `=> %g0 @PLACE(1, 2)`, `%g0` is an object of PLACE; usually it is not bound, and the statement fails because `%g0` is not found. The diagnostic may ask whether a `=>` is missing.
4. In a segment without a keyword, each argument is a statement of its own: `#M1 #M2` in `{#M1 #M2}` is two statements. Such a segment, when it is named (`%p =` or `=> %p`), has only one object: write `%p = {#M1 #M2}`, not `%p = #M1 #M2`.
5. Roles are marked only on the arguments of a segment that has a keyword. The segments after a pipe mark no roles.
6. In a segment, either all arguments are marked with roles or none are. The same role is not marked twice.
7. For the keywords that take a definition, the keyword fixes the slot of the definition: for ATTACH it is the first object written; for AS, SET-SYMBOL, SET-STYLE, SELECT-USAGE and SELECT-ANCESTOR-USAGE it is the last object. SELECT-USAGE may leave out its definition: its last object fills the definition slot only when it is a definition (10.1). The definition is not marked with a role and does not count for rule 6 (5.6).
8. There are two ways of naming, and they may be used together. `name =` can only be the first token of a member (free text may come before it), at most once in a member. `=> name` follows a segment, at the end of the member or before the `=>` of the next segment, and a member may have several. Rule 3 tells it apart from a pipe.
9. The parameters given by position come first, then those given by name: after one `name=value`, all the following parameters are given by name. Only the parameters for which the signature lists a position order can be given by position, in that order. No parameter is given twice.
10. A subscript follows only a host name after `#` or `&`, or a terminal name or port name in a `#` reference. RNL names, the terminal names of usages (role names) and sub-object names take no subscript (6.11).
11. One reference has at most one subscript that expands to a list of names (6.11).

4.3 Word order

- The keyword, with its parameters, may stand anywhere among the objects; the order of the objects matters. `#M1 #M2 @DISTANCE(…)`, `#M1 @DISTANCE(…) #M2` and `@DISTANCE(…) #M1 #M2` are the same.
- Each keyword has one standard form, and tools must write only that form. The standard form is recorded in the word list (5.3): where the keyword stands, which parameters are given by position and which by name, and whether roles are marked.
- By default, the standard form is: the objects first, then the keyword with its parameters, then the definition after the keyword. ATTACH writes its definition first, then the keyword, then the objects: `$GROUND @ATTACH &vss`. The parameters for which the signature lists a position order are given by position, first; the others are given by name, in the order of the word list: `@PLACE(10, 20)`, `@ORIENT("R90")`, `@DISTANCE("bottom-to-top", min=4)`, `@TEXT("W/L", key="wl")`. PLACE gives x and y by position only when both are written; one axis is written by name, as in `@PLACE(x=10)`. Parameters that can be removed without changing the meaning are not written, such as `mode="auto"` of WIRE. Built-in keywords are written without roles; AS and ATTACH mark all roles when the general definition has two or more roles.

- Layout: one space after each comma in brackets; no space around the `=` of a parameter; one space on each side of the `=` of a naming and of `=>`; the keyword touches its bracket; a space between `{` and a keyword, as in `{ @…` (2.3); numbers are written as the shortest decimal that reads back as the same value.
- **Standard form of naming.** A member without a pipe is named at the start, `%x = statement`; a member with a pipe is named at the end, `statement => … => %x`, with the name after the segment it names. So a name at the start always names a whole member, and readers need not guess which segment of a pipe it names. The syntax accepts both forms.
- "Only that form" applies to the statements that a tool writes anew. When a tool rewrites existing records, it must not reorder the statements that users wrote (16.4); it brings them to the standard form only when the user asks for formatting.
- Standard sentences are in 4.5.

4.4 Examples of errors

Each of the following is an error; the comment says why.

Error example

```
#M1 @PLACE (10, 20)    /* a bracket after white space is a point, and PLACE has no parameters (3.5) */
```

Error example

```
#M1 (in_p) #M2 (in_n) @AS $DIFFERENTIAL_PAIR    /* role brackets touch the object; a separate bracket is a point (3.5) */
```

Error example

```
#M2 @PLACE(62, 40) (MY)    /* the second bracket is a point, and its content is not a point */
```

Error example

```
&vss @ATTACH $GROUND    /* the definition of ATTACH is written first: $GROUND @ATTACH &vss */
```

Error example

```
R1 -> R2 {#R1 #R2} @ORDER("x")    /* ">" cannot be free text (3.3) */
```

Error example

```
#M1 @PLACE(1, 2), #M2 @PLACE(3, 4)    /* a comma does not separate statements: two keywords in one segment */
```

Error example

```
#M1 #M2 @SET-SYMBOL $NMOS    /* three objects, and SET-SYMBOL has two slots (5.2.1); write {#M1 #M2} @SET-SYMBOL $NMOS */
```

Error example

```
#M1 => #M2    /* the segment after a pipe has no keyword */
```

Error example

```
#M1 #M2 #M3 @DISTANCE("x", min=1)    /* three objects for two slots */
```

Error example

```
#M1 @PLACE(x=10, 20)    /* a parameter given by position after one given by name (4.2 item 9) */
```

Error example

```
{#M1 #M2} @ORDER(axis="x", direction="increasing")
/* ORDER has no axis: the axis and the direction are one parameter, direction, as in @ORDER("left-to-right") (14.3) */
```

Error example

```
#M1 @SET-SYMBOL($NMOS)    /* brackets hold only literals */
```

Error example

```
{#M1 #M2} @ALIGN(axis=y)    /* enumerations are written as strings */
```

Error example

```
%ax = (_, _);
#M1(a) #M2 %ax @AS $MIRRORED    /* roles are marked on all objects or on none */
```

Error example

```
$NMOS(symbol) #M1(target) @SET-SYMBOL    /* the definition is written last and is not marked with a role */
```

Error example

```
#R1(source) #R2(chain) #R3(chain) #R4(load) @AS $CHAIN    /* the same role is marked twice */
```

Error example

```
(0, 0) @PIN(terminal=1)    /* a pin cannot be written in the scene (13.4) */
```

Error example

```
#M1 @CURRENT("in")    /* CURRENT takes only terminals (15.2) */
```

Error example

```
&D<1:0>[1:0]    /* a name has at most one subscript (6.11) */
```

Error example

```
#XI0<1:0>:D<1:0>    /* one reference has at most one expansion (6.11) */
```

Error example

```
&D<_, 1>    /* a "<" directly after a name starts a subscript, and its content is not a subscript (3.5) */
```

Error example

```
%p<1, 2>    /* RNL names take no subscript; a point is written (1, 2) (6.11) */
```

4.5 Standard sentences

A standard sentence is a statement in the standard form with free text added between its tokens, so that it reads like a sentence. Standard sentences are used in writing guides and by formatting tools that add the words to statements or remove them. Free text does not affect the meaning (3.3): the sentences are only suggestions, and users and language models may write them, write none, or write other words.

- **Facts.** A statement says what holds in the drawing, not an operation to perform. The objects come first as the subject, the keyword serves as the verb or a noun, and the parameters are its values. Only linking words are added, such as `is`, `are`, `has`, `is at` and `there is` in English, and `是`, `处于`, `位于`, `有`, `的`, `按……排列` in Chinese. A few statements may read as operations, such as `#M1 @SET-SYMBOL to $NMOS`.

- **Words follow from the statement.** The words that a tool generates depend only on the keyword, the parameters, the number and kinds of the objects and the level the statement is written in, not on the meaning of the circuit. For example, `is` or `are` follows the number of objects; `the x of` (`x 坐标`) is written when only one axis is given; the wording of the candidates of a selection follows the scope and the kind selected (Annex B). This is how sentences are generated; it does not restrict users.
- **No words that seem to mean something the statement does not.** Words such as `must`, `required` and `only` (`必须`) are not written, and a word of direction is written only when the parameter has that direction. Readers believe the words in a sentence, and writing guides should warn them about this.
- **Glosses.** Readers may not know orientation codes and axes. After the keyword, `, i.e., …` (`，即……`) may say what the code itself means, as in `#M2 @ORIENT("MX"), i.e., flipped upside down` and `#M2 @朝向("MX")，即上下镜像`.
- **ORIENT-RELATION and DISTANCE**, which have two slots, read `from A to B` (`A 到 B`); in Chinese, ORIENT-RELATION takes `的` and DISTANCE takes `有`. A DISTANCE whose direction is only an axis (`"x"` or `"y"`) reads `between A and B` (`A 和 B 之间有`).
- **Selections** always read as choosing among candidates: `among <candidates>, @SELECT-…` in English, `从<候选>中 @选出…` in Chinese. When the scope is `#`, the candidates are `the <kind> at each level in # and below` (本层 # 及以下每一层的<种类>), which says that the rule is evaluated in each level (10.4). When the kind cannot be worked out, the sentence reads `for X,` (对于 X，).
- **AS and ATTACH.** AS is used when the objects together make up the thing: `#M1(in_p) and #M2(in_n) are treated @AS $DIFFERENTIAL_PAIR`. ATTACH is used when the thing is added to the objects: `$GROUND is @ATTACH to &vss` (9.4).
- **Pipes.** The next segment refers to the result of the one before it. English uses `it` (after AS, ATTACH or a single object) or `they` (after a selection); Chinese always uses `其`, and `其与` when other objects are listed with it. A name in a pipe is preceded by `named` (`记作`), as in `… => named %pout` (`… => 记作 %pout`); it is still a naming (4.2 item 3).
- **Naming.** The name of a single statement is written at the start (4.3). When an AS statement is named, English leaves out `are`: `%dp = #M1(in_p) and #M2(in_n) treated @AS $DIFFERENTIAL_PAIR`.
- **Names of sets.** A name that refers to a set should end in `_SET` for `$` names and `_set` for `%` names (`集` in Chinese), such as `$GROUND_FILL_SET`, `%bias_set` and `%"接地补位集"`; the standard package follows this.
- **No `each`.** Words such as `each` (`每一个`) are not written. Whether a set is applied as a whole or member by member depends on the role list of the definition, and the standard sentences do not say which.
- **Chinese.** Chinese sentences use the aliases of the alias package `rnl-zh` (16.5). After a Chinese comma, a space is still needed before `@` (2.3).

The standard sentence of each keyword is listed in Annex B.

5 Objects and statements

5.1 Objects

- Every statement with a keyword makes a new RNL object, made of a handle and facts. A `{…}` also makes a new object, a set (5.2).
- A reference (`#M1`, `&vss`, `%x`, `$X`, a terminal or a sub-object) makes no object; it takes an existing handle. A lone `#M1` is a statement too: it gives M1 with no facts. A lone `#` gives this level with no facts.
- A handle is the thing described: an instance, net or terminal of the host, or the level; or a point, wire, set, usage, constraint and so on made by RNL. A handle is never another RNL object.
- The object of a statement either has a new handle (the statement creates, such as WIRE, graphics, `{…}`, selections, AS, ATTACH and constraints), or keeps the handle of its input and adds new facts to it (the statement modifies, such as PLACE, ORIENT, SET-SYMBOL, SET-STYLE, GROUP and CURRENT; ADD-TO and REMOVE-FROM keep the handle of the target set). The signature says which (5.3); WIRE creates when it is given a net and modifies when it is given a wire (13.1).
- Two statements with the same content are two objects, such as two `#M1 @PLACE(1, 2)`. Their facts do not contradict each other and take effect together.
- A fact is committed once, at the statement that makes it (5.4). When the level takes effect, all facts about one handle are combined; contradictions are handled as 1.5 item 3 says.
- Properties are facts: position, orientation, symbol and style are facts about the object, and there are no property fields to overwrite. When the level takes effect, the facts about one handle are combined into the final values: the x and y of PLACE as geometric facts (5.8, 8.5), symbols and styles through the appearance queues (8.3).
- The kind of a handle follows from where it comes from: host objects get it from the host, RNL objects from the statement that makes them. The signature says which kinds each slot takes (Annex A); a description that receives an object of a wrong kind fails. What can be checked when the text is read is checked then; what is known only later (whether a usage has a symbol, the roles of a definition) is checked when the level takes effect (7.1).

Kinds of objects:

Kind	Made by	Drawn
level	the host; `#`	no
instance	the host; `#M1`	with its symbol
terminal	the host; `#M1:d`, `#:inp`. The terminals of a usage are the roles of its definition (6.4)	as a pin of the symbol
net	the host; `&vss`	no; wires and labels are objects of their own
point	`(x, y)`	no; a junction dot is `$JUNCTION` applied to a point (13.2)
wire	WIRE given a net; a wire whose net slot receives a set is a bus. Given a wire, WIRE rewrites it (13.1)	yes
label	a TEXT attached to an object; with a key it is the sub-object `object::key` of that object (13.3)	yes
bundle	BUNDLE (11.8)	once, in place of its members
annotation	a graphic in the scene (LINE, RECTANGLE, CIRCLE, ELLIPSE, ARC, BEZIER), or a TEXT attached to a point (13.4)	yes
usage	AS, ATTACH (9.4)	only when it has a symbol; it is then a virtual part (chapter 12)
constraint	ALIGN, ORDER, DISTRIBUTE, DISTANCE, ORIENT-RELATION (chapter 14)	no
set	`{…}`, selections, UNION, the result of generating one by one (5.2, chapter 10)	not itself; its members are drawn on their own
definition	DEFINE-SYMBOL, DEFINE-STYLE, DEFINE (chapter 9)	no; it acts only where it is used

Of these kinds, instance, terminal, net, point, wire, label, bundle, annotation and usage have kind names (5.3), and so does `text`, which covers every TEXT, whether a label or an annotation; Annex A gives how signatures name the others.

There are also sub-objects (6.5), whose handle is a parent object and a name, and failed objects (chapter 7).

5.2 Sets

- `{…}` makes a set object. Its members are ordered. A set can be named (`%pair_set = {#M1 #M2}`), and other statements can add members to a named set or remove them (5.2.2).
- A set keeps repeated members and the order in which they are written, such as `{#M1 #M2 #M3 #M1}`. Whether two members are the same member is decided by handle: two `{…}` written separately are two handles and not the same member, even when their contents are equal.
- **A reference is spread one level**: a reference in a set that refers to a set puts the members of that set in its place, so `{%a_set %b_set}` is the members of a_set followed by those of b_set. The same holds for a statement whose result is a set, such as a selection, and for the list of references that a subscript expands to, such as `&D<7:0>` (6.11). To remove duplicates, write `%a_set %b_set @UNION` (5.2.2).
- **A `{…}` written out is kept**: a `{…}` written directly in a set is kept as one member, so `{{%a_set} {%b_set}}` is a set of two sets.
- **Loose objects are spread too**: when a keyword with one slot receives several objects written without braces, they are put together exactly as if they were written in `{…}` (5.2.1 item 3), so a set among them is spread. After `%bias_set = {#R1 #R2 #R3}` and a text `%lbl`:
  - `%bias_set %lbl @ALIGN("x")` is `{%bias_set %lbl} @ALIGN("x")`: R1, R2, R3 and the text, one by one, get the same x;
  - `{%bias_set} %lbl @ALIGN("x")` keeps the set as one member: the text gets the x of the centre of R1–R3;
  - a set that is the only object is used as itself: `%bias_set @ALIGN("x")` aligns R1, R2 and R3.

  Among the members of ORDER, ALIGN and DISTRIBUTE, a set counts as one member, by its centre, only where it is written `{%x}` (14.1).

  Note: Spreading is most often misread in constraints. `{%stage1_set %stage2_set} @ORDER("x")` orders the members of both sets one by one, which often contradicts an ALIGN within each set. To order the two sets, write `{{%stage1_set} {%stage2_set}} @ORDER("x")`, or use the two slots of DISTANCE (14.5). Spreading is kept because it is the only way to join sets. Loose objects are read exactly as if written in `{…}`, so that adding braces never reverses the meaning: were loose objects not spread, `%bias_set @ALIGN("x")` would count the set as one member and align nothing, `{%bias_set} @ALIGN("x")` would spread it and align R1–R3, and `{{%bias_set}} @ALIGN("x")` would keep it whole again.
- The content of a set is computed in the phase that uses it, as for a selection: its initial content, and the selections in the statements that add or remove members, are evaluated on the state before that phase begins, and the additions and removals are applied in the order of writing (their position, 8.5). So a use written earlier also sees all additions and removals (1.5 item 1), and one set can differ between phases: the rules that generate objects see it without the generated objects, style rules see it with them (8.2).

  Note: The content is not computed once at a single step, because then naming a selection would change its result: `%g_set = # @SELECT-USAGE $GROUND; %g_set @SET-STYLE $grey` would miss the generated grounds, which the same selection written without a name finds; a name is only another way to refer to an object (1.5 item 5).

5.2.1 Objects and slots

The objects written in a statement are matched to the slots of the signature as follows.

1. **Marked with roles**: an object marked with a role goes into that role whole and is not split. In one statement either all objects are marked or none.
2. **Splitting one level first**: when no roles are marked and the whole statement has a single object, which is a set, that set is first split one level. A definition and the result passed in by a pipe count as objects. So

`{#M1 #M2} @DISTANCE(…)` is `#M1 #M2 @DISTANCE(…)`, while `{#M1 #M2} @SET-SYMBOL $X` is two objects (the set and the definition) and is not split.

3. **Matching by the signature**:
   - A keyword with several slots (a definition takes a slot too): the number of objects must equal the number of slots, and they are matched one to one in order; otherwise the statement fails. For AS and ATTACH, the set roles at the end of the role list may be left without objects (9.4).
   - A keyword with one slot (such as PLACE, ORIENT, ALIGN, ORDER, DISTRIBUTE, GROUP, the selections other than SELECT-USAGE and SELECT-ANCESTOR-USAGE, which also take a definition (5.6), and the definition keywords, whose slot is the body): a single object goes into the slot as it is; several objects are put together into one set, in which referenced sets are spread one level and each `{…}` written out is kept as a member. When the objects are the members of the only set, split by item 2, the slot receives that set itself.
   - When the only object is an empty set:
     - A slot that takes a single object does its work zero times: nothing happens and nothing fails. A name is bound to the empty set as usual, and the statements that use the name do nothing.
     - An empty set written out for a slot that takes a single object gives a warning, such as `{} @PLACE(1, 2)`. An empty set from a selection or a rule gives none: a rule is evaluated in its level and in each level below, and a level without objects of that kind is normal. When a selection written by the user selects nothing in every level where it is evaluated, an implementation may give one warning; the condition is probably wrong.
     - A slot that takes a set accepts an empty set, such as the empty body of `@DEFINE(roles=["a"]) {}`, or a constraint without members, which is trivially met. The exceptions are BUNDLE and the net slot of WIRE: they make a new object from the set, which cannot be made from nothing, so the statement fails (11.8, 13.1).
     - A statement without any object, such as a lone `@PLACE(1, 2)`, does not match the signature and fails.
4. **A set in a slot**:
   - A slot that takes a single object does its work once for each member. Each member is a description of its own, and the result is the set of the results. A member that is itself a set is split again and done member by member. The one exception is PRIORITY: given a set marked with GROUP, it acts on the group itself (11.5, 14.8).
   - A slot that takes a set takes it whole.
   - A slot that takes an object or a set, such as either slot of DISTANCE, takes the set whole; there the set counts by its extent (14.5).
   - In one statement at most one slot that takes a single object may receive a set of more than one member. When more than one such slot does, the statement fails: pairing the members in order and taking all combinations would both make sense, so the meaning is not unique.

Written	Result
`{#M1 #M2} @SET-SYMBOL $X`	Two objects, not split. The instance slot receives a set: M1 and M2 each get the symbol
`#M1 #M2 @SET-SYMBOL $X`	Three objects for two slots: the statement fails
`{#M1 #M2} @DISTANCE(…)`, `%pair_set @DISTANCE(…)`	The only object is a set and is split: from M1 to M2
`%a_set %b_set @DISTANCE(…)`	Two objects, each a set and taken whole: between the two sets (14.5)
`{#M1 #M2} @ALIGN("y")`, `#M1 #M2 @ALIGN("y")`	ALIGN has one slot, which receives {M1 M2}
`%bias_set %lbl @ALIGN("x")`, with `%bias_set = {#R1 #R2 #R3}`	The loose objects are put together as `{%bias_set %lbl}`, in which bias_set is spread: R1, R2, R3 and the text, one by one, get the same x
`{%bias_set} %lbl @ALIGN("x")`	The `{…}` written out is kept: the text gets the x of the centre of R1–R3
`%pair_set @PLACE(10, 20)`	The slot receives pair_set itself and places it member by member: both members at (10, 20). To place the centre of the set, write `%pair_set::position @PLACE(10, 20)`
`%a_set %b_set @PRIORITY("disabled")`, both sets with GROUP	The loose objects are spread as in `{…}`: PRIORITY receives the members and fails on them (wrong kind). To turn off two groups, write one PRIORITY for each (11.5, 14.8)

Written	Result
`{#M1 #M2} @SET-SYMBOL $X => @ALIGN("y")`	The result of the pipe is the only object and goes into ALIGN: aligns M1 and M2
`{#M1 #M2} @SET-SYMBOL $X => #M3 @DISTANCE(…)`	Two objects, not split. The slot `from` takes the set whole and counts by its extent: from M1 and M2 together to M3 (14.5)
`# @SELECT-NET(…) => $PULL_DOWN @ATTACH &vss`	Three objects for three slots (two roles and the definition). The first role is a single role, so the definition is applied once for each selected net, each time with vss in the second role
`# @SELECT-INSTANCE(type="nmos") => @ORIENT("MY")`	Mirrored one by one; with no NMOS, done zero times, and nothing happens
`# @SELECT-INSTANCE(name="Q*") => @SET-SYMBOL $NPN`	Two objects, not split; with no BJT, done zero times, and nothing happens
`{{#M1 #M2 #M3}} @ALIGN("y")`	Split one level, one set object remains: aligns M1, M2 and M3
`#R1 {#R2 #R3} #R4 @AS $CHAIN`	Four objects for four slots (three roles and the definition); the set role `chain` receives {R2 R3} (9.4)
`&D<7:0> @WIRE {…}`	Two objects, not split. The net slot takes the list of nets whole: one bus (13.1)
`#XI0<3:0> @ALIGN("y")`	The only object is the list that a subscript expands to, and it is split: aligns the four instances (6.11)

- A constraint with too few members is trivially met and does not fail, such as an ALIGN with one member (chapter 14).
- To treat a set as a whole: PLACE takes its `::position` (6.5); as a member of ORDER, ALIGN or DISTRIBUTE, write it as `{%x}`, which counts by its centre; as an object of DISTANCE it counts by its extent (14.1, 14.5).
- The roles of a definition are single roles or set roles (9.4). A role written `"{name}"` in the role list is a set role and takes a set whole; any other role is a single role, and given a set, the definition is applied once for each member.

5.2.2 Adding and removing members, and duplicates

- `X @ADD-TO S` appends X at the end of the set S (the members of X when X is a set); `X @REMOVE-FROM S` removes every occurrence of these members from S. The target is written last, so a selection can be passed in by a pipe: `# @SELECT-NET(name="vss") => @ADD-TO %gnd_set`.
- The target must be a set; it is not wrapped into one. After `%tail = #M5`, `#M3 @ADD-TO %tail` fails, because M5 is not a set.
- Additions and removals are applied in order of position (1.5 item 2). A set evaluated in a level holds only the members of that level (10.3).
- `%a_set %b_set @UNION` makes a new set without duplicates, in the order of first occurrence; whether two members are the same is decided by handle.
- To make a new set and leave the old ones as they are: the concatenation is `{%a_set %b_set}`; the union `%a_set %b_set @UNION`; the difference `%c_set = {%a_set}; %b_set @REMOVE-FROM %c_set`.
- A set depends on the sets written in the statement that makes it (inside its braces, in UNION, or as the scope of a selection) and on the sets that ADD-TO and REMOVE-FROM add to it or remove from it. Dependencies that form a cycle are an error, with or without a removal: taking the statements in order of position (8.5), the one whose dependency closes the cycle fails, and its diagnostic lists the whole cycle; the others take effect as usual. So `%a_set @ADD-TO %b_set; %b_set @ADD-TO %a_set` fails at the second statement, and `%b_set = {%a_set}; %b_set @REMOVE-FROM %a_set`, whose removal would have no definite result, fails at the removal.

5.3 Statements and signatures

- A statement is made of at most one keyword, some objects and a list of parameters.
- Every keyword has a signature, which declares:

  - the number of objects, and the name of each slot and the kinds of objects it takes. For AS and ATTACH the number is the length of the role list of the definition plus one for the definition itself (written last for AS, first for ATTACH), and the kinds are those declared for the roles (9.4);
  - the parameters: their names, types, whether they are required, their allowed values, and the order in which they may be given by position;
  - the output: a new object, the handle of the input, a sub-object, a definition, or no handle; WIRE gives a new wire when it is given a net, and the wire it is given when it is given a wire (13.1);
  - how each input is consumed (5.4), and whether it is strict or tolerant of failures (7.3);
  - where the keyword may be written (see the word list below).
- The number of objects is fixed; a varying number of objects is given as a set. There are no optional slots, with these exceptions: a set role of AS and ATTACH that is not filled is an empty set (9.4); the definition of SELECT-USAGE may be left out (10.1); the body of USE-PACKAGE may be left out (16.2). A definition takes a slot too.
- Objects are matched to slots by role or by order, all marked or none; a role can be marked only once. The object passed in by a pipe is the first object; AS and ATTACH match it by role, and it does not take the slot of the definition (5.5). How objects are matched to slots is given in 5.2.1.
- The objects written in a statement can be marked with roles, whatever the keyword. The role name is the slot name in the signature: `#M5:3(to) #M1:3(from) @DISTANCE("x", min=1)` is `#M1:3 #M5:3 @DISTANCE("x", min=1)` written in order. A role name that is not a slot name of the keyword makes the statement fail; it is not ignored (Principles 2.3, 2.8). The objects of a keyword with one slot can be marked too, but the mark adds nothing.
- The parameters of AS and ATTACH are those of the parameter list of the definition (9.4), plus one standard parameter, `priority` (14.8); no parameter of a definition can be named `priority`.
- **Standard word list**: each language version publishes a machine-readable list of its standard words, which is part of the standard. What each keyword takes, and which parameters it has, are given only in the list; the main text gives their meaning. Annex A is generated from the list, and a contradiction between the two is an erratum. The checks of an interpreter, documentation, completion in editors, the skeleton of conformance tests and writing references for LLMs are generated from the list. For version 0.3 the list is published as `rnl-words-0.3.json`.
  - The list holds all standard words: the keywords with their signatures, slot names, parameter names and values, the fields of the record head, kind names, sub-object names, the attribute names that `${…}` reads, style properties, and `true` and `false`. Aliases (16.5) are declared for words of the list, and version checks and suggestions of the closest name look words up in it.
  - The list is a JSON file, and the JSON Schema `rnl-words.schema.json` describes the structure of the list itself. Tools and the maintainers of the standard read and write it, not those who describe circuits. Every word has `since`, the version that added it; reserved words are marked (version 0.3 reserves no keywords). A word that is not in the version a file declares is treated as an unknown word, and the diagnostic says which version is needed (16.1).
  - The list records, for each keyword:
    - each slot: its name, the kinds it takes, whether it takes a single object or a set (5.2.1), and how it is consumed (5.4), which decides whether it is strict or tolerant (7.3); the slot of the definition; for AS and ATTACH, the slots follow the role list of the definition;
    - each parameter: its name; its type (number, string, truth value, or array with the type of its elements; where the values are fixed, the values); whether it is required; its default; the order of the parameters that may be given by position (the others are given by name only); and conditions on combinations, such as at least one of `x` and `y` of PLACE, or only one of the centre form and the corner form of RECTANGLE;
    - the output: a new object and its kind, the input passed on, the target passed on (ADD-TO), a definition, a sub-object (TEXT with `key=`), or none; for WIRE, also the kind of object that it rewrites when it is given one (13.1);
    - where it may be written: in a level; only in a symbol body (PIN); only in the header record (such as VERSION, ALIAS, DEVICE); only in the header record of a package file (PACKAGE);
    - how its facts are overridden (5.8): by key, with the key; as geometric facts (8.5); through the appearance queues (8.3); or added up;

  - its standard form and standard sentence (4.3, 4.5, Annex B);
  - a one-line description and the section of the main text.
- Not in the list: the meaning (what facts a keyword makes, in which phase, how they conflict) is given in the main text. The signature of a general definition in a package is its role list and parameter list (9.4).


```
{
 "keyword": "PLACE",
 "since": "0.2",
 "group": "Placement and appearance",
 "summary": "Puts the origin of an object at a point; one axis is enough.",
 "section": "11.1",
 "where": [
  "level",
  "symbol"
 ],
 "slots": [
  {
   "name": "target",
   "takes": "one",
   "kinds": [
    "instance",
    "usage",
    "bundle",
    "point",
    "text",
    "position"
   ],
   "mode": "absorb"
  }
 ],
 "params": [
  {
   "name": "x",
   "type": "number",
   "since": "0.2"
  },
  {
   "name": "y",
   "type": "number",
   "since": "0.2"
  },
  {
   "name": "priority",
   "type": "enum",
   "values": [
    "disabled",
    "low",
    "normal",
    "high"
   ],
   "since": "0.2"
  }
 ],
 "positional": [
  "x",
  "y"
 ],
 "at_least_one": [
  "x",
  "y"
 ],
 "output": "same",
 "override": "geometry",
 "key": "object and axis",
 "form": "{target} @PLACE(x, y)",
```

```
 "sentence": "#M1 is at @PLACE(10, 20)"
}
```

- **Slot names of built-in keywords**: every slot has a name, which is the role name that can be marked. Slot names are standard words and are versioned like keywords: renaming one is an incompatible change; the slots of one keyword have different names. The slot of a definition has a name too, used in the list and in diagnostics, but it cannot be marked (4.2 item 7). There is one set of kind names, used both in signatures and in the role declarations of definitions (9.4), `instance`, `terminal`, `net`, `point`, `wire`, `usage`, `bundle`, `text`, `annotation` and `label` (Annex A).

5.4 Ways of consuming input

Way	Keywords	Facts of the input	Failures (default)	Names
absorb	most keywords	carried to the output	strict	names in a nested set hold only in that set
enclose	DEFINE, DEFINE-SYMBOL, DEFINE-STYLE	become the content of the definition and are not passed on	tolerant	local; renamed for each application, so that each usage has its own (9.4)
commit	the record, `@RNL`	merged into the level in order	tolerant	published to the level and shared by all its records; may be used before they are bound

- What a member produces is all the facts made by its segments and by the sets nested in it, not only the handle at its end. They are committed with the top-level member that contains them.
- Every statement of a record is committed, whether it is named or not. What it does then depends on the object: a description in the scene is drawn or takes part in solving; a definition becomes available and draws nothing by itself; references, and sets made only of references, have no effect. The statements in a plain `{…}` are committed with the member that contains it, so the line of `{ @LINE {…}}` is drawn (13.4).
- The statements in the body of a definition are not committed; they become the content of the definition. A general definition is expanded only when it is applied (9.4).
- A USE-PACKAGE with a body takes the body by enclosing too: the statements of the body are the content of the package and are not committed to the level. How the names of a package are seen is given in 16.2.

5.5 Pipes and nesting

- `a => @K(…)` makes the result of `a` the first object of K; AS and ATTACH put it into the first role, not into the slot of the definition. A result that is a set is still one object (5.2.1). The segment after `=>` takes this object only by position and cannot mark it with a role.
- A pipe is not run step by step. It expands into several descriptions linked by handles: each segment describes the handle that results from the segment before it, and a segment that works member by member expands into parallel descriptions (1.5 item 1).
- **The order of the segments.** Each segment has a place of its own, and a later segment counts as written later than an earlier one wherever the place of writing matters (1.5 item 2): in override by key (8.4), in the appearance queues (8.3), in the order of resolution (8.5), in the drawing order (11.6) and for the rewrites of a wire (13.1). So in `#M1 @SET-SYMBOL $A => @SET-SYMBOL $B`, M1 is drawn with `$B`.
- **A failure reaches only the descriptions that depend on it.** A description fails for one of two reasons only: it cannot be understood or contradicts itself; or a handle that it uses should have been made by a description that failed.
  - In `$GROUND @ATTACH &vss => @PLACE("20", 38) => @ORIENT("R90") => %g1` only PLACE fails (a string where PLACE takes a number, 5.3); the ground is still made, and ORIENT still takes effect.
  - In `#XI0<3:0> @BUNDLE(bad=1) => @PLACE(10, 20) => %s` the bundle is not made (BUNDLE has no parameter `bad`), so PLACE has no object to describe and fails as well.
- What a name binds: `%x =` in front binds the last handle of the whole pipe; `=> %x` binds the result of the segment to its left, which is the last handle when it is written at the end, and the result of that segment when it is

written in the middle, such as `#M1 @AS $A => %a => @AS $B => %b`. PLACE, ORIENT, SET-SYMBOL, SET-STYLE and GROUP keep the handle of their input, so in `$GROUND @ATTACH &vss => @PLACE(24, 38) => %g0` the name `%g0` is the usage of this application, that is, this ground. AS and ATTACH make new handles: the second segment of `#M1 @AS $A => @AS $B` applies `$B` to the usage made by the first segment (nesting), not to M1. To apply two definitions to M1, write two statements. When the segment to the left of a name fails: if it would have kept the handle of its input, the name still refers to that handle (`%g1` above is the ground); if it would have made a new object, the name refers to the failed object (`%s` above).

- An object may be written as a set that holds statements (nesting). Names bound in a nested set hold only in that set. Each nested statement is a description of its own: in `{%a = #M1 @PLACE(1, 2); #M9} @ALIGN("y")`, when the host has no M9, ALIGN fails and the coordinates of M1 still take effect.
- The standard form writes statements flat: name an object, then refer to it in other statements. A single statement is named with `%x =` in front, a pipe with `=> %x` (4.3).

5.6 Definitions as objects

A keyword that uses a definition takes it as one of its objects, and the keyword fixes which slot holds it: for ATTACH it is the first object written; for SET-SYMBOL, SET-STYLE, AS, SELECT-USAGE and SELECT-ANCESTOR-USAGE it is the last object, and SELECT-USAGE may leave it out (10.1).

```
%ax = (_, _);
%w13 = &x @WIRE {#M1:1 #M3:1};
#M1 @SET-SYMBOL $NMOS;
%w13 @SET-STYLE $signal;
#M1(a) #M2(b) %ax(axis) @AS $MIRRORED;
$GROUND @ATTACH &vss;
# @SELECT-USAGE $GROUND => @SET-SYMBOL $GROUND_SYMBOL
```

- A definition is not marked with a role and does not count for "all marked or none". It is an object and takes a slot, so `{#M1 #M2} @SET-SYMBOL $X` is two objects (5.2.1).
- Because the keyword fixes the slot of the definition, a statement can be parsed without knowing whether a `$` name refers to a definition or to a set: in `$GROUND_STUB @ATTACH $GROUND_FILL_SET` the first is a definition and the second a set.
- A definition cannot be passed in by a pipe: the object passed in fills the first slot, and AS and ATTACH match it by role, outside the slot of the definition (5.5). So `$GROUND @ATTACH &vss` cannot be written `$GROUND => @ATTACH &vss`.
- A definition may be a nested anonymous definition: `#M1 @SET-SYMBOL { @DEFINE-SYMBOL {…}}`.
- A general definition may be written later or in another level, so the number of objects, the role names and the parameters of AS and ATTACH are checked when the level takes effect (7.1).

5.7 Handles and names

- The host gives the handles of host objects: instance names and net names. A terminal is an instance and a terminal number, so `#M1:1` and `#M1:d` are the same terminal.
- For what RNL makes, the handle comes from the statement that makes it. Another statement can refer to it only through a name: the name is the only way to find it, except in the record of the solver, which refers to an object that has no name by its number (5.9). An object without a name can otherwise be used only within its own statement (in a pipe or by nesting).
- A name never refers to two things: it is bound only once in a level (6.6).
- The handle of a sub-object is its parent object and its name; it needs no name of its own.
- The handle of a local object made by expanding a general definition is its usage and its local name, written `%c1::mid` from outside (9.4). The local objects of a usage without a name are shown in diagnostics by their position only; only the record of the solver can refer to them, through the number of the usage, such as `%#5::mid` (5.9).
- A single object of a generated set has no name and cannot be referred to on its own: objects are not identified by their kind and the objects they relate to (there are no natural keys), and sets offer no reference by index.

5.8 Keys of facts

- Facts with a key are overridden by key: for the same key the later fact takes effect; when the later one fails, the earlier valid value is kept.
- Geometric facts are the exception: the x and the y of PLACE (one key for each coordinate) and ORIENT (one key for each object). Two different values written for the same key conflict and are resolved by 8.5: with the same priority the later one is kept, which is the same result as override by key; with different priorities the higher one is kept. PLACE changes only the coordinates it gives; a coordinate never given stays free.
- Making an object under a name that is already bound is not override by key: a name is bound only once in a level (6.6). To change an object made by RNL, write a statement that modifies it, such as `%g0 @PLACE(40, 38)`.
- Facts without a key add up: those that do not conflict hold together, and conflicts are resolved by 8.5.
- SET-SYMBOL and SET-STYLE are not overridden by key; they take effect through the appearance queues (8.3).
- VERSION is written only once, in the header record (16.1), so it is never overridden.
- CURRENT has one key for each terminal (15.2), and LAYER one for each object (11.6).
- Usages have no key: a definition applied several times makes as many usages, which do not override each other. Two identical applications make two usages, so `$GROUND @ATTACH &vss` written twice draws two grounds. Facts written on one usage by statements that modify it, such as PLACE, do have keys and follow the rules above.
- The priority of an object is a fact with a key; the order in which priorities override each other is given in 14.8.
- A TEXT with a key (`key=`) is one text for each attached object and key, and a later one overrides it parameter by parameter (13.3). A TEXT without a key is not overridden by key: each one written is one more text (13.3).
- A wire has no key: written twice, it is two wires (13.1). WIRE given a wire rewrites it; the rewrite has a key, the wire, so a later rewrite replaces an earlier one (13.1).
- Constraints have no key. To change a constraint, switch it off (`"disabled"`, 14.8) and write a new one, or edit the text.

5.9 Numbers of objects without a name

```
* @RNL(type="header") { @VERSION("0.3")}
.model nch nmos level=1
.subckt amp in out vdd vss
M1 out in vss vss nch
R1 vdd out 10k
* @RNL {
*   #M1 #R1 @ALIGN("x");
*   &out @WIRE {#M1:1 (_, _) #R1:2};          /* %#1: the wire; %#2: the free point */
*   {#M1 #R1} @TEXT("${name}")                /* %#3: the label of M1; %#4: that of R1 */
* }
* @RNL(type="generated", priority="high", author="rnl-solver 0.1", hash="sig1:…", numbering="…") {
*   #M1 @PLACE(10, 0); #R1 @PLACE(10, 8);
*   %#2 @PLACE(12, 4);
*   %#1 @WIRE(mode="straight") {#M1:1 (12, 2) %#2 (12, 6) #R1:2};
*   %#3 @PLACE(13, 1); %#4 @PLACE(13, 9)
* }
.ends amp
```

- The record of the solver refers by number to the objects that cannot be reached by a name, so that it can write back every decision (14.8). A number is written `%#` followed by an unsigned integer, such as `%#3`, and is used like a name: `%#5:p` is a terminal of the usage `%#5`, and `%#5::mid` one of its local objects. A number only refers to its object; it is not bound like a name, so `%#3 = …` and `=> %#3` are syntax errors (4.1).
- **Which objects.** The points, wires, labels, bundles, annotations and usages that have no name of their own and cannot be reached through `::` from an object that has a name or a number (6.5, 9.4). A name bound only in a nested set (6.7) does not count, since the record of the solver cannot use it. For one record of the solver, the objects are, in this order:
  1. those made by the members of the authored records of the level (`type="authored"`, 2.8), and in the bodies of the usages that these members make. They are counted on the level interpreted as the solver sees it, without the records with `type="generated"` (2.8), and keep their numbers when the level is interpreted with those records;

2. those made by the members of the record itself, and in the bodies of the usages that they make. They are counted on the record as written, as if every number in it could be resolved; whether the check below matches does not change them.

Constraints and sets are not numbered. Objects that rules generate one by one (10.5) are not numbered either: the solver writes them as explicit statements, which fall under item 2. Each record of the solver has its own numbers: item 1 is the same for all of them, and item 2 counts only the record's own members.

- **Order.** The authored records are taken in the order of 2.6: the level's own text, then the records written into the level from outside with `scope`. Within a member, objects are numbered in the order of the token that makes them: the `(` of a point, the keyword of the others. When one token makes an object for each member of a set (5.2.1), each object has a number of its own, in the order of the members of the set. The objects in the body of a usage are numbered right after the usage, in the order of the body, and so on down through nested usages. Numbers start at 1.
- **Where.** Numbers may be written only in records with `type="generated"` that have the field `numbering` and no `scope` (2.6, 2.8). Anywhere else a number is an error: an authored record refers to objects by name. Within its record, a number may be used before the member that makes its object, as a name may (6.7).
- **Check.** `numbering` holds a check of how the numbered objects are made. For each number, in order, there is one line:
  - the tokens of the member of the record, or of the definition body, that makes the object, with all its segments (4.1), without free text and comments, separated by one space. Here a token is a lexical unit: a keyword; a reference, with its terminals, sub-objects and subscript, written without white space; a string or a number as written; a name; `_`; or one of `{ } ( ) [ ] , = ; =>`. So `&out @WIRE {#M1:1 (_, _) #R1:2}` gives `&out @WIRE { #M1:1 ( _ , _ ) #R1:2 }`;
  - for an object in the body of a usage, the line of the usage comes first, formed in the same way whether or not the usage has a number, then a tab and the tokens of the member of the body, as the definition writes it;
  - when the token makes an object for each member of a set, a tab and the handle of the member follow, written as a reference: a host instance or net by its name; a terminal by its instance and terminal number, such as `#M1:1`; the position of an object, or a text on a key, by the object followed by `::position`, or by `::` and the key; an RNL object by a `%` name that the level binds to it, the one bound first in the order of 2.6 when there are several, whatever name the set was written with; an object without such a name by its number, or, as a local object of a usage, by the reference of the usage, formed by these same rules, `::` and its local name: the name bound first in the body of the definition when there are several, also within one member (in `%p = (0, 0) => %q` it is `p`), whatever name the set was written with, and through the usage applied first when several usages have it as a local object. A name is written as it is when it consists of ASCII letters, digits and `_` only, and otherwise in double quotes, with each `"` doubled (3.6). For example, after `%p = (0, 0); %q = %p; %s = {%q}`, the annotation that `%s @TEXT("x")` makes has the line `%s @TEXT ( "x" )`, a tab and `%p`, and the check `675df5171032726d`. With `$D = @DEFINE(roles=["x"]) { %p = (0, 0) => %q }; %u = #M1 @AS $D; %s = {%u::q}`, the line of `%s @TEXT("x")` ends with a tab and `%u::p`, and the check is `a8c9571193175903`.

The check is the first 16 hexadecimal digits, in lower case, of the SHA-256 of these lines in UTF-8, each ended by a line feed.

- When the check of a record does not match the level, the statements of the record that use a number take no effect, and, when there are any, the record gives one hint: the solution is out of date for the objects without a name. A record that uses no number need not have `numbering`. The other statements of the record take effect as usual (chapter 17). A number beyond the last numbered object is an error.
- In the example, the check has one line for each of six numbers: the WIRE member twice (for the wire, then for the point), the TEXT member twice, followed by `#M1` and by `#R1`, and the rewriting WIRE twice more, since the two points written as coordinates in it are objects of the record itself, with the numbers 5 and 6. A viewer that does not solve draws the free point, the route of the wire and the two labels as the solver chose them.
- Writing `{#R1 #M1}` instead swaps the labels 3 and 4: the check no longer matches, and the statements with numbers take no effect until the level is solved again, while M1 and R1 keep their positions. Writing the ALIGN differently, or moving M1, leaves the numbers as they are.

- Note: the check covers only how the numbered objects are made. Moving a device, or changing a statement that makes no object without a name, leaves the numbers as they are; inserting, removing, changing or reordering a member that makes such objects puts them out of date until the level is solved again. A record of the solver normally holds no statements that change sets or definitions; item 1 is counted without such records in any case.

6 Names and scope

6.1 Hierarchy of levels

- **Levels follow the text.** The RNL of a level applies to that level and to the levels that it contains in its text; which subcircuit instantiates which plays no part.
  - A file contains the subcircuits it defines and the files it includes.
  - An `.include` written in a subcircuit, and a `.subckt` definition nested in a subcircuit (a local definition in SPICE3 and ngspice), are below that subcircuit.
- From the top, the chain of levels down to a subcircuit is: the packages at the root (the standard package, and the packages of USE-PACKAGE in header records, 16.3) ⟶ the file level of the main file ⟶ … (the included files, one within another) ⟶ the file that defines the subcircuit ⟶ the subcircuit itself. For example:

```
packages at the root (the standard package)
└ file level of tb.sp (the top level)
   ├ load                   a subcircuit defined in tb.sp
   ├ file level of ota.sp   a file included by tb.sp
   │  ├ ota                 ota instantiates cell
   │  └ cell
   └ file level of bias.sp
      └ bias                also instantiates cell, but neither the RNL of bias.sp nor that of bias
reaches cell
```

- **The RNL of a subcircuit covers only the subcircuit.** In resolving conflicts its statements come after the content of the file levels, and so take precedence over it (8.5); but they are not passed to the subcircuits it instantiates, except to those defined inside it. When amp instantiates cell, the definitions and rules of amp do not reach cell.
- **An included file does not affect the level that includes it.** The definitions, rules and USE-PACKAGE of its file level apply only to its subcircuits and to the files it includes in turn. In the netlist, the devices it writes outside subcircuits belong to the level that includes it, but its RNL cannot change them. So a file is never changed by the files it includes.
- **The file level of an included file has no circuit of its own.** Only the file level of the main file is a circuit. Host references (`#M1`, `&vss`) at the file level of an included file fail (6.3); only definitions, rules, USE-PACKAGE and `scope` have effect there. A file opened on its own is the main file.
- **An included file inherits from the level that includes it.** The house style of the main file (its rules and the packages of its USE-PACKAGE) passes down into the subcircuits of library files. So one library file may be drawn differently under different main files, or when opened on its own, while its solver result is stored only once (chapter 17). To keep the symbols that a library uses free of the appearance rules of upper levels, use the standard package again with USE-PACKAGE in the library file or the subcircuit (16.2). To pin only the version of the standard package and still let the house style from above override it, write the USE-PACKAGE in the header record of the library file (16.3). For rules of the main file that should apply only to its top level, write `recursive=false` (10.1).
- **One schematic for each subcircuit.** A viewer does not expand subcircuits: an instance of a subcircuit is always drawn with its symbol, and the inside of the subcircuit is drawn and solved separately. Each subcircuit has one chain of upper levels, so in a document each subcircuit has exactly one schematic.
- **Packages.** The `$` definitions and rules of a package take effect in the level where `@USE-PACKAGE` is written and in the levels below (16.2). The standard package is used implicitly at the root, and the packages of USE-PACKAGE in a header record are placed at the root too (16.3). An include places a file below the level, and the file inherits from the level; a USE-PACKAGE places a package above the level, and the level inherits from the package.
- **Rules are treated as definitions.** A rule written in a level is evaluated in that level and once in each level below, each time looking only at that level (10.4). With `recursive=false` its selection selects only in the level where the rule is written, so the rule in effect applies to that level only. This is what Principles 2.5 asks of definitions: a definition can limit its effect to the current level or let the levels below inherit it, and is inherited by default. The

descriptions that Principles 2.5 says belong only to their own level are the statements written object by object, whose host references resolve only in that level anyway.

6.2 Prefixes

Prefix	Refers to	Resolved in
`#`	An instance of the host; `#` alone is this level	The host of this level
`&`	A net of the host	The host of this level
`%`	A name given in this level: of a scene object, of an existing object such as `%tail = #M5`, or of a definition used only in this level	This level
`$`	A definition, or a set (6.6)	The level where it is bound, and the levels below
`?`	A role or a parameter of a definition; in a symbol body, `?` alone is the object drawn with the symbol (13.3)	The body of a general definition or of a symbol

`$X` and `%X` are two different names.

6.3 Host references

- `#M1` and `&vss` are resolved only in the host of this level, never in another level (Principles 2.5). Instances with the same name in different subcircuits are unrelated.
- A host reference that is not found fails (Principles 2.4). The file level of an included file has no circuit of its own, so every host reference there fails (6.1).
- A name may be followed directly by a subscript, which stands for one or several host names, such as `&D<3>` and `#XI0<3:0>` (6.11).
- The host adapter decides which nets a level has, following the semantics of the host:
  - Local nets: the nodes that the netlist lines of the level use, ports included.
  - Global nets: `0` is the global ground in every SPICE dialect. The aliases of ground that a simulator itself knows (such as `gnd` in ngspice, and `GND`, `GND!`, `!GND` and `GROUND` in HSPICE) are merged into `0` by the adapter for the dialect; after merging, `&gnd` and `&0` are the same handle. A name declared with `.GLOBAL vdd` (`global vdd` in Spectre) is the same node in every subcircuit. A name ending in `!`, such as `vdd!`, is a convention of Virtuoso schematics; what counts is the declaration.
  - A global net has one handle in each level that uses it: the copy in that level. A level whose netlist lines do not use it has no copy. So `&0` refers to the copy in the level where it is written: each schematic draws its own, the facts belong to that level, and descriptions in a lower level cannot change an upper level.
  - To mean the global net in every level, select it by name in an upper level with the condition `global=true` (10.1).

6.4 Terminals

- `object:terminal` is a terminal: `#M1:1`, `#M1:d`, `#XU1:inp`, `%g0:p`. `#:inp` is the port inp of this level.
- A number is the terminal number of the host, counted from 1.
- A name is a terminal name of the host: a name given by `@DEVICE` in the header record (6.12), or a port name of the subcircuit. A name that cannot be written unquoted (3.6) is quoted, such as `#XU1:"in+"`. A terminal or port name may be followed directly by a subscript, such as `#X1:D<7:0>` and `#:D<7:0>` (6.11).
- Case: the port names of subcircuits and the terminal names given by DEVICE follow the rules of the host: SPICE ignores case, so `#M1:D` works too; Spectre does not. The terminal names of a usage are the names in the role list of its definition and follow the rules of RNL names (3.6).
- The `label=` of a pin is only for display and cannot be used to refer to a terminal: terminals belong to the netlist, and a reference to one does not depend on the symbol in use.
- The terminals of a usage are the roles of its definition, numbered in the order of the role list; set roles are not terminals (9.4). Only a usage with a symbol has pins for its terminals, to which wires can connect (chapter 12).
- A terminal that is not found fails. Whether a terminal has a pin in its symbol is checked after the symbols are decided (7.1).
- Apart from the port names of subcircuits, terminal names are given by DEVICE; 6.12 lists the standard names of the SPICE family.

- A subcircuit that the netlist does not define (such as the PDK device in `XM1 … sky130_fd_pr__nfet_01v8`) has terminal names only when the user gives them with DEVICE in the header record (6.12), or uses a package that holds DEVICE statements (16.2); otherwise only numbers can be used.

6.5 Sub-objects

- `object::name` is a sub-object. It is always an object, never a value. Its handle is the parent object and the name.
- The sub-objects are:
  - A key of a text: the sub-object that a TEXT with `key=` makes on the object it is attached to, such as `#M1::wl`. A text with a key in a symbol is a sub-object of each object drawn with the symbol (13.3). `::name` and `::value` are not standard sub-objects, only common keys (13.3).
  - `::position`: the origin of the object. `#M1::position @PLACE(…)` is `#M1 @PLACE(…)`. For a drawn usage it is the point where the usage is drawn: the point of its main object when that object has one, otherwise the usage's own origin (12.1). For a TEXT it is the position of the text itself (13.3); for a RECTANGLE, CIRCLE or ELLIPSE written in the centre form, the centre (13.4).
  - `::net` of a terminal: the net it is on, such as `#M5:1::net`, which is `&tail`. For the terminals of a bundle see 11.8: when the members are on different nets, it is the set of these nets.
  - A role name of a usage: what fills that role, such as `%dp1::in_p`, which is M1, and `%g0::p`, which is the net vss. For a set role it is that set.
  - A named local object in the body of a usage, such as `%c1::mid` (9.4).
  - `::position` of a set: the middle of the extent of its members, computed from the point of each member (origin, pin or point); nested sets are followed down to the innermost objects, and symbols play no part. `%pair_set::position @PLACE(10, 20)` puts the centre of pair_set at (10, 20). There is no sub-object for the extent of a set: to keep two sets apart, write DISTANCE (14.5); to make a set one member of a constraint, write it as `{%x}` (14.1).
- The standard sub-object names are reserved: no role name, parameter name or local name of a general definition, and no key of a text, can be the same as one of them (compared as written, 3.6). A local name cannot be a role name, and the key of a text cannot be a role name or a local name of the usage it is attached to. A local name may be a parameter name: a parameter is a value, and `::` does not take values. No parameter of a definition can be named `priority` either (14.8).
- **The name after `::` is looked up in this order**: the role names and local names of the usage itself; their `definition=` aliases; the standard sub-object names and their aliases; the keys of texts (aliases: 16.5).
  - Keys come last, so a key cannot be spelled like a name that comes before it: a standard sub-object name or one of its aliases declared in this file (such as `"位置"` of `rnl-zh`), a role name or local name of the usage, or a `definition=` alias of these names. Such a key is an error, like the clashes above (13.3).
- `::` takes only parts of the object itself; finding objects in the scene by conditions is always done with selection keywords. So "all usages of a definition" is written as a selection (10.1), not as a sub-object of the definition. For the extent of the members of a usage, name the set of the members.
- Terminals use `:` and sub-objects `::`, so that sub-object names added in later versions cannot clash with the port names of a netlist.
- **Sub-object names**: there are two standard sub-object names, `position` and `net`, and both are reserved. All other sub-object names are dynamic: keys of texts, role names of usages, and local names in the bodies of definitions. Standard sub-object names added in later versions apply by version (16.1): in a file of an older version, a key with the same name stays valid.

6.6 Naming

- `%name = statement` and `$name = statement` name the result of a statement; in a pipe, `=> %name` and `=> $name` name the result of the segment to their left (5.5). A name is only another way to refer to an object and makes no object.
- `$name` binds a definition or a set, including the result of a selection. A `$` set evaluated in a level holds only the members of that level (10.3). Other scene objects belong to a level and cannot have a `$` name.
- `%name` can bind any object of this level: an object made by the statement, an existing object (`%tail = #M5`), a set (`%pair_set = {#M1 #M2}`), or a definition used only in this level.

- **A name is bound only once in a level.** Binding it again to the same object is allowed, such as `%x = #M1` written twice; binding it to another object is an error, and the name keeps its first valid binding. Only when all its bindings fail is the name bound to a failed object (7.2). This holds alike for statements that create objects, definitions, sets and the results of selections.
- Overriding applies to facts, not to names: to change an object made by RNL, write a statement that modifies it, such as `%g0 @PLACE(40, 38)` (5.8).

6.7 Scope of names

- The names committed by `@RNL` are published to the level: all records of the level share them, and a name can be used before the statement that binds it (Principles 2.9).
- A name bound in a nested set holds only in that set.
- A name bound in a definition body is local to the body (6.9).

6.8 Inheritance and hiding

- `$` definitions are inherited down the hierarchy of levels (6.1), never up: into the subcircuits of the file and into included files. A level does not see the `$` definitions of the files it includes; to provide definitions to upper levels, make a package and use it with USE-PACKAGE (16.2). The block symbol of a subcircuit is the only exception that goes up (9.2.6).
- A `$X` defined in a lower level with the same name as an inherited one hides the inherited definition in that level and the levels below, and gives a hint; the upper level still sees its own.
- The `$` definitions of a package used with USE-PACKAGE in a level count as one level above that level (16.2). When they hide definitions of the levels above or of the packages at the root, a hint is given, as for any hiding. A hint is also given when a package used in a header record defines a name of the standard package, or of a package placed at the root before it (16.3).
- `%` names hold only in their own level and are not inherited: not into the subcircuits of the file, and not across include. The `%` definitions of a package are private helpers of the package.

6.9 Names in definition bodies

- The names in a definition body are resolved where the definition is written, not again where it is applied.
- So hiding a definition changes only what the name refers to: the definitions already used inside other definitions stay as they are. For example, hiding `$DIFFERENTIAL_PAIR` in a lower level does not change the differential pair used in the body of a definition written in an upper level.
- A `%name` in a body is looked up first among the local names of the body, then among the definitions of the level. The scene objects of the level (points, wires and so on) cannot be referred to from a body; such a reference is an error.
- **Notation in bodies**: roles and parameters are written `?name`. Local names are bound with `%name =` (or `=> %name` in a pipe) and referred to as `%name`. From outside, local names are reached through the usage (9.4).
- A definition body does not see the name being defined: in `$NMOS = @DEFINE-SYMBOL {$NMOS; …}`, the `$NMOS` in the body is the definition of that name outside (in an upper level or a package). This is how a lower level hides and extends an upper definition; without such an outer definition it is an error.
- Host references in the bodies of DEFINE-SYMBOL and DEFINE are errors: `#M1`, `&vss`, `#:inp`, `#M1:d` and a lone `#` cannot be written there. Names in a body are resolved where the definition is written, but a general definition is applied in other levels and other files, where a host reference would still point into the level of the definition, so its meaning would not be unique. A general definition that needs host objects receives them through its roles.
- **When the body of a general definition is checked**:
  - Once for each definition: everything that does not depend on the arguments is checked then, including keywords, signatures (number of objects, parameter names and types), `$` and `%` names (resolved where the definition is written), host references, clashes of local names (9.4), and whether roles declared with a kind are used correctly: when the role t is declared `"instance"`, `?t::net` in the body is an error. Signatures are checked when the text is read (7.1); names can be resolved only after the whole level has been read, so the checks that

need names are made when the level takes effect. Both are made only once for each definition, however many times it is applied.

- For each application: what depends on the arguments, such as the objects that roles without a declared kind receive, or whether the symbol has a pin, is checked at each application and reported at the application, pointing also to the statement in the general definition.
- Extent of the errors: a statement of the body found wrong (a local name that clashes included) is reported once, at the definition, and is left out of every expansion; the rest of the body works as usual (the body is tolerant, 9.1). A wrong role list or parameter list (repeated names, clashes with standard sub-object names, unknown kind names) makes the whole general definition fail, and with it every application that uses it.
- A general definition that is never applied is checked too. These checks do not depend on applications, so they find errors in the unused definitions of a package.

6.10 Placeholders

- `?name` appears only in the body of a general definition, and the name must be in the role list or the parameter list; otherwise it is an error when the text is read.
- Where an object goes, it is a role, such as `?t` or `?t::net`; where a literal goes, it is a parameter, such as `min=?gap`, `[?w, 2]` and `(?x, 0)` (9.4). A role where a literal goes, or a parameter where an object goes, is an error when the text is read. Role names and parameter names are in one table and cannot repeat.
- A role placeholder may have a path, such as `?t::net`; a parameter placeholder has none.

6.11 Subscripts

Written	Stands for
`&D<3>`	The same as `&"D<3>"`
`&D<7:0>`	A list: `D<7>`, `D<6>`, …, `D<0>`
`&D<1,2:3>`	`D<1>`, `D<2>`, `D<3>`
`&D<0:6:2, 9>`	`D<0>`, `D<2>`, `D<4>`, `D<6>`, `D<9>`
`&D[3:0]`	`D[3]`, `D[2]`, `D[1]`, `D[0]`
`#XI0<3:0>`	Four instances
`#X1:D<7:0>`	Eight terminals of X1
`#:D<7:0>`	Eight ports of this level
`&"D"<1:0>`	`D<1>`, `D<0>`
`&"D<7:0>"`	The node whose name is `D<7:0>`: nothing in quotes is parsed

- A netlist has no buses: when tools such as Virtuoso write a netlist, they split a bus into single bits, such as `D\<3\>` in a Spectre netlist; bus ports of subcircuits and iterated instances are split too. A subscript is a short way of writing one or several such names. It only builds names and does not rely on the host knowing buses, so its meaning is unique.
- **Form**: a host name, quoted or not, followed directly by `<…>` or `[…]` that holds items separated by `,`. Each item is `i`, `a:b` or `a:b:s`, and the forms can be mixed. The numbers i, a, b and s are unsigned decimal integers, and s is at least 1. There is no white space between the name and the bracket; inside the brackets white space and comments are allowed (3.5).
- **Expansion**: the items are expanded in the order written and joined; repeats are kept. `a:b` runs from a to b, and `a:b:s` in steps of s without passing b: `<7:0>` is 7, 6, …, 0, and `<0:7:2>` is 0, 2, 4, 6. The step comes last, as in Virtuoso and Xschem, not as in MATLAB's `a:s:b`. A generated name is the name before the bracket, the opening bracket, the decimal number without leading zeros, and the closing bracket, in the kind of bracket written: `&D<07>` is `D<7>`, and `&D[3]` is `D[3]`.
- **One or a list**: one item that is a single number gives one reference: `&D<3>` is `&"D<3>"`. Anything else gives a list, which is a reference to a set, such as `&D<3:3>` and `&D<3,5>`. A list in a set is spread one level (5.2); as the only object of a statement it is first split one level (5.2.1), so `#XI0<3:0> @ALIGN("y")` aligns four instances; in a slot that takes a single object it works member by member; in the net slot of WIRE it is taken whole and makes a bus (13.1).

- **Where**: host names after `#` and `&`, and terminal names and port names in `#` references. RNL names (after `%`, `$` and `?`), the terminal names of usages (which are role names of definitions) and sub-object names take no subscript; a `<` or `[` right after them is an error.
- **No point right after a name**: a `<` or `[` right after a name always starts a subscript, and when what follows does not fit the form, it is a syntax error; it is never read as a point. To write a point after a name, separate them, such as `&D (1, 2)`.
- **One expansion**: a name has at most one subscript, so a name such as `A<1>[2]` is written in quotes, `&"A<1>[2]"`. At most one subscript in a reference expands to a list: `#XI0<3:0>:a` and `#X1:D<7:0>` are allowed, `#XI0<1:0>:D<1:0>` is an error, and two subscripts that are each a single number are allowed, such as `#XI0<3>:D<1>`. When a reference that expands to a list is followed by `::`, the sub-object is taken for each member, such as `#XI0<3:0>::name`.
- **Nothing in quotes is parsed**: `&"D<7:0>"` refers to the node named `D<7:0>`; `&"D"<7:0>` expands. When a name in the netlist contains `<` or `[` that do not mark the bits of a bus, quote it to refer to it as it is.
- **The names compared are those the adapter reads** (3.6): `D\<7\>` in a Spectre netlist, where the backslash is Spectre's escape, is read by the adapter as `D<7>`, so `&D<7>` refers to it.
- Each member is resolved on its own. A member that is not found is a failed object, handled strictly or tolerantly as the slot that receives it says (7.3).
- Strings in parameters are not expanded, such as the `terminal=` of PIN (9.2.4) and the condition `name=` of a selection.
- Not accepted: the repetition of Virtuoso, `<*2>A` and `A<0:2*2>`, and its juxtaposition `A,B` (sets can already be written); `D[0..7]` of KiCad and Xschem, whose members are D0 … D7 without brackets.
- Iterated instances (such as `I0<3:0>` of Virtuoso: several host instances drawn as one) are written as bundles: `#XI0<3:0> @BUNDLE` (11.8).

6.12 Terminal names: DEVICE

```
* sky130 transistors, defined in the PDK and not in this netlist
* @RNL(type="header") {
*   @VERSION("0.3");
*   @DEVICE(name="X*", model="sky130_fd_pr__nfet_*", terminals=["d", "g", "s", "b"]);
*   @DEVICE(name="X*", model="sky130_fd_pr__pfet_*", terminals=["d", "g", "s", "b"])
* }
XM1 out in 0 0 sky130_fd_pr__nfet_01v8 w=1 l=0.15
XM2 out in vdd vdd sky130_fd_pr__pfet_01v8 w=2 l=0.15
* @RNL { #XM1:d @CURRENT("in") /* XM1 is a sky130 NMOS; its terminal 1 is named d */}
.end
```

- **RNL names the terminals.** `@DEVICE` names the terminals of a class of devices: `terminals` lists the names in the order of the nodes on the netlist line. It only gives names to the terminal numbers of the host, and changes neither the order of the terminals nor the interface of a subcircuit; it is a display alias, which Principles 2.2 allows.
- **Only in the header record** (2.8): DEVICE written anywhere else is an error. The header record of a package file counts (16.2). An inline package cannot hold DEVICE; write it in the header record of the file.
- **Matching**: `name` is compared with the instance name and may use the wildcard `*` (any number of characters). `model` may be left out; it is compared with the model name or subcircuit name written on the instance line (in Spectre, the master of the instance, such as `resistor` for a primitive), and may also use `*`. Both are compared by the case rules of the host, and a DEVICE matches an instance only when both match. The conditions of selections that compare names take `*` by the same rules (10.1).
- **When several match, the later one counts**: the defaults of the adapter come first, then the DEVICE statements of the packages used by the level (in the order of the packages' positions, 16.2), then those in the header records of the outer files, then those of the included files; within one header record, in the order written. So what a file writes overrides its packages, packages override the adapter, and specific statements should be written after general ones. The DEVICE statements of all packages come before those of the files, wherever their USE-PACKAGE is written: the header record is always at the start of the file, so placing a package at its USE-PACKAGE would let it override what the file itself writes. A DEVICE that gives `model` is not ranked as more specific; there is no separate rule of specificity.

- **Scope**: DEVICE in the header record of a file applies to that file and the files it includes; an included file can change only itself and the files it includes in turn (6.1). DEVICE in the header record of a package file applies to the levels that use the package (16.2). For example, a package for sky130 can hold DEVICE statements, symbols and the rules that choose the symbols, and a user who writes `@USE-PACKAGE(name="sky130-rnl")` gets both terminal names and symbols.
- **Defaults of the adapter**: the standard element definitions of each dialect, which count as written at the outermost level. They need not be written in the file, and a tool may write them out. Tools for the same dialect must give the same definitions. They are the defaults of the version of the file in which the instance line stands (16.1), as a DEVICE in the header record of that file is. For the SPICE family:

```
@DEVICE(name="M*", terminals=["d", "g", "s", "b"]);
@DEVICE(name="Q*", terminals=["c", "b", "e", "s"]);
@DEVICE(name="J*", terminals=["d", "g", "s"]);
@DEVICE(name="Z*", terminals=["d", "g", "s"]);
@DEVICE(name="D*", terminals=["a", "k"]);
@DEVICE(name="R*", terminals=["p", "n"]);              /* and one each like it for C, L, V, I, B,
F, H, W */
@DEVICE(name="E*", terminals=["p", "n", "cp", "cn"]);  /* and one each like it for G, S */
@DEVICE(name="T*", terminals=["p1", "n1", "p2", "n2"]); /* and one like it for O */
@DEVICE(name="U*", terminals=["a", "b", "ref"])
```

  Instance names in Spectre do not start with an element letter, so the defaults are matched by `model`, the master of the instance. For the primitives of Spectre:

```
@DEVICE(name="*", model="resistor", terminals=["p", "n"]);      /* and one each like it for
capacitor, inductor, vsource, isource, ccvs, cccs, bsource, port */
@DEVICE(name="*", model="diode", terminals=["a", "k"]);
@DEVICE(name="*", model="vcvs", terminals=["p", "n", "cp", "cn"]); /* and one each like it for
vccs, relay */
@DEVICE(name="*", model="transformer", terminals=["p_plus", "p_minus", "s_plus", "s_minus"])
```

  - `switch` has one terminal for each of its nodes, named as Spectre numbers its positions: `t0`, the common terminal, then `t1` to `tn`, the throws.
  - An instance of a model card, whose master is the name of a `model` statement, takes the names of the family of the master of that model. Its type, for the condition `type` (10.1), is written in the words of SPICE:

Family	Masters of the model	Terminals	Type
MOS	`bsim1`, `bsim2`, `bsim3`, `bsim3v3`, `bsim4`, `bsim6`, `bsimbulk`, `bsimsoi`, `bsimcmg`, `bsimimg`, `b3soipd`, `mos0`, `mos1`, `mos2`, `mos3`, `mos9`, `mos11`, `mos705`, `mos902`, `mos903`, `psp102`, `psp103`, `psp104`, `pspnqs102`, `pspnqs103`, `ekv`, `hisim`, `hisim2`, `hisimhv`, `hvmos`, `bta_soi`, `ucsd_hvmos`	`d g s b`	`nmos`, or `pmos` when the `type` of the model is `p` or `pmos`
Bipolar	`bjt`, `bjt500`, `bjt503`, `bjt504`, `bjt504t`, `bjt3500`, `bjt301`, `vbic`, `hicum`, `hicum0`, `hicum2`, `mextram`	`c b e s`	`npn`, or `pnp` when the `type` of the model is `pnp`
Diode	`diode`, `juncap`, `juncap200`	`a k`	`d`
JFET	`jfet`	`d g s`	`njf`, or `pjf` when the `type` of the model is `p` or `pjf`
Resistor	`resistor`, `rdiff`, `r3`	`p n`	`r`
Capacitor	`capacitor`	`p n`	`c`
Inductor	`inductor`	`p n`	`l`
Relay	`relay`	`p n cp cn`	`sw`

- **Subcircuits defined in the netlist** (including those in library files brought in by include or lib): the port names are those of the definition, which is a fact of the host (Principles 2.2). DEVICE does not change them; a DEVICE that matches such an instance with different names gives a warning. DEVICE names only devices without terminal names: primitives, and subcircuits that the netlist does not define (PDK devices), for which the user, or an LLM, writes the DEVICE.

- **Number of names**: when an instance has more nodes than `terminals` has names, the extra terminals have numbers only; when it has fewer, a reference to an extra name fails.
- **Case**: the terminal names given by DEVICE are compared by the rules of the host (6.4).
- Numbers always work, and so do the port names of subcircuits and `#:port`.

7 Failures and diagnostics

7.1 Two phases of checking

- **Reading**: the lexical structure; the grammar; the signatures of the built-in keywords, that is, the number of objects, their kinds and the parameters (including the shape of arrays); and whether each `?name` in the body of a general definition is in its role list or parameter list and stands in a place that fits it (6.10). Signature errors in the body of a general definition that do not depend on the arguments are also reported at this point (6.9).
- **Taking effect**:
  - after the whole level has been read, names and host references are resolved, and the kinds of the objects that names refer to are checked;
  - the names in the body of a general definition are resolved once for each definition, not again for each application (6.9);
  - the number of objects, the role names, the kinds declared for the roles and the parameters of AS and ATTACH are checked (9.4);
  - each application checks the parts of its expansion that depend on the arguments;
  - once symbols are assigned, it is checked whether the terminals that wires and constraints connect to have a pin in their symbol, and whether the usages that PLACE, ORIENT and constraints apply to, or that wires connect to, have a symbol; PLACE and constraints applied to a usage drawn at the point of its main object fail (12.1).

7.2 Failed objects

- A statement that fails produces a failed object, which carries a diagnostic: the cause, the location and the upstream cause.
- A failed object keeps its place among the members, and the other members keep theirs. So when the host netlist has no M9, `{#M1 #M2 #M9} @ALIGN("y")` fails as a whole; it does not become an alignment of M1 and M2 only.
- When a name is bound to a failed object (including a definition that fails when it is read), the statements that refer to the name receive the failed object too. Statements that depend on a failed description fail with it, whatever their order (Principles 2.9); these are errors as well, and their diagnostics state the upstream cause (7.6).
- Binding a name a second time to another object is an error, and the name keeps its first valid binding; only when all its bindings fail is the name bound to a failed object (6.6).

7.3 Strict and tolerant inputs

- The signature declares for each input whether it is strict or tolerant:
  - strict: when the input contains a failed object, the whole statement fails;
  - tolerant: the failed members are removed and a diagnostic is recorded; the rest takes effect as usual.
- By default, absorbed inputs are strict; enclosed inputs (definition bodies) and committed inputs (`@RNL`) are tolerant (5.4).
- The arguments of AS and ATTACH are strict; the statements of the expansion are tolerant: each one that fails is discarded by itself.
- In pipes and nesting, a failure affects only the descriptions that depend on it (5.5): when a strict input contains a failed object, the statement fails, but statements nested in it that do not depend on the failure take effect as usual; when a statement works once for each member, each member is one description and fails on its own.

7.4 Recovery

- A syntax error makes the whole member in which it occurs fail, up to the next `;` or the end of the enclosing set.
- When brackets do not match, the whole record fails.
- When a record is cut off before it is complete, the whole record fails, and the diagnostic says that the record is incomplete (2.4).
- Recovery from an error never goes beyond its record: the other records are read as usual (Principles 2.9).

7.5 Conflicts are not failures

- Conflicts are handled after the whole level has been read, by the defined methods: override by key (8.4), the appearance queues (8.3) and the resolution of geometric facts (8.5).
- A conflict for which no method has been defined yet makes the whole interpretation fail (1.5 item 3; 7.6). This version has no such case: a cycle of sets fails at the statement that closes it (5.2.2).

7.6 Diagnostics

- **Checking and interpreting are one process.** An interpreter reads the input and gives the resulting scene together with one set of diagnostics; used only for checking, it is also called lint. This standard defines only this one set of diagnostics. The local checks that an editor makes for immediate feedback, such as syntax highlighting, are not diagnostics in this sense: this section does not apply to them, and they cannot replace the diagnostics.
- **Items.** A failure applies to an item: a field of a record head, a statement, a record, a file, or the whole interpretation. Which one is said where each failure is defined (see the extent of a failure below).
- **Three severities**:
  - **Error**: an item fails, whether the cause lies in the item itself, in another item, or upstream in something it depends on. Errors include the basic errors of Principles 2.4 (syntax errors, unknown words, a mismatched signature, a reference not found, a wrong kind, self-contradiction, an unsupported version), the further failures defined in this standard (such as ORIENT applied to a usage without a symbol, or a terminal without a pin in its symbol; 12.1, 9.2.4), and the failures that follow from them (7.2).
  - **Warning**: the item is valid and takes effect as usual, unless this standard says otherwise (as for a fact discarded because its check did not finish, 8.5), but may not be what the author meant, such as an empty set written out for a slot that takes single objects, as in `{} @PLACE(1, 2)` (5.2.1); an unnamed definition written directly as a member of a record, not nested in a statement (9.1); a role of a generation rule that cannot reach the levels below (10.5); an unknown field in a record head (2.8); a package version replaced by another (16.2); a wire connected to a terminal that is not on the net the wire names (13.1); a geometric fact discarded because the check for it did not finish within the time limit of the solver (8.5).
  - **Hint**: a common override, hiding or resolution that may also be a mistake, such as a lower level, or a package used in it, hiding a definition of an upper level or of the packages at the root (6.8); a package of a later USE-PACKAGE hiding one of an earlier USE-PACKAGE (16.2); a geometric fact discarded in conflict resolution (8.5); the latest version of a package used because none is written (16.2); a main file without a declared version, read with the tool's default version (16.1); a record of the solver whose numbers are out of date (5.9).
- **The extent of a failure** is given where each failure is defined: a field (a known field of a record head written wrongly is treated as not written, 2.8); a statement (most cases); a record (unmatched brackets, a record cut off, a `scope` target not found; 7.4, 2.4, 2.6); a file (an unsupported version, 16.1); the whole interpretation (a conflict for which no method has been defined yet, 1.5 item 3: the interpretation stops and nothing is drawn; a later version could make such a case tolerant; this version has none).
- **Requirements**:
  - Errors must be reported. Warnings and hints are those listed in this standard, and implementations should report them.
  - An implementation may add checks of its own, but only as warnings or hints; they must not make an item fail, or different implementations would draw different results.
  - An implementation must let the user see the diagnostics; how they are presented is up to the implementation (Principles 2.9).
  - An implementation may be stricter, such as stopping or rejecting the whole input (Principles 2.9), but this must not be its default. It may let the user raise or lower the severity of warnings and hints; errors cannot be lowered, since an error is a failure.
  - Conformance tests check errors (their location and the extent of the failure) and the resulting scene; warnings may be optional expectations; hints are not tested.
  - Note: warnings guess at what the author meant and will change with experience; making them requirements would freeze those guesses into the standard. Principles 2.4 also requires only the basic errors.
- **Content**: a diagnostic gives

  - the location: the file, the line and column or range; which record, statement, field or reference (Principles 2.8);
  - the cause: what was expected and what was received, such as "in_p takes an instance, got a net" (9.4); when a name is not found, the closest name;
  - the upstream cause: each statement that fails as a consequence states its upstream cause, and a presentation may gather such statements under the root cause, so that one typo does not produce dozens of errors across the drawing. When one rule fails on many objects, the failures may be combined into one diagnostic that lists the objects, such as when a package writes `W/L=${w}/${l}` for every MOS and some instances have no `w`;
  - the origin: for a statement expanded from a general definition, both the application and the original statement in the definition; for one produced by a rule, the rule and the level where it was evaluated; for one in a package, the name, version and file of the package (16.2).
- The wording and language of diagnostics are not specified. This version has no standard diagnostic codes; implementations may number their diagnostics, and the numbers do not define classes of tolerance.
- **Packages**: in a document that uses a package, errors in the content of the package are reported as usual and attributed to the package; warnings and hints about the content of the package are given only when the package itself is checked. Note: users cannot change a package, and broad rules that its author wrote on purpose would otherwise flood them with warnings.
- A file cannot suppress diagnostics or ask for a stricter mode.
- Unknown keywords, parameters and values are errors, like syntax errors (Principles 2.3): the statement fails (16.4). In this version the only content that is recognized but not interpreted is an unknown field of a record head, which gives only a warning (2.8).

8 Taking effect

8.1 A level takes effect after it is read completely

- The interpreter parses the complete host netlist and reads all records; then each level takes effect in the phases of 8.2.
- Within a level, the order of statements matters only as 1.5 item 2 says; it does not affect how names and references are resolved (Principles 2.9).
- Which of two levels overrides the other is decided by the hierarchy of levels, not by the order of the text (Principles 2.9; 8.3).

8.2 Phases

1. **Reading**: the lexical structure, the grammar and the signatures.
2. **Names and definitions**: `$` names, `%` names and host references are resolved; the bodies of DEFINE-SYMBOL and DEFINE-STYLE are evaluated.
3. **Creating objects**: the scene objects written explicitly (points, wires, labels, bundles, graphics, constraints, usages) and the sets are created, and the bodies of general definitions are expanded. Selections are not expanded here.
4. **Symbol queue**: SET-SYMBOL takes effect and assigns symbols to the objects of step 3. Only now are the selections of the rules in this queue evaluated, in each level.
5. **Generation one by one**: AS and ATTACH whose objects are the result of a selection, and TEXT attached to the result of a selection (10.5, 13.3), are evaluated once in each level, in the order: packages, the levels above, this level.
6. **Symbols for generated objects**: the same symbol queue is run once more for the new objects. Generation sees the symbols as they were before generation; if an expansion changes the symbol of an existing object, the generation is not recomputed.
7. **Geometry**: PLACE, ORIENT and constraints are taken in one by one in the order of resolution, and one that conflicts with those already taken in is discarded (8.5, 14.8); then the layout is solved. This step needs the positions of pins and the boxes of symbols (9.2.3), so it comes after symbols are assigned; PLACE, ORIENT and constraints applied to a usage without a symbol fail here. For a usage drawn at the point of its main object (12.1), the position is computed here from the main object, and PLACE and constraints applied to the usage fail. Objects without any position or constraint are placed by the solver, which writes their positions back (14.8).
8. **Style queue**: SET-STYLE takes effect, with the two tiers of 8.3. Only now do the style properties that the graphics of a symbol leave unwritten, and the colours of `fill=`, get concrete values from the style of the owning object.
9. **Rendering**.

- These phases are the internal order of 1.5 item 4: a description reads the state from before its phase began; neither its own effects nor those of the other descriptions of the same phase are included.
- A set is evaluated in the phase that uses it (5.2): the same set contains no generated objects when it is used in step 5, and contains them when it is used in step 8.
- Selection conditions read only information fixed in earlier phases:
  - host information, the kinds of objects, usages, the current directions of terminals and explicit wires are fixed in step 3;
  - a selection in the symbol queue cannot select by the symbol an object uses;
  - generation rules can see symbols.
- Whether a terminal has a pin in its symbol is checked after step 6 (7.1).
- The methods of solving and rendering are not specified by this standard; only their inputs are (Principles 1).

8.3 Appearance queues

- There are two queues: the symbol queue takes SET-SYMBOL, and the style queue takes SET-STYLE.
- For one level, each queue consists of these groups, in this order:

1. the rules of the root packages: the standard package first, then the packages of USE-PACKAGE in header records (16.3); when a package is used again with USE-PACKAGE in this level or a level above (in a file level or a subcircuit), it is placed at the position of that USE-PACKAGE instead (16.2);
2. the symbols that subcircuits assign to their own `#`, that is, their block symbols (symbol queue only; 9.2.6);
3. the rules of the levels above, from the outermost inward;
4. all SET-SYMBOL or SET-STYLE statements written in this level, both rules and statements on individual objects;
5. SET-SYMBOL or SET-STYLE written into this level from outside with `scope`, those from outer levels later (2.6).

- Within each group, the order is the text order. The content of another package is placed at the position of its USE-PACKAGE (16.2): a package used in a level above goes with that level, and one used in this level goes at the position of the USE-PACKAGE in the text of this level. SET-SYMBOL and SET-STYLE produced by generation one by one belong to the group of the rule that generates them, at the position of that rule.
- **Two tiers**:
  - Direct: SET-SYMBOL and SET-STYLE that name this object directly, that is, whose input is its handle, its name or a `::` path to it, or a set written in `{…}` that lists it as a member (also when that set is named). Style parameters written when the object is defined (9.2.5, 9.3) count too, as if a SET-STYLE had been written for the object at the statement that creates it.
  - Passed on or selected: what a member receives through a selection or through generation one by one; what an owning object passes to the graphics of its symbol (9.2.5); what an object passes to the texts attached to it (11.3); what a usage passes to its symbol and to the drawable objects created in its body, and so on down through nested usages, but not to the members that fill its roles (12.1 item 4). A fact passed on keeps the position of its original statement.
  - For each property (in the symbol queue, the symbol), the direct facts are considered first and the last one by position is taken; if there is none, the facts passed on or selected are considered and the last one by position is taken; if there is none of either, the renderer's default is used (9.3). Positions follow the order of the queue above.
  - So a symbol or style that an editor changes for one object is not overridden by rules written later; rules, packages and themes still govern the objects that were not changed individually.
- The `type` and `author` of a record head do not affect the order (2.8).
- A symbol is replaced as a whole; styles merge by property. Applying a style is the same as writing each property it defines once, so a style that defines only `linewidth` changes only the line width.
- Rules apply to the level where they are written and to the levels below, not to the levels above; the rules of a package apply to every level that uses the package (10.4). They are evaluated once in each level, looking only at the objects of that level; a rule whose selection has `recursive=false` takes effect only in the level where it is written (10.4).

8.4 Override by key

Facts with a key (5.8) override one another by key: the later one takes effect (Principles 2.9). Overriding is a legitimate way of describing, not an error; whether to give a hint is up to the implementation. PLACE and ORIENT are geometric facts and are resolved by 8.5 (5.8).

8.5 Conflicts

Geometric facts together may have no solution even when no two of them contradict each other, such as `{#M1 #M2} @ALIGN("y")`, `#M1 @PLACE(y=0)` and `#M2 @PLACE(y=10)`. The following decides which facts are discarded.

- **Geometric facts** are the four atomic constraints (ALIGN, ORDER, DISTRIBUTE, DISTANCE), the orientation relation ORIENT-RELATION (14.9), each coordinate of PLACE, the `dx` and `dy` of TEXT (13.3; on the same axis these and PLACE replace one another, which is not resolved here) and ORIENT. Symbols (SET-SYMBOL) are not among them; they are fixed before geometry (8.2). Nor are the implementation's own criteria (devices do not overlap, a grid and so on): they are the solver's own considerations and rank after all the descriptions of the language. If the descriptions force devices to overlap, they overlap.
- **Units of resolution**:

  - One atomic constraint statement is one unit, discarded as a whole in a conflict (ORIENT-RELATION too); it is never kept for only some of its members. This agrees with failures: the members of an atomic constraint are strict inputs (7.3). When a slot that takes a single object receives a set and the statement works once for each member, each of these descriptions is a unit (5.2.1).
  - One PLACE is two units, one for x and one for y, which can be discarded separately. The `dx` and `dy` of TEXT are likewise one unit each; they first replace, or are replaced by, a PLACE of the same text on the same axis by position (13.3), and those left take part in resolution.
  - ORIENT is one unit: the orientation of this object equals a value. Without ORIENT there is no constraint: the solver chooses, and writes the orientation back (14.8); a renderer that has no solution for it draws `"R0"`. An orientation is a variable with eight values from which the positions of the terminals are computed; conflict checking can try the orientations directly and need not rewrite them as relative positions of terminals.
  - A general definition is not a constraint: each fact expanded from a usage is a unit, ordered and discarded on its own.
- **Facts that contradict themselves**: each fact is first checked on its own. A fact that has no solution on its own, such as a DISTANCE from an object to itself with `min=4` (9.5), contradicts itself: it fails (7.6) and takes no part in resolution. Only the facts that have a solution on their own are ordered and taken in.
- **Order of resolution**: the geometric facts are put in one order. They are ordered first by `priority`, `"high"` above `"normal"` above `"low"` (14.8); with equal priority, by position, the later one first. Positions:
  - by level and text order: the root packages (16.3), the content of the levels above, the statements of this level, the statements written into this level from outside with `scope` (those from outer levels later, 2.6); the content of another package is at the position of the USE-PACKAGE nearest to this level (16.2);
  - facts expanded from a general definition take the position of the AS or ATTACH statement, and facts generated one by one take the position of the generation rule (10.5);
  - when one statement produces several facts, those of an expansion follow the order of the definition body, those of a statement that works once for each member follow the order of the members, and for PLACE x comes before y; in each case the later one comes first. The segments of a pipe are descriptions of their own, and the later segment is written later (5.5): the facts of `#R1 @PLACE(0, 0) => @PLACE(5, 5)` come in the order y = 5, x = 5, y = 0, x = 0, and in `#M1 @PLACE(0, 0) => @ORIENT("R90")` the ORIENT comes before the PLACE.
- **Resolution**: the facts are taken in one by one in the order of resolution; after each one, the facts taken in so far are checked for a solution, and if there is none, this fact is discarded. The result is unique: it is the consistent subset that is lexicographically greatest in this order. Facts have a solution when there are real coordinates, and for each orientation that is free one of its eight values, that satisfy all of them within the tolerance (11.7); integer points are only an aim of the solver.
- **Time limit**: a check for a solution can take long, because some facts hold in one of several ways that must be tried in turn: an ORDER or a DISTANCE that gives only an axis, the upper bound of a DISTANCE between sets, the centre of a set of three or more members, and a free orientation. An implementation may therefore limit the time it spends on checking. The limit is an option of the solver that the user can set (14.8); this standard does not fix its value. When the check for a fact does not finish within the limit, the fact is discarded, with a warning that says that it was not decided (7.6); the result may then differ from the one defined above, and from one run to the next. Without a limit, and whenever no check reaches it, the result is the one defined above.
- The facts taken in are all hard for the solver; the discarded ones have no further effect. Facts with priority `"disabled"` are removed before ordering (14.8).
- Note: this section fixes which facts are kept, not how they are found. An implementation may check a whole run of facts at once and look for the fact to discard only when the run has no solution, or check only the facts that share objects with the new one, as long as it keeps the same facts.
- A solving implementation must resolve conflicts as this section specifies, and its result must satisfy all the facts taken in. An implementation that does not solve draws only the explicit geometry and is not required to check conflicts between statements (Principles 2.4).
- Discarding is a legitimate resolution and gives a hint, which an implementation may raise to a warning. When given, the hint names every set of facts taken in that has no solution together with the discarded fact and from which no fact can be left out, each fact by the statement that wrote it (7.6). Each statement whose facts are in such a set also gets a hint that names the facts discarded because of it, so that the author sees which statements

lost their effect through it. When finding all of these sets does not finish within the time limit, the hint says that its list is not complete. Discarding a `"low"` fact is expected, and its hints may be left out (7.6): an implementation need not list the sets of facts that it conflicts with, nor give the statements in them a hint.

- A conflict is not a failure (7.5). Conflicts of high-level meaning between statements, and agreement with the netlist, are not required to be checked (Principles 2.4).

9 Definitions

9.1 Common rules

- There are three kinds of definition: DEFINE-SYMBOL makes a symbol, DEFINE-STYLE makes a style, and DEFINE makes a general definition. Symbols and styles are the specific definitions (1.4.4). Each of them takes a set as its body; the statements of a body are tolerant one by one (7.3).
- A definition is an object. It is named with `$name =` or `%name =` (6.6) and is used as an object of a keyword, in the slot that the keyword sets for it (5.6).
- Committing a definition only publishes its name. A definition draws nothing itself and has an effect only when it is used: a symbol when SET-SYMBOL assigns it, a style when SET-STYLE assigns it, a general definition when AS or ATTACH applies it.
- A definition written directly as a member of a record, not nested in a statement, and without a name is never used, and gives a warning. An anonymous definition nested in a statement is used as usual.
- **Inclusion**: a reference to a definition of the same kind in the body of DEFINE-SYMBOL or DEFINE-STYLE includes its content at that place. Members with a key that are written later override those written earlier; members without a key accumulate. There is no `base=` parameter.
- General definitions do not include one another; they are combined by AS and ATTACH in their bodies. Derivation and extension work the same way; there is no inheritance relation (9.4).

9.2 Symbols: DEFINE-SYMBOL

9.2.1 Evaluation

- A symbol is a value: its body is evaluated where it is defined, and the statements in it produce pins and graphics in the local coordinates of the symbol. These become the content of the definition and are not committed to the scene.
- The order of the members is the drawing order.
- Local coordinates have the same unit and axes as the scene (x to the right, y up), and (0, 0) is the origin of the symbol (6.5); this version has no scaling (11.7). A point in the body must give numbers on both axes; `_` cannot be used.

9.2.2 Statements in the body

- The body can contain only graphics, pins, named local points (such as `%tip = (0, 0.7)`), inclusions of other symbols, usages applied only for drawing (with ATTACH, or AS, 9.4) together with the SET-SYMBOL and ORIENT written on them (9.2.7), and TEXT attached to the owning object `?`, that is, the object drawn with this symbol, together with the PLACE that positions such a text (13.3).
- Any other statement fails and is dropped (the body is tolerant); host references fail too.
- Pins can be written only in a DEFINE-SYMBOL body. Graphics can also be written in the scene, in scene coordinates; there they are annotations (13.4).

9.2.3 Parameters

- `box=[x1, y1, x2, y2]`: the box of the symbol, in local coordinates and axis-aligned; it is transformed with the orientation of the instance. The language itself does not use it; viewers usually use it as the selection box and solvers as the boundary for keeping symbols apart, and how it is used is up to the implementation. When it is not written, it is the bounding box of all the graphics in the symbol except text, since the size of text depends on the font and the renderer.

9.2.4 Pins

```
$RES_BOX = @DEFINE-SYMBOL {
  %p = (0, 3) @PIN(terminal=1, label="P", direction="up");
  %n = (0, -3) @PIN(terminal=2, label="N", direction="down");
  @LINE {%p (0, 1.5)};
  @LINE {(0, -1.5) %n};
```

```
  (0, 0) @RECTANGLE(width=1.2, height=3)
}
```

- A pin is a point bound to a terminal. Once named, it can be used as a point, as the leads above start at `%p` and end at `%n`.
- Pins should be placed on integer points (11.7): when the instance is placed on an integer point, its pins and the wires connected to them fall on integer points too. Other positions are not an error; an implementation may give a hint. Graphics in a symbol are not subject to this.
- `terminal=`: a terminal of the object drawn with the symbol, written as its number, a terminal name given by DEVICE (6.12) or a port name of the subcircuit; for a usage, a role name of its definition or the number of the role (6.4). In one symbol a terminal has only one pin: pins are keyed by terminal.
- **Bus pins**: `terminal=` can take an array, giving one pin for several terminals, such as a bus port of a subcircuit: `(-4, 0) @PIN(terminal=["D<3>", "D<2>", "D<1>", "D<0>"], label="D<3:0>", direction="left")`. The array has at least one element, and each element is a number or a name. Strings in parameters do not expand subscripts (6.11), so the names are written one by one.
  - All these terminals are drawn at this pin: a wire or constraint connected to any one of them is connected to this pin; when a wire connects to any one of them, all of them count as wired (10.2).
  - When the same terminal is written in several pins, the pins override by terminal: the later pin gets it, and the earlier pin keeps its other terminals (9.2.7).
  - Several pins at the same point are not a substitute: equal coordinates do not connect (13.1), a wire connects to only one of the pins, the others stay `wired=false`, and the fill rules of the standard package would add labels and grounds to them.
- `label=`: the name of the pin, for display only; it cannot be used to refer to the terminal (6.4). Whether and where it is shown is up to the viewer. It can use text markup, such as `\overline{CS}` (13.6).
- **A terminal that is not drawn has no pin.** A terminal without a pin has no meaning for drawing: a wire or constraint connected to it fails ("the symbol has no pin for this terminal"), and the fill does not select it (10.1). There is no notation for hiding a terminal that has a pin; to leave out the bulk, use a three-terminal symbol without a bulk pin.
- `direction=`: the direction in which a wire leaves the pin: `"right"`, `"up"`, `"left"` or `"down"`. It is written in the local coordinates of the symbol and turns with the orientation (11.2). It is a hint for solvers and viewers, such as the direction in which free routing leaves the pin, the way current runs at the terminal (15.2), or a reference when choosing orientations; it is not a constraint and takes no part in conflicts. It can be left out; then the pin has no direction, and an implementation may infer one from the graphics.
- There is no lead length: leads are drawn in the symbol with LINE.
- There is no electrical type: pins and terminals have no parameter such as input or output, and the netlist has no such information either. Where it is needed, users express it with usages of semantic definitions; an exporter to tools such as Virtuoso maps general definitions to electrical types by its own configuration.
- A pin is only a point: whether a lead is drawn is up to the symbol, and how the point of a pin is marked (a small square, a dot, or not at all) is up to the viewer. So a pin without a lead needs no special notation: write PIN and no LINE. For example, a drawing whose wires connect to a bulk pin that its editor hides is imported as a symbol that keeps the bulk PIN and draws no bulk lead. Ordinary symbols, including those of the standard package, draw leads.

9.2.5 Graphics

Positions are point objects; scalars are parameters. In the table, *point* stands for any point object.

Form	Meaning
`@LINE {point point …}`	Two points make a line segment, more make a polyline; when the last point is at the same position as the first, the figure is closed and the joint is drawn as a corner
`@RECTANGLE {point point}`	A rectangle; the two points are opposite corners, in either order

Form	Meaning
`(0, 0) @RECTANGLE(width=4, height=2)`	A rectangle; the point is the centre, and `width` and `height` are its width and height
`@CIRCLE {point point}`	A circle; the first point is the centre and the second is on the circle
`(0, 0) @CIRCLE(radius=1)`	A circle; the point is the centre
`@ELLIPSE {point point}`	An ellipse; the two points are opposite corners of its bounding box
`(0, 0) @ELLIPSE(width=4, height=2)`	An ellipse; the point is the centre, and `width` and `height` are those of its bounding box
`@ARC {point point point}`	A circular arc from the first point through the second to the third
`@BEZIER {point point point point …}`	A cubic Bézier curve of 3n+1 points: the start point, then for each segment two control points and an end point; neighbouring segments share their end points
`(1, 0) @TEXT("…")`	A text attached to a point: fixed text of the symbol, which cannot change from instance to instance
`? @TEXT("${name}", key="name") => @PLACE(1, 0)`	A text attached to the owning object (the object drawn with this symbol) that reads its attributes; its key is `name`, so for M1 it is `#M1::name`, and a text with the same key in the scene overrides it parameter by parameter (13.3)

- **Two forms**: RECTANGLE, CIRCLE and ELLIPSE are told apart by whether a size parameter is written. With a size (`width` and `height` for RECTANGLE and ELLIPSE, `radius` for CIRCLE), the object slot takes one point, the centre; given a set, it draws one figure for each point (5.2.1). Without a size, the object slot takes a set of two points, and a different number of points makes the graphic fail. A size written only in part is a wrong parameter, and the graphic fails. In the two-point form the points are objects: they can be named or be the positions of pins, and the figure follows them.
- Rectangles and ellipses are axis-aligned; a slanted rectangle is drawn with a closed LINE.
- **Fill**: every graphic except TEXT takes `fill=`. An open LINE, ARC or BEZIER is filled as the region closed by a straight line from its last point to its first, and this closing line is not stroked; a filled ARC is the segment bounded by the chord and the arc. A figure that is filled but not stroked (the gate of a MOS, an arrow head, a junction dot) is written `fill="linecolor", linestyle="none"` (9.3).
- **Arrow heads**: LINE, ARC and BEZIER take `arrow=`: `"none"` (default), `"start"`, `"end"` or `"both"`, such as `@LINE(arrow="end") {%from %to}`. The tip is at the start or end point and points in the direction of the line at that end: for LINE, the first or last segment; for ARC, the tangent of the arc at the end point; for BEZIER, the line from the neighbouring control point to the end point (when the two coincide, the next point that does not). The length of the arrow head is given by `arrowsize` (9.3), and its shape is up to the implementation. The end points are point objects; when constraints move them, the arrow head follows, at any angle, whereas an arrow usage drawn at a point has only eight orientations. WIRE does not take `arrow=`.
- **Style parameters**: graphics can take style properties directly as parameters (9.3): LINE, ARC and BEZIER take `linecolor`, `linestyle`, `linewidth`, `cap`, `join`, `fillcolor` and `arrowsize`; RECTANGLE, CIRCLE and ELLIPSE take the first six; TEXT takes `textcolor`, `size`, `font`, `bold` and `italic`. Those written are the direct style of the graphic, which ranks above the style passed on by the owning object (8.3), so a colour written in a symbol does not change with the theme; those not written come from the style of the owning object, so changing the theme needs no change to the symbol. The fill is written with `fill=`: `"none"` (default), `"linecolor"` (the colour of `linecolor`) or `"fillcolor"` (the colour of `fillcolor`).
- **Not provided**: SVG path strings, POLYGON (use a closed LINE), a separate arrow graphic (for arrow heads at line ends, see above; elsewhere, use a filled closed LINE or an arrow usage drawn at a point, 9.2.7), rounded rectangles, elliptical arcs (approximate them with BEZIER), and images.
- **Text**: TEXT also has `size=` (font size, 11.7), `halign=` (horizontal: `"start"` (default), `"middle"`, `"end"`), `valign=` (vertical: `"baseline"` (default), `"top"`, `"middle"`, `"bottom"`) and `rotate=` (0, 90, 180 or 270; default 0). In a symbol body, a TEXT attached to `?` refers to the attributes of the owning object, and one attached to a point is fixed text; in the scene, a TEXT is an annotation or a label depending on what it is attached to (13.3, 13.4). Text markup is described in 13.6.

- **Baseline**: with `valign="baseline"`, the anchor is on the baseline of the first line, and further lines go downward, as in SVG, PostScript and most editors; `"top"`, `"middle"` and `"bottom"` align the whole text box. The baseline is the default.
- **Direction of text**: the anchor is the text's own position (13.3). The text box is placed beside the anchor according to `halign` and `valign`, and turns about the anchor by `rotate`; a text in a symbol is then transformed with the orientation of the instance, both rotations and mirroring. The characters in the box always read upright: horizontal text from left to right, vertical text from bottom to top, and glyphs are never mirrored. So `rotate=180` is shown as 0 and 270 as 90, with the text box on the other side of the anchor; when the instance is mirrored, the anchor is mirrored as usual and the text box moves to the other side of the anchor in the direction of the mirroring, but the characters are not mirrored. For example, the name of an NMOS has its anchor at (1.5, 1.5) and extends to the right with `halign="start"`; at `"MY"` the anchor moves to (−1.5, 1.5) and the text extends to the left, away from the body; at `"R90"` the anchor moves to (−1.5, 1.5) and the text runs vertically, reading from bottom to top. Texts in the scene do not turn with instances.

9.2.6 Block symbols

Written inside the subcircuit `.subckt ota5t … .ends`:

```
%ota5t = @DEFINE-SYMBOL(box=[-4, -4, 4, 4]) {
  (-4, 1) @PIN(terminal="inp");
  (-4, -1) @PIN(terminal="inn");
  (4, 0) @PIN(terminal="out");
  @LINE {(-4, -4) (-4, 4) (4, 0) (-4, -4)};
  ? @TEXT("${name}", key="name") => @PLACE(-1, 0)
};
# @SET-SYMBOL %ota5t
```

- The symbol that a subcircuit assigns to its own `#` is the default appearance of the instances of that subcircuit, wherever they are: its block symbol. The pins correspond to the ports of the subcircuit by name.
- It is the exception to the rule of Principles 2.5 that definitions take effect only downward: it applies to the levels that instantiate the subcircuit.
- The bus ports of a subcircuit are already split into single ports in the netlist (6.11); a block symbol draws them as one with a bus pin (9.2.4).
- In the symbol queue it comes after the rules of the packages at the root and before the rules of the levels, including packages used in a level, so an upper level can replace it with a rule or with a SET-SYMBOL for an individual instance (8.3).
- No instance instantiates a file level, so `# @SET-SYMBOL` written in a file level has no effect.
- When a subcircuit has no block symbol, how its instances are drawn is up to the viewer; this standard does not specify it. When a subcircuit has several `# @SET-SYMBOL` statements, they are in the same group of the symbol queue, in text order, and the later one takes effect (8.3); an implementation may give a warning.

9.2.7 Variants and fragments

```
$NMOS_BULK = @DEFINE-SYMBOL(box=[-3, -3, 2, 3]) {
  $NMOS;
  %b = (1, 0) @PIN(terminal=4, label="B", direction="right");
  @LINE {(0, 0) %b}
}
```

- Including another symbol in a body and then adding to it or changing it makes a variant: pins override by terminal (a bus pin counts by each of its terminals, 9.2.4), and graphics accumulate.
- A fragment used only in its own level is named with `%`, such as `%MOS_CHANNEL = @DEFINE-SYMBOL {…}`, and another symbol body includes it with `%MOS_CHANNEL;`.
- Graphics have no key: inclusion only adds, and an included graphic cannot be removed. To draw less, define another symbol: split the common part into fragments and include in each symbol the parts it needs, such as `$NMOS3 = @DEFINE-SYMBOL {%MOS_BODY}` and `$NMOS_4TERM = @DEFINE-SYMBOL {%MOS_BODY; %BULK}`.

- **Usages in the body**: a figure that is needed at different positions or in different orientations, such as an arrow head, is written as a usage drawn at a point:

```
$ARROWHEAD = @DEFINE(roles=[["p", "point"]]) {};
$ARROWHEAD_SYM = @DEFINE-SYMBOL {
  @LINE(fill="linecolor", linestyle="none") {(0.4, 0) (-0.4, 0.3) (-0.4, -0.3) (0.4, 0)}
};
$ISRC_DOWN = @DEFINE-SYMBOL {
  (0, 3) @PIN(terminal=1, direction="up");
  (0, -3) @PIN(terminal=2, direction="down");
  @LINE {(0, 3) (0, 1.5)};
  @LINE {(0, -1.5) (0, -3)};
  (0, 0) @CIRCLE(radius=1.5);
  @LINE {(0, 0.9) (0, -0.5)};
  $ARROWHEAD @ATTACH (0, -0.6) => @SET-SYMBOL $ARROWHEAD_SYM => @ORIENT("R270")
}
```

  - The first role is filled with a point of the body, in local coordinates. The usage is drawn with its symbol at this point, its orientation turns about this point, and it is then transformed together with the enclosing symbol; the transformations of nested usages compose outward in turn, so the arrow head turns with the device.
  - Only for drawing: rules outside cannot see into a symbol body, so its symbol is assigned with SET-SYMBOL in the body; a usage without one is not drawn, and a warning is given. The bodies of general definitions are not expanded inside a symbol, and constraints and wires have no meaning there; ORIENT here is internal data of the symbol, takes no part in conflicts and has no priority.
  - The same notation in the scene is a virtual part drawn at a point (12.1), and rules can assign it a symbol; the rules of a house-style package cannot reach inside symbols.
- Pin names and formulas: whether pin names are shown is up to the viewer (9.2.4); a formula is simply a TEXT that refers to attributes (13.6), with no special rules. Text markup is described in 13.6.

9.2.8 Matching symbols to objects

Symbols are not divided into symbols for devices and symbols for usages. Whether a symbol can be used for an object depends on whether its pins match the terminals of the object.

9.3 Styles: DEFINE-STYLE

```
$ink = @DEFINE-STYLE(linecolor="#202020", textcolor="#202020", fillcolor="#ffffff", size=1.2) {};
$accent = @DEFINE-STYLE(linecolor="#1f4fd8", linestyle="bold") {$ink};
$shade = @DEFINE-STYLE(fillcolor="#1f4fd833") {}
```

- The parameters in the brackets are the style's own properties, and they take effect last. The set can refer only to other styles, whose properties are included at that place, later ones overriding earlier ones. A style without a base style is written with `{}`.
- A style takes effect only when SET-STYLE assigns it (11.3). The same properties can also be written directly as parameters of graphics, TEXT and WIRE (9.2.5, 13.1), with values taken by the two tiers of 8.3.
- A style only strokes and colours: position, orientation, shape (the points of a symbol or graphic) and the content of text are not part of a style.
- **Properties**:

Group	Property	Values
Line	`linecolor`	A colour; also the colour of `fill="linecolor"`
	`linestyle`	`"solid"` (default), `"dashed"`, `"dotted"`, `"dash-dot"`, `"double"` (two parallel solid lines), `"bold"`, `"none"` (no line)
	`linewidth`	The line width, in coordinate units (11.7); `0` is a hairline, equally thin at every zoom, and is the default
	`cap`	Line ends: `"butt"`, `"round"`, `"square"`
	`join`	Corners: `"miter"`, `"round"`, `"bevel"`
Fill	`fillcolor`	The colour of `fill="fillcolor"`
Text	`textcolor`	The colour of text; when it is not given, `linecolor`

Group	Property	Values
	`size`	The font size, in coordinate units (11.7)
	`font`	A font name, only a hint; a viewer without this font uses its own
	`bold` , `italic`	Bold or italic for the whole text, `true` or `false` ; the markup `\textbf` and `\textit` can still apply them to parts (13.6)
Other	`arrowsize`	The length of arrow heads at line ends, in coordinate units (9.2.5)

- A line has only five properties: colour, style, width, cap and join. `"bold"` draws twice the line width, and twice a hairline when the width is 0; the dash lengths and gaps of dashed lines and the spacing of double lines are up to the implementation, in proportion to the line width. More values of `linestyle` may be added in later language versions (16.1).
- `"none"` draws no line, and no arrow heads at line ends either (they belong to the line, 9.2.5); `fill="linecolor"` still takes the colour of `linecolor`, so a figure that is filled but not stroked is written `fill="linecolor", linestyle="none"`, and its colour still follows a change of theme. Wires and graphics both take it; a wire with it is an invisible wire, but still a wire (13.1).
- **Colours** are written as CSS hexadecimal colours: `"#RRGGBB"` or `"#RRGGBBAA"`, or the short forms `"#RGB"` and `"#RGBA"`. The alpha comes last: `00` is fully transparent and `FF` opaque; without it the colour is opaque. Lines and fills each have their own colour, so their transparency is set separately, as `$shade` above makes only the fill partly transparent. This version does not accept colour names or `rgb()`. Colours are strings.
- This version does not provide pattern fills, gradients or shadows; visibility, `visible` (showing and hiding is the viewer's business, 16.2); or objects such as themes and profiles (a theme is a package of styles and rules, brought in with USE-PACKAGE, 16.2).
- When no style and no parameter gives a value for a property, the renderer's own default is used; this is not an error (Principles 2.7). The standard package provides a portable default appearance (16.3), which the rules and statements written by the user override.
- Unknown properties are treated like syntax errors (Principles 2.3).

9.4 General definitions: DEFINE

```
$TAIL_MID = @DEFINE(roles=["a", "b", "t"]) {
  %mid = (_, _);
  {?a %mid ?b} @DISTRIBUTE("x");
  {%mid ?t} @ALIGN("x")
};
$CHAIN_ROW = @DEFINE(roles=["source", "{chain}", "load"]) {
  {?source ?chain ?load} @ORDER("x")
};
$GND_BELOW = @DEFINE(roles=[["t", "terminal"]], parameters=[["gap", 2]]) {
  %g = $GROUND @ATTACH ?t::net;
  ?t::net @WIRE {?t %g:p};
  {?t %g:p} @ALIGN("x");
  %g:p ?t @DISTANCE("bottom-to-top", exact=?gap)
};
%c1 = #M1:3(a) #M2:3(b) #M5:1(t) @AS $TAIL_MID;
#R1(source) {#R2 #R3}(chain) #R4(load) @AS $CHAIN_ROW;
# @SELECT-TERMINAL(net="vss", wired=false) => $GND_BELOW @ATTACH(gap=3)
```

- A general definition consists of a role list, a parameter list and a body. In the body, roles and parameters are referred to with `?name` (6.10). What a definition does is decided by its content; see 9.5.
- **Applying**: AS and ATTACH fill the roles of a definition with objects and produce a usage. The two keywords do the same thing and differ only in where the definition is written:
  - AS has the definition last and reads "these objects, as such-and-such"; it is used for meaning and layout, such as `#M1(in_p) #M2(in_n) @AS $DIFFERENTIAL_PAIR` ;
  - ATTACH has the definition first, before the objects written out, and the standard form puts the keyword after the definition, such as `$GROUND @ATTACH &vss` ; it reads "such-and-such, added onto these objects", and is used for virtual parts and the fill (chapter 12, 10.5).

- Choosing the other keyword is not an error, only awkward to read; writing the definition in the wrong place is an error (4.2 item 7). Objects passed in by a pipe are matched to the roles directly, and the two keywords then look the same; choose by what the definition is for.
- How to choose: use AS when the objects together make up the thing, such as a circuit structure, a layout, or a local way of drawing made up of these objects; use ATTACH when the thing is added onto the objects, such as a ground, a port tag, a no-connect mark, a junction dot, a label for each instance, or the fill. The standard sentences are `#M1(in_p) and #M2(in_n) are treated @AS $DIFFERENTIAL_PAIR` and `$GROUND is @ATTACH to &vss` (4.5).

- **Role list**: the parameter `roles`, an array whose items are role names or `[name, kind]` pairs. Both parameters of DEFINE can only be given by name.
  - An item written `"{name}"` is a set role, which takes a whole set; the others are single roles (5.2.1 item 4).
  - A set role given a single object takes it as a set with only that object; an empty set is valid too, and constraints in the body that get too few members hold trivially (chapter 14).
  - A single role given a set applies the definition once for each member, and zero times for an empty set; in one application, at most one single role can receive a set with more than one member (otherwise the statement fails, 5.2.1).
  - When the `?name` of a set role is written inside `{…}`, the set is spread one level, so the ORDER in the example above orders R1, R2, R3 and R4.
  - A set role given a set that has sets among its members takes it as it is; the body decides how to use it.
  - A set role does not correspond to a terminal (6.4).
  - In one application, each role can be marked only once; to give a set role several objects, write them as a set, as `{#R2 #R3}(chain)` above.
  - **Kinds**: an item written `[name, kind]` declares which kind of object the role takes; an item with only a name takes any kind. The kind names are those of the word list (5.3): `"instance"`, `"terminal"`, `"net"`, `"point"`, `"wire"`, `"usage"`, `"bundle"`, `"text"`, `"annotation"` and `"label"`. In a role declaration, `"instance"` also takes bundles, and `"terminal"` the terminals of bundles (11.8); an array means any one of several kinds, such as `["p", ["net", "terminal"]]`. The kind of a set role applies to each of its members, such as `["{members}", "instance"]`. There is no kind for the usages of a particular definition.
  - When a role receives an object of the wrong kind, the whole application fails, and the diagnostic states which role, what it takes and what it received; the same holds when one member of a set role has the wrong kind. When a selection makes the applications one by one, only the application that received the wrong object fails.
  - Note: in a definition with an empty body, nothing checks what fills the roles, so only a declared kind detects a role filled wrongly; in a body that is not empty, each statement of the expansion is also checked by itself (9.5). Declared kinds also let the uses of roles in the body be checked at the definition (6.9).
  - **Optional roles and counts**: a single role must be filled; a set role may be left unfilled, which makes it an empty set. When objects are given in order and there are fewer objects than roles, the remaining roles must all be set roles; when objects are marked with roles, a set role that is not written is an empty set. There is no minimum or maximum count: constraints with too few members hold trivially (chapter 14). A variant of a different shape is written as another definition (see derivation and extension below). Rules of validity such as the same object filling two roles, or expected connections, are covered in 9.5.
- **Parameter list**: the parameter `parameters`, which can be left out. Each item is a parameter name or `[name, default]`.
  - Parameters are passed in the brackets of AS and ATTACH, such as `$GND_BELOW @ATTACH(gap=3)`, by position in the order of the parameter list or by name; when both are used, those by position come first (4.2 item 9). The standard form writes them by name (4.3). A name that is not in the parameter list makes the application fail. When a selection makes the applications one by one, every application uses the same parameters.
  - Wherever the body can contain a literal, a parameter placeholder can be written instead: the value of a parameter, an element of an array, a coordinate of a point. So inside a definition body, the rule that parameters hold only literals (4.1) becomes "literals or parameter placeholders"; after expansion they are literals again. The parameters of AS and ATTACH themselves can only be literals and cannot pass objects; objects come in through roles.

  - A parameter that is not passed and has no default counts as not written where it is used: as a named parameter it is not written, such as `size=?size` in `@TEXT(?label, size=?size)`, after which the text has TEXT's own default size or the size of its style (11.3); as a coordinate of a point it is `_`; in a parameter given by position or in an element of an array it has no defined meaning, and that statement fails. So no marker for required parameters is needed.
  - Parameters are meant for quantities such as numbers and text; a variant that differs in structure should be written as a separate definition; the language does not enforce this. Parameters are only passed into the body; usages cannot be selected by the values of their parameters.
- Role names and parameter names are in one table and must all be different; none of them can be a standard sub-object name (6.5). A parameter cannot be named `priority`, which is a standard parameter of AS and ATTACH (14.8).
- The `%` names bound in the body are local names: each application makes new objects for them, which are referred to from outside through the usage (see below). An object shared by several applications, such as an axis of symmetry, is passed in through a role.
- An application can mark the objects with roles (`#M1(a) #M2(b) … @AS $T`, `$T @ATTACH #M1(a) #M2(b) …`) or give them in the order of the role list (`#M1 #M2 … @AS $T`). When a single role is left unfilled, or more objects are given than there are roles, the whole statement fails; a set role left unfilled is an empty set.
- An application produces one usage, which records the definition and what fills each role, and which can be named (such as `%c1`). Deleting the statement removes the descriptions expanded from it as well. A usage has no key: each application is a separate object (5.8).
- **Expansion**:
  - placeholders are replaced by the arguments and parameter values;
  - local names are replaced by new handles, "usage + local name", written `%c1::mid`; an unnamed usage is shown in diagnostics by the position of its statement;
  - the names from outside the body are not resolved again where the definition is applied (6.9);
  - the result of the expansion is committed at the position of the AS or ATTACH statement, and it takes this position in conflict resolution (8.5).
- Nested usages are reached step by step (`%c1::inner::mid`). A usage can also fill a role of another definition: with a definition of one's own `$OTA = @DEFINE(roles=[["pair", "usage"], ["load", "usage"]]) {}`, `#M1(in_p) #M2(in_n) @AS $DIFFERENTIAL_PAIR => %cm1 @AS $OTA` is an amplifier whose role pair is filled with the usage of the differential pair, and its role load with the current mirror `%cm1` (5.5).
- **Cyclic application**: an application within an expansion that applies a definition already being expanded in an enclosing usage, directly or through other definitions, closes a cycle. That application fails, and its diagnostic lists the whole chain; the rest takes effect as usual. Below, applying `$A` to M1 expands `$B`, whose application of `$A` closes the cycle `$A → $B → $A` and fails; the ORIENT in `$B` still flips M1:

Error example

```
$A = @DEFINE(roles=["x"]) {?x @AS $B};
$B = @DEFINE(roles=["x"]) {
  ?x @AS $A;                /* closes the cycle $A → $B → $A: this application fails */
  ?x @ORIENT("MY")
};
#M1 @AS $A
```

- **Referring to local objects from outside**:

```
$DP_BLOCK = @DEFINE(roles=["in_p", "in_n"]) {
  %pair = {?in_p ?in_n} @GROUP
};
%dp1 = #M1(in_p) #M2(in_n) @AS $DP_BLOCK;
%c1::mid @PLACE(x=20);              /* moves the midpoint of c1 */
{%c1::mid #M9:1} @ALIGN("x");       /* aligns the drain of M9 with the midpoint of c1 */
%dp1::pair @PRIORITY("disabled")    /* turns off only the group expanded for dp1 */
```

- Every named local object of a body can be referred to from outside through the usage, written `usage::local_name`; unnamed ones cannot be reached, except by number from the record of the solver (5.9). Whether to name one is the choice of the author of the definition, as with "to refer to an object, name it" (5.7); there is no separate export list.
- Paths can be chained: `%st1::dp::mid` is the mid of the local usage dp in the body of st1. Going through a role works the same way: `%st1::pair::mid` is the mid of the usage that fills the role pair.
- These are ordinary references: facts can be added to them (PLACE, SET-STYLE, TEXT, PRIORITY and so on), they can be used in other statements, and they can fill roles of other definitions; conflicts with the facts of the expansion are resolved by 8.5. A local name cannot be bound again from outside: `%c1::mid = …` is an error.
- Names: the names reached through `::` on a usage must be unique. So a local name cannot be the same as a role name or a standard sub-object name, which is checked at the definition, and the key of a text on the usage cannot be the same as any of these names (6.5, 13.3). A local name can be the same as a parameter name: parameters are values, and `::` does not reach values.
- Priority: a PRIORITY statement written from outside directly on a local object overrides the priority of the usage (14.8).
- Out of reach: the local objects of unnamed usages, except by number from the record of the solver (5.9), and those of usages generated one by one by a selection. A selection whose scope is a usage still selects only the members of its roles, not its local objects (10.1): local objects can be reached only by name, and the rules of a package cannot see inside definitions.
- Local names are therefore part of the interface of a definition: when a definition changes a local name, the references from outside no longer find it and fail (chapter 7), the solver's result becomes stale with them, and a hint suggests solving again (chapter 17). The standard package states which local names of its definitions can be relied on.

- **Notation in the body**: roles and parameters are written `?name`, local names `%name` (6.9, 6.10).
- **Derivation and extension**: only by composition; there is no inheritance relation (9.1). The body of a derived definition applies the base definition and adds its own statements:

```
$FILTER = @DEFINE(roles=["in", "out"]) {};
$LOWPASS = @DEFINE(roles=["in", "out", "r", "c"]) {
  %f = ?in(in) ?out(out) @AS $FILTER;
  {?r ?c} @ORDER("x")
}
```

- Content is added with statements in the body, and numeric variants with parameters; to hide and extend a definition in a lower level, the body refers to the outer definition of the same name (6.9).
- Selecting by the base definition needs no special rule: the FILTER usage expanded from the derived definition is in the scene, so `# @SELECT-USAGE $FILTER` selects it, and rules written for `$FILTER` work on low-pass filters too, acting only on the FILTER usage inside them and not on the statements that the derived definition adds.
- There is no separate mechanism for dropping part of the base definition: the body of a derived definition can write `%f::x @PRIORITY("disabled")` to turn off part of it, but a priority on the outer usage overrides that (14.8); to leave out a part reliably, split the base definition into smaller ones, or write another one.

9.5 Uses of general definitions

What a definition does is decided by its content, not by a keyword:

```
$PAIR = @DEFINE(roles=["in_p", "in_n"]) {};                  /* meaning: roles only */
$SIDE_BY_SIDE = @DEFINE(roles=["a", "b"]) {                 /* layout: constraints in the body */
  {?a ?b} @ALIGN("y");
  {?a ?b} @ORDER("left-to-right")
};
$PROBE = @DEFINE(roles=["p"]) {};                            /* virtual part: a rule gives its
usages a symbol (chapter 12) */
$PROBE_SYM = @DEFINE-SYMBOL {(0, 0) @PIN(terminal=1); (0, 0) @CIRCLE(radius=0.5)};
# @SELECT-USAGE $PROBE => @SET-SYMBOL $PROBE_SYM
```

In the standard package, `$CURRENT_MIRROR`, `$MIRRORED` and `$GROUND` are of these three kinds; `$DIFFERENTIAL_PAIR` has its roles and, in its body, a default layout (below).

- They are all the same kind of thing: a usage (recording which definition was applied and what fills each role), plus the descriptions expanded from the body. The two parts have different users:
  - the constraints expanded from the body are handled by the solver, must be satisfied, and are resolved by 8.5 in a conflict;
  - the usage is for people and LLMs to read, and for the selections and rules of RNL itself (10.1).

  So the usages of layout definitions carry meaning too; a definition with an empty body, such as `$PAIR`, records meaning through its roles alone, and its meaning is stated by the standard package or by its author. A definition that records meaning may also carry a drawing in its body (below).
- A semantic usage is not a comment: its references must be found, its roles must match, its basic validity is checked (Principles 2.4), and selections and rules can use it; only how it is drawn is not specified.
- "What it is" and "how it is drawn" may be written in one definition or in two. A definition can carry a default drawing in its body, at priority `"low"`: the author's own facts come before it, and the priority of the usage turns it off or raises it, such as `%dp1 @PRIORITY("disabled")`, or `# @SELECT-USAGE $DIFFERENTIAL_PAIR => @PRIORITY("disabled")` for every usage of the definition, while the usage still records what the objects are (14.8). The step from meaning to drawing can also be written as a rule in a package, and changing the package changes the drawing: `# @SELECT-USAGE $DIFFERENTIAL_PAIR => @AS $DP_LAYOUT`; deleting the layout usage does not affect the semantic usage. The standard package writes its default drawings in the bodies of its definitions, every constraint of them at priority `"low"`, such as the tail of a differential pair under the middle of the pair, the members of `$BRANCH` in one column and `$CHAIN` from left to right (16.3).
- **Drawing is always written in RNL**: a meaning is drawn in a certain way only by what RNL writes, in the body of its definition or in a layout definition applied by a rule, as above, possibly in a package; do not rely on default conventions of a renderer for a meaning. Whether a definition carries a drawing is up to its package; this standard makes no further rule about it. How the parts that RNL does not describe are laid out is up to the implementation (Principles 1), but that is not a drawing convention for a meaning and cannot be relied on. That a renderer decides the default symbol and style of objects without descriptions (Principles 2.7) is a different matter.
- **Vocabulary belongs to the standard package**: the first semantic definitions and the standard virtual parts are content of the standard package, which is not part of the language standard (16.3). How the definitions of the standard package state their meaning and roles is also up to the standard package. The meaning of a user's definition is its body; its name is only for people to read.
- **Basic validity** (Principles 2.4): the syntax of definitions does not check whether two roles are filled with the same object. The basic validity of one application is: the references are found, the roles receive the right kinds, the single roles are filled (9.4), and each statement of the expansion is valid by itself.

Error example

```
$DP_LAYOUT = @DEFINE(roles=[["in_p", "instance"], ["in_n", "instance"]]) {
  {?in_p ?in_n} @ALIGN("y");
  ?in_p ?in_n @DISTANCE("x", min=4)
};
#M1(in_p) #M1(in_n) @AS $DP_LAYOUT      /* the expanded DISTANCE asks M1 to be at least 4 from
itself */
```

  - In this example the expanded DISTANCE contradicts itself: it fails with a diagnostic, and only it is dropped (5.5, 7.3); the ALIGN holds as usual. A definition with an empty body, such as `$PAIR` above, cannot detect such filling: `#M1(in_p) #M1(in_n) @AS $PAIR` reports no error.
  - A virtual part with both ends on the same net has a definite meaning, so "the same object fills two roles" cannot be an error in general.
  - Further checks, such as requiring two roles to be different, are not provided in this version.
- **Keywords and general definitions** (1.5 item 7): what an implementation must understand itself, and what cannot be written with other descriptions, is a keyword, such as the atomic constraints, GROUP, WIRE, TEXT and CURRENT; what can be written into the body of a general definition is a general definition, such as differential

pairs, current mirrors, grounds and supplies, and signal flow (14.6). DEFINE-SYMBOL, DEFINE-STYLE, SET-SYMBOL and SET-STYLE are keywords too: a symbol body is in local coordinates and is evaluated where it is defined; the properties of a style are literals; SET-SYMBOL and SET-STYLE set one property of existing objects and are decided in queues once all objects exist, whereas AS and ATTACH produce a new object each time.

- A definition is identified by the definition that its name resolves to, not by the name string: when a lower level defines its own `$DIFFERENTIAL_PAIR`, that is another definition, and the rules of a package written for the standard `$DIFFERENTIAL_PAIR` do not recognize it.

10 Selections, rules and generation one by one

This chapter covers the selection keywords, the rules that use them, and generation one by one.

10.1 Selection keywords

There is one selection keyword for each kind of object selected, each with a fixed set of conditions. The result of a selection is a set (5.2).

Each selection keyword takes one scope object. `#` is the level in which the selection is evaluated (10.3). Given an object or a set, the selection looks among those objects: given a set of nets, for example, SELECT-TERMINAL selects the terminals on those nets (how the scope leads to the kind selected is described after the table). The scope cannot be left out, so a simple selection is also written with `#`: `# @SELECT-INSTANCE(name="R*") => @SET-SYMBOL $RESISTOR`.

Keyword	Selects	Conditions
SELECT-INSTANCE	Host instances. With terminals in the scope, the instances they belong to; with nets, the instances that have a pin on them	`name` (the instance name, such as `name="R*"`); `model` (the name in the model position of the instance line: a `.model` name or a subcircuit name); `type` (the type in `.model`, such as `"nmos"`)
SELECT-TERMINAL	Terminals of host instances; only those that have a pin in the symbol of their instance	`name`, `model` and `type` of the instance; `terminal` (a terminal name or number); `net` (the name of the net it is on); `wired` (10.2); `current` (the current direction, 15.2: `"in"`, `"out"` or `"unspecified"`)
SELECT-NET	Nets. With instances, terminals or wires in the scope, the nets they are on	`name`; `port` (whether it is a port of this level); `wired` (10.2); `global` (whether it is a global net, 6.3)
SELECT-WIRE	Wires. With nets in the scope, the wires drawn on them	`net` (the name of the net of the wire; a bus matches when one of its nets matches); `bus` (`true`: only buses, 13.1; `false`: only wires that are not buses, whose net slot receives a net rather than a set)
SELECT-LABEL	Labels, that is, TEXTs attached to objects (13.3). With objects in the scope, the labels attached to them	`key` (the key of the text; takes `*` and is case-sensitive; a label without a key matches no `key=`)
SELECT-ANNOTATION	Annotations, that is, graphics in the scene and TEXTs attached to points (13.4)	No condition other than `role` and `recursive`
SELECT-USAGE	`scope @SELECT-USAGE $D`: the usages of the definition `$D` in the scope, looking inward. With `#`, the usages of the level; with usages, these usages themselves and the usages inside them. Without `$D`, the usages of any definition	The definition is the last object (5.6) and may be left out. No condition other than `recursive`
SELECT-ANCESTOR-USAGE	`objects @SELECT-ANCESTOR-USAGE $D`: the usages of `$D` that have these objects as members, looking outward; indirect members count	The definition is the last object (5.6) and is required. No condition other than `recursive`

Besides these, every selection keyword takes the condition `recursive`, and the first six take `role`, which has a meaning only when the scope holds usages (both below).

- **One keyword for each kind.** Each selection keyword is named after the kind it selects, such as `&clk @SELECT-WIRE => @SET-STYLE $signal` for the wires of a net, or `#M1 @SELECT-LABEL` for the labels of M1. No selection takes objects of all kinds: to cover every kind, write one rule for each kind, as the standard package does to give every object its base style.
- **How the scope leads to the kind selected.** Each object in the scope takes one step towards the kind being selected; an object that has no such step selects nothing, which is not an error:

- SELECT-INSTANCE: an instance is itself; a terminal leads to its instance; a net leads to the instances that have a pin on it.
- SELECT-TERMINAL: an instance leads to its terminals; a net leads to the terminals on it; a terminal is itself.
- SELECT-NET: a net is itself; an instance leads to the nets of its terminals; a terminal leads to its net; a wire leads to the nets it is drawn on.
- SELECT-WIRE: a net leads to the wires drawn on it; a wire is itself. SELECT-LABEL: an object leads to the labels attached to it; a label is itself. SELECT-ANNOTATION takes only the annotations in the scope.
- A step through terminals counts only the terminals that have a pin in their symbol, as SELECT-TERMINAL, the fill and ROUTE do (9.2.4). So `&vss @SELECT-INSTANCE` is `&vss @SELECT-TERMINAL => @SELECT-INSTANCE`: an NMOS drawn with a three-terminal symbol is not selected when only its bulk is on vss, and for such a transistor M1, `#M1 @SELECT-NET` does not include vss either. A port of this level, such as `#:inp`, still leads to its net. Before the symbol queue has given the symbols, all terminals count (8.2).
- With usages as the scope, only the members of the kind being selected are taken; members are not used as a scope again (see below).
- For example, the transistors controlled by the clock are `&clk @SELECT-INSTANCE`; only those whose gate is on clk are `&clk @SELECT-TERMINAL(terminal="g") => @SELECT-INSTANCE`; the nets of M1 are `#M1 @SELECT-NET`.
- Note: counting only terminals with a pin keeps what is selected the same as what is drawn, so the grounds, wires and styles that later rules add fit the drawing.

- **Only this level.** Every selection keyword has the condition `recursive`, `true` by default. With `recursive=false`, it selects only objects of the level where the statement is written; evaluated in a lower level, it is empty, so the rule takes effect only in its own level (10.4): `# @SELECT-TERMINAL(net="vss", wired=false, recursive=false) => $GROUND_STUB @ATTACH` adds grounds only in this level. A single host object is simply referred to, such as `&vss`: host references resolve only in their own level anyway.
- **SELECT-USAGE and SELECT-ANCESTOR-USAGE.** Both recognize a definition by what its name resolves to (9.5), always select usages, and have no `role=`.
  - SELECT-USAGE looks inward, like the other selection keywords. With `#` as the scope, it selects the usages of the level being evaluated; with a usage or a set of usages, these usages themselves and the usages inside them, down to the innermost. Instances, nets and terminals in the scope are not usages and select nothing; a warning points to SELECT-ANCESTOR-USAGE for the usages they are in.
  - SELECT-ANCESTOR-USAGE looks outward: it selects the usages that have an object of the scope as a member, directly or indirectly (see the example below). The usages in the scope are not included themselves. With `#` as the scope, it is an error: a level is not a member of anything.
- **Usages as the scope.** With usages as the scope, the other selection keywords select the members of their kind in these usages, and `role=` takes only the members of that role. Members that are usages are followed inward, down to the innermost:

```
$OTA = @DEFINE(roles=[["pair", "usage"], ["load", "usage"]]) {};
%dp1 = #M1(in_p) #M2(in_n) #M5(tail) @AS $DIFFERENTIAL_PAIR;
%cm1 = #M3(ref) #M4(out) @AS $CURRENT_MIRROR;
%ota1 = %dp1(pair) %cm1(load) @AS $OTA;
%all_set = %ota1 @SELECT-INSTANCE;                /* {#M1 #M2 #M5 #M3 #M4} */
%pair_set = %ota1 @SELECT-INSTANCE(role="pair");  /* {#M1 #M2 #M5} */
%up_set = #M1 @SELECT-ANCESTOR-USAGE $OTA         /* {%ota1} */
```

  - The value of `role=` also accepts the `definition=` aliases of the definition of the usage (16.5). For the direct members only, use the role as a sub-object, such as `%dp1::tail`. The walk stops at a usage already visited. Members are the objects that fill roles, not the local objects of the body (9.4).
  - **Only members of the kind selected.** Members are not used as a scope to go further, and members of other kinds do not count. SELECT-TERMINAL takes only terminal members: after `$GROUND @ATTACH &0; $GROUND @ATTACH #M5:3`, `# @SELECT-USAGE $GROUND => @SELECT-TERMINAL` is only , not the other terminals on net 0. The members of the differential pair %dp1 are instances, so `%dp1 @SELECT-TERMINAL` is empty. For the terminals of member instances, write `%dp1 @SELECT-INSTANCE => @SELECT-TERMINAL(terminal=2)`

(the gates of all three transistors, the tail included); for the terminals on member nets, `… => @SELECT-NET => @SELECT-TERMINAL`. Each segment does one thing; to go further, add a segment. Objects given directly in the scope follow the rules above: a net gives the terminals on it, an instance its terminals.

- To find objects in the scene by condition, always use a selection keyword; `::` takes only parts of the object itself (6.5).
- **Bundles.** The members of a bundle cannot be selected; the bundle is selected instead (11.8). SELECT-INSTANCE selects the bundle itself by the conditions of its first member, once for each bundle. SELECT-TERMINAL selects the terminals of the bundle, and `net=` matches when it matches the net of any one member.
- **Conditions** are written by name, as `name=value`; an object is selected only when all the conditions written hold. For "or", write several rules. The conditions that compare names (`name`, `model`, `net`) take the wildcard `*`, which matches any number of characters, and compare by the case rules of the host, as DEVICE does (6.12). To select by element letter, write `name="R*"`. Spectre instance names do not start with an element letter; select them by `model`, such as `model="resistor"` (6.12).
- **`model`** compares the name written in the model position of the instance line: for M, Q, D and other elements, the `.model` name; for X instances, the subcircuit name, including PDK devices that the netlist does not define, such as `# @SELECT-INSTANCE(model="sky130_fd_pr__nfet_01v8")`; in Spectre, the master. It is the same as the `model=` of DEVICE (6.12). `type` compares the type in `.model`; X instances have no type.
- Comparison follows 3.6: condition names and standard enumeration values are case-sensitive; values that name host things (names, model names, net names) follow the rules of the host.
- A net can be selected by name: `# @SELECT-NET(name="0")` selects it, and `# @SELECT-TERMINAL(net="0")` the terminals on it. Objects can also be selected by meaning: `# @SELECT-USAGE $DIFFERENTIAL_PAIR => @SELECT-INSTANCE(role="in_p")` selects the positive input transistor of every differential pair.
- There is no keyword for an intersection, such as the members of a set that belong to a usage; write it with REMOVE-FROM twice (5.2.2):

```
%s_set = {#M1 #M3 #M5};
%s_set @SELECT-ANCESTOR-USAGE $DIFFERENTIAL_PAIR => @SELECT-INSTANCE => %dp_m_set;
%out_set = {%s_set};
%dp_m_set @REMOVE-FROM %out_set;    /* the members of %s_set in no differential pair */
%in_set = {%s_set};
%out_set @REMOVE-FROM %in_set       /* the members of %s_set in a differential pair */
```

- **Ground and supply nets** are given by sets that the standard package provides, `$GROUND_NET_SET` (net 0 by default) and `$SUPPLY_NET_SET` (empty by default). There are no net classes.
  - A package cannot contain host references (16.2), so the default content of such a set can only be written as a selection, such as `# @SELECT-NET(name="0")`.
  - The user adds and removes nets with ADD-TO and REMOVE-FROM:

```
&vss @ADD-TO $GROUND_NET_SET;                                  /* the vss of this level
*/
# @SELECT-NET(name="vss") => @ADD-TO $GROUND_NET_SET;          /* the nets named vss of
every level */
# @SELECT-NET(name="vdd!", global=true) => @ADD-TO $SUPPLY_NET_SET  /* only the global vdd! */
```

    The second statement also adds local nets named vss in subcircuits. To leave a net out in one place only, copy the set and remove the net from the copy: `%g_set = {$GROUND_NET_SET}; &vss2 @REMOVE-FROM %g_set`.
  - A fill rule uses such a set as its scope, such as `$GROUND_NET_SET @SELECT-TERMINAL(wired=false) => $GROUND_STUB @ATTACH`.
  - Writing `$GROUND_NET_SET = …` in one's own level hides the set of the package, and the rules of the package do not see the new one (6.8).
  - Grounds and supplies are not guessed from net names. An importer writes the power domain of a net in an editor file (such as `powerDomain`) as ADD-TO.

  - A solver may read the standard sets as hints (such as supplies at the top and grounds at the bottom), as it may read the definitions of the standard package; this is not required.
  - Note: there is no keyword for ground and supply nets (1.5 item 7): the uses of these sets, such as selecting terminals to add grounds or to change symbols, can all be written as rules in a package.
- **Global nets.** The condition `global=true` of SELECT-NET selects only global nets (6.3). Local nets with the same name in different levels are different nodes, and selecting by name selects all of them; to take only the global one, add this condition.

10.2 The condition `wired`

- A terminal is `wired=false` when it is not one of the points of any explicit wire (an end or a point in between), and does not share a bus pin with such a terminal (9.2.4). Bus wires are explicit wires too (13.1). A wire to a terminal of a bundle connects that terminal of every member (11.8).
- A net is `wired=false` when no terminal of a host instance on it is touched by an explicit wire; the terminals of virtual parts do not count.
- Only wires written explicitly count, not generated ones. Whether there are virtual parts makes no difference.
- Wires count as they are written, from step 3 of 8.2, also a wire that fails later because a terminal has no pin (7.1, 9.2.4). For a wire that is rewritten, the last rewrite counts (13.1).

10.3 Evaluating selections

- A selection is not expanded where it is written, and each evaluation looks at one level only. A statement that contains a selection (a rule, 10.4) is evaluated once in the level where it is written and once in each level below; in each evaluation, `#` is that level, and the selection selects only objects of that level. So `# @SELECT-INSTANCE(model="nch") => @ALIGN("y")` written at the top level aligns once in each drawing, and does not align instances of different drawings with each other.
- A selection is evaluated in the phase that uses it: appearance rules in the symbol queue and the style queue, generation rules in the generation phase (8.2). The same holds for sets (5.2).
- The kind that a selection selects is known when it is read. Giving it to a slot that does not take that kind is an error at once, also inside a set, such as `# @SELECT-TERMINAL(net="vss") => @SET-SYMBOL $NMOS`.
- SELECT-USAGE and SELECT-ANCESTOR-USAGE always select usages; when a selection goes from usages to their members, the kind selected is given by its keyword. So the kind of every selection is known when it is read.
- The result of a selection is a set and can be named with `%` or `$`. A set evaluated in a level takes only the members of that level: `&vss @ADD-TO $GROUND_NET_SET` written at the top level adds only the vss of the top level, which is not in the set when it is evaluated in a subcircuit (6.6).
- When the scope of a selection is written as host objects of its level, such as `&Q<0:3:2, 1> @SELECT-TERMINAL(name="R*") => @TEXT("${net}", key="netname")` at the top level, these objects do not exist when the rule is evaluated in a lower level: the selection is empty there and does nothing, with no warning, just as a set takes only the members of the level. The warning of 10.5 is given only for the other roles of generation rules.

10.4 Rules

- A statement that takes the result of a selection is a rule.
- Any keyword can take the result of a selection, one by one or as a whole as 5.2.1 says: `# @SELECT-INSTANCE(type="nmos") => @ORIENT("MY")`, `# @SELECT-INSTANCE(model="nch") => @ALIGN("y")`.
- SET-SYMBOL and SET-STYLE with the result of a selection as their object are appearance rules and go into the appearance queues (8.3). AS and ATTACH are generation rules (10.5). With any other keyword, a rule is the same as a statement written for each selected object, or one written for all of them, and takes effect in the phase of its keyword.
- A rule applies to the level where it is written and to the levels below it, evaluated once in each level and looking only at that level (10.3); a rule in a package applies to every level that uses the package. With `recursive=false` in its selection, a rule takes effect only in the level where it is written. Rules are treated as definitions (6.1).

10.5 Generation one by one

"For each terminal on vss without a wire, place a ground and connect it with a wire":

```
$GROUND_STUB = @DEFINE(roles=[["t", "terminal"]], parameters=[["gap", 2]]) {
  %g = $GROUND @ATTACH ?t::net;
  ?t::net @WIRE {?t %g:p};
  {?t %g:p} @ALIGN("x");
  %g:p ?t @DISTANCE("bottom-to-top", exact=?gap)
};
# @SELECT-TERMINAL(net="vss", wired=false) => $GROUND_STUB @ATTACH => %stubs_set
```

The standard package provides a `$GROUND_STUB` like this one, except that it gives only the least distance (`min=?gap`), so that a solver may draw the wire longer, as when it lines up the grounds of parts of different lengths, that it points the ground down (`@ORIENT("R0")`), and that its constraints are of priority `"low"`; the definition here hides it (6.8).

- When a single role of AS or ATTACH receives the result of a selection, the definition is applied once for each selected object, and the other roles are the same in every application (5.2.1, 9.4). A set role takes the result of a selection as a whole, and the definition is applied once.
- Virtual parts are created one by one in the same way: `# @SELECT-NET(port=true) => $IO_PORT @ATTACH` gives each port net a port tag; with `$RLOAD`, a definition of two roles drawn as a resistor, `# @SELECT-NET(name="out*") => $RLOAD @ATTACH &vss` attaches one between each selected net and vss; since `&vss` belongs to this level, the rule takes effect only here, with the warning described below.
- A ground can also be drawn right on the pin, without a wire: `# @SELECT-TERMINAL(net="vss", wired=false) => $GROUND @ATTACH`. The role is then filled with a terminal, and the usage is drawn at its pin (12.1). Such a ground has no wire, so the terminal stays `wired=false`: when a ground on the same pin is also written explicitly, the rule still generates one, and the two overlap. This is why the fill of the standard package adds grounds and supplies with a wire (`$GROUND_STUB`, `$SUPPLY_STUB`): once they are written back, the wire makes the terminal no longer `wired=false`, and the fill skips it. For a ground that the user writes right on a pin, the fill takes the terminal out with a rule such as `# @SELECT-USAGE $GROUND => @SELECT-TERMINAL => @REMOVE-FROM $GROUND_FILL_SET` (10.1), so no second ground is added. A ground placed as usual and connected by a wire of length 0 (13.1) also works.
- The pairing is within each application: the local names of a definition are new in each application, so the `%g` of one application is the ground made in that application.
- The result is a generated set. It can be named and given as a whole to SET-SYMBOL or SET-STYLE (such as `%stubs_set @SET-STYLE $grey`), but no single member of it can be referred to (5.7).
- **Writing one by hand.** To describe one of them on its own, create it by hand and wire it. Its terminal is then no longer `wired=false`, and the fill skips it. There is no separate way to cancel a generation.
- **Snapshot.** All generation rules see the same snapshot, the result of steps 3 and 4 of 8.2 (after override by key, with the symbols of all objects), and do not see each other's results. So the same rule written twice generates two copies.
- Generated facts take the position of their generation rule in conflict resolution (8.5).
- Generated objects without a position are placed by the solver; to say how they are placed, use a definition with constraints.
- **Write-back.** A single generated object cannot be referred to (5.7). When the solver writes them back, it writes them as explicit statements, such as `$IO_PORT @ATTACH &out => @PLACE(40, 20) => %p_out` with the wire to it. The explicit object takes the place of the generated one only when the rule leaves out what is already written, as the fill of the standard package does by writing the difference of two sets (see the fill by net below); a rule written otherwise makes the object again beside the explicit one, without a position. A rule whose objects the solver places should therefore be written as such a difference. An object that needs only an orientation and no wire (such as a ground drawn on a pin) may also be written back through a selection, such as `#M1:3 @SELECT-ANCESTOR-USAGE $GROUND => @ORIENT("R180")`. The objects in the body of an explicit usage written this way that have no local name are referred to by number (5.9). The usage is written with the priority that the

rule writes on its application, as written, or with `"disabled"` when it has that from the head of the record of the rule (14.8); otherwise it is written without a priority, so that its body keeps the priorities it had, and a record head with another priority is not written.

- Generation does not recurse: when the content expanded by a generation rule contains rules, those rules fail. There is only one generation phase, and rules made by such an expansion have no phase in which to take effect.
- When another role of a generation rule is filled with an object of the level (a scene object or a host object), the rule takes effect only in the level where it is written, with a warning: these objects belong to that level, and the levels below do not have them. A role given a definition has no such limit.
- A SET-SYMBOL or SET-STYLE made by a generation belongs to the group of the rule that made it, at the position of the rule (8.3).
- A name bound to a selection or to a generated set cannot be bound again (6.6).
- **Fill by net.** There is no built-in rule "add none where one exists". The fill of the standard package writes the difference of two sets; the standard package writes:

```
$PORT_FILL_SET = # @SELECT-NET(port=true);                              /* the nets that get a
port tag */
$SUPPLY_NET_SET @REMOVE-FROM $PORT_FILL_SET;                            /* none on a supply net
*/
$GROUND_NET_SET @REMOVE-FROM $PORT_FILL_SET;                            /* nor on a ground net:
the stubs stand there */
# @SELECT-USAGE $IO_PORT => @SELECT-NET => @REMOVE-FROM $PORT_FILL_SET;   /* less those that
have one */
$IO_PORT @ATTACH $PORT_FILL_SET
```

  - Written in a file rather than in the package, these statements would hide the set of the standard package (6.8) and add a second port fill.
  - A port tag written by hand takes the place of the generated one: after `$IO_PORT @ATTACH &inp => @PLACE(0, 20) => %p`, inp gets no generated port tag. The user can also take nets out of `$PORT_FILL_SET`, such as `# @SELECT-NET(name="vdd") => @REMOVE-FROM $PORT_FILL_SET`, or count the usages of their own port definition: `# @SELECT-USAGE $MY_PORT => @SELECT-NET => @REMOVE-FROM $PORT_FILL_SET`.
  - Generated port tags do not count: the generation rules use `$PORT_FILL_SET` in step 5, and the selections in the difference are evaluated on the state before step 5 (5.2).
  - Note: "one already exists" is not built in, because it is unclear for a definition with several roles, the user's own port definitions would not count, and a repeated layout definition only repeats constraints, which conflict resolution and priorities handle (8.5, 14.8).
- In a fill by terminal, it is still the wire of an object written by hand that takes the place of the generated one: a terminal with a wire is no longer `wired=false`. There is still no way to cancel a generation: taking a net out of the set only changes the set.

11 Placement and appearance

11.1 Position: PLACE

```
#M1 @PLACE(10, 20);
#M1 @PLACE(x=10)
```

- PLACE gives an object a position; at least one of x and y must be written. PLACE sets only the coordinates written. Each coordinate is free at first, and a coordinate never written stays free, to be decided by constraints and the solver.
- PLACE sets only coordinates, not rotation, mirroring, scale or the directions of terminals; the orientation is set with ORIENT (11.2).
- A PLACE is two geometric facts, one for x and one for y, and in a conflict each can be discarded on its own (8.5). The parameter `priority=` gives its priority (14.8), such as `#M1 @PLACE(12, 20, priority="high")`: x and y by position come first, the parameters by name after them (4.2 item 9).
- Objects: instances, virtual parts with a position of their own (12.1), bundles (11.8), points, TEXTs (annotations and labels; PLACE sets the position of the text itself, 13.3), the origin `::position` (including the centre of a RECTANGLE, CIRCLE or ELLIPSE written with a centre), and the `::position` of a set. PLACE does not take other graphics; to move them, move their points (13.4). PLACE on a usage without a symbol, or on a usage drawn at a point of its main object, fails (12.1). In a symbol body, PLACE only positions the TEXTs attached to `?`, in local coordinates (9.2.2).
- The output keeps the handle of the input (5.1).
- A set as the object is placed member by member (5.2.1): `%pair_set @PLACE(10, 20)` puts both members at (10, 20). To place the centre of the set, write `%pair_set::position @PLACE(10, 20)` (6.5).
- For the axes, unit, decimals and tolerance of coordinates, see 11.7.

11.2 Orientation: ORIENT

```
%pair_set = {#M3 #M4};
#M2 @ORIENT("MY");          /* flipped left-right */
%pair_set @ORIENT("R90")    /* each member turns about its own origin and stays where it is */
```

- Objects: instances, usages with a symbol, and bundles. ORIENT on a usage without a symbol fails (12.1); points, labels, graphics, wires and nets are not accepted.
- A set as the object is oriented member by member (5.2.1): each member turns about its own origin and stays where it is, and each member is one description. The set is not turned as one block. Turning a whole set together is not provided: `%pair_set::position` is a point, which ORIENT does not take.
- A usage drawn at a point of its main object turns about that point; it has an orientation of its own and does not turn with the main object (12.1).
- Without ORIENT there is no constraint, and the solver chooses; a renderer that does not solve draws `"R0"` (8.5). Relations between the orientations of several objects (the same orientation, mirror images) are written with ORIENT-RELATION (14.9).
- ORIENT is one geometric fact and takes part in conflict resolution (8.5); the parameter `priority=` gives its priority (14.8).
- **Orientation codes.** Each code stands for one orientation. Only the eight spellings of the table are accepted, and they are case-sensitive (3.6). The names are those of Virtuoso and, with y up (11.7), so are their meanings. Equivalent spellings (such as `"MXR180"`, which is MY) and arbitrary angles are not accepted: they are wrong parameter values, and the ORIENT fails. Tools write back only these eight. There is no relative form that turns an object further from its current orientation (chapter 17).

Code	(x, y) becomes	Meaning
`"R0"`	(x, y)	Not turned

Code	(x, y) becomes	Meaning
`"R90"`	(−y, x)	Turned 90° anticlockwise
`"R180"`	(−x, −y)	Turned 180°
`"R270"`	(y, −x)	Turned 270° anticlockwise, that is, 90° clockwise
`"MY"`	(−x, y)	Mirrored about the y axis: flipped left-right
`"MX"`	(x, −y)	Mirrored about the x axis: flipped upside down
`"MYR90"`	(−y, −x)	MY, then R90
`"MXR90"`	(y, x)	MX, then R90

- **Order of composition.** MXR90 and MYR90 mirror first and then turn; the names read in the order of the operations, as in Cadence and in the OpenDB of OpenROAD.
- **How it acts.** The orientation acts on the local coordinates of the symbol, about its origin, and the origin is then put at the position of the object: a point of the symbol is drawn at the position of the object plus the orientation applied to its local coordinates. The graphics and pin positions of the symbol are transformed rigidly with it, and the directions in which wires leave the pins turn too (9.2.4). Texts are shown as 9.2.5 says; glyphs are not mirrored.
- **Only where things are drawn.** An orientation changes only where pins are drawn, not which terminals they belong to. When a block symbol is flipped upside down, the pin of inp is drawn at the bottom, and the wire to inp follows it. The two pins are not swapped to keep inp at the top: that would change which wire connects to which terminal, a matter of the netlist (Principles 2.2). To draw inp elsewhere, use another symbol (11.3).

11.3 Symbols and styles: SET-SYMBOL and SET-STYLE

```
%w13 = &x @WIRE {#M1:1 #M3:1};
#M1 @SET-SYMBOL $NMOS;
# @SELECT-INSTANCE(model="nch") => @SET-SYMBOL $NMOS;
%w13 @SET-STYLE $signal;
# @SELECT-INSTANCE => @SET-STYLE $base
```

- SET-SYMBOL gives an object a symbol. The object can be an instance, a usage or a bundle, or a set of them (including the results of selections and of generation one by one); each member is given the symbol once (5.2.1). In a subcircuit, `# @SET-SYMBOL` gives the block symbol (9.2.6).
- A usage that is given a symbol is drawn: it is a virtual part (chapter 12).
- SET-STYLE gives an object a style. The object can be an instance, a usage, a bundle, a wire or a graphic (texts included), or a set of them (including the results of selections and of generation one by one). It does not take nets, levels, constraints, definitions or points. Terminals are drawn with the symbol of their object and take the style of their object. A style given to a usage passes to its symbol and to the drawable objects created in its body (8.3), so a style given to a semantic usage without a symbol is not an error; when nothing is drawn in the body, it has no visible effect.
  - Note: otherwise the rule `# @SELECT-USAGE => @SET-STYLE $base` of the standard package would give a warning for every semantic usage.
- The output keeps the handle of the input; when the object is a set, the output is that set.
- Both take effect through the appearance queues, not by override by key. There are two tiers: a SET-SYMBOL or SET-STYLE that names the object directly is above those that reach the object through a selection or are passed on to it (8.3).
- Where default rules exist (the standard package, 16.3), they need not be written.
- **Parameters and styles.** Style parameters written on an object, such as `size=` of TEXT, are the direct style of that object, above the styles from selections or passed on. A SET-STYLE that names the object directly and is written later overrides them by position (8.3); this is how an editor changes a single object. A parameter of a definition that is not passed counts as not written, so the style takes effect (9.4).
- **Labels and junctions.** The style of an object is passed on to the texts attached to it. In the tier of styles selected or passed on, it is compared by position with the rules that select the text (such as SELECT-LABEL); the parameters of the text itself and a SET-STYLE that names the text directly are in the direct tier. So when `#M1 @SET-STYLE $signal`

is written after the label rules, the name of M1 takes that colour too, and dragging the name does not change its colour. Texts attached to `?` in a symbol are labels as well, and SELECT-LABEL selects them; fixed texts attached to points are graphics of the symbol and take the style of the object. Texts attached to nets or points receive no style from what they are attached to and use only their own. A junction is a usage of `$JUNCTION` (13.2); the dot is its symbol, and its colour follows the style of the usage.

11.4 Several drawings of one object

There is no way to draw one host object several times: a host handle such as `#M1` always means the one drawing of that object. Should a later version add several drawings of one object, they would be separate objects with names of their own, and host handles would keep their meaning (Annex C).

Note: several instances drawn as one are a bundle (11.8).

11.5 Groups: GROUP

```
%stage1_set = {#M1 #M2 #M5} @GROUP;
%stage2_set = {#M3 #M4} @GROUP;
{%stage1_set %stage2_set} @GROUP;
%stage1_set %stage2_set @DISTANCE("x", min=1);
{%stage1_set::position %stage2_set::position} @ALIGN("y")
```

- GROUP adds one fact to a set: the solver treats it as one block. It keeps the handle of the set, so `%stage1_set` is still that set.
- The set itself is what the solver registers. GROUP is not a fifth atomic constraint, nor another kind of object.
- To treat several objects as a whole, PLACE takes the `::position` of a set (6.5); a set written as `{%x}` is one member of ORDER, ALIGN or DISTRIBUTE, counted by its centre, and a set given to DISTANCE counts by its extent (14.1, 14.5). Any set works this way; GROUP is not needed for it.
- Nesting follows from containment: the members of `{%stage1_set %stage2_set}` are the five instances (a reference is spread one level, 5.2), so both stages are contained in it.
- **Meaning.** GROUP is only a hint to the solver, "treat it as one block when solving", and guarantees no geometric relation. Requirements on positions, such as not interleaving (the DISTANCE in the example) or being centred, are written as atomic constraints; how GROUP affects search and scoring is up to the implementation. There is no default "no interleaving": that would add constraints behind the user's back. Two groups may overlap, with a member in two groups neither of which contains the other; the solver handles that.
- A grouped set is still a set, and a reference to it in `{…}` is spread one level. To use a whole group as one member of a constraint, write it in its own `{…}` or write its `::position`, such as `{#R1 {%rc_set} #M1} @ORDER("left-to-right")` (14.1, 14.3, 14.6).
- GROUP is a keyword, not a definition of the standard package: its meaning is known only to the solver and cannot be written in the body of a definition (1.5 item 7).
- A viewer may draw a group, such as with a frame; this is up to the viewer, and this standard does not specify it.
- GROUP is not an atomic constraint: it takes no part in conflict resolution (8.5) and has no priority, only on and off. `%stage1_set @PRIORITY("disabled")` withholds the group from the solver, which cancels the grouping. When an outer usage is disabled, the groups in its expansion are off too, unless a PRIORITY written from outside through `::` directly on a group turns it on again (14.8). Requirements on positions (such as two groups not interleaving) are written as atomic constraints, each with its priority. A group in the expansion of a definition can be turned off on its own only when it is a named local object, referred to from outside (9.4).

11.6 Drawing order and layers: LAYER

```
@RECTANGLE(fill="fillcolor") {(0, 0) (40, 30)} => @LAYER("bottom") => %f1;    /* a frame, drawn
below everything */
(2, 28) @TEXT("input stage") => @LAYER("top") => %n1
```

- There are three layers: `"bottom"`, `"middle"` and `"top"`. The bottom layer is drawn first, then the middle, then the top. Within a layer, objects are drawn by position: a later object is drawn over an earlier one and covers it.

- `@LAYER(…)` sets the layer of an object; without it, the object is on `"middle"`. There is one key for each object, and the later LAYER overrides (5.8). LAYER does not change the position of the object within the layer.
- **Position** is the order of writing (1.5 item 2), the same order as in 8.3 and 8.5: within a level, the order of the text; the content of a package at the position of its USE-PACKAGE, with the packages at the root first (16.3); statements written in with `scope=` after the target subcircuit's own text. Each kind of object takes the position of:
  - wires, graphics and texts (annotations and labels): the statement that creates it;
  - usages: the SET-SYMBOL that takes effect, or, for a symbol given by a rule, the rule (8.3);
  - objects made by rules, by the expansion of definitions and by generation one by one: the statement that makes them, and within one expansion the order in the body;
  - instances: the very bottom of their layer, in netlist order among themselves; an instance put on another layer is at the bottom of that layer. A bundle counts as an instance, with the netlist order of its first member.
- Statements that change an object, such as PLACE, ORIENT, SET-STYLE and LAYER, do not move its position.
- Objects: instances, usages with a symbol, bundles, wires, graphics and labels; a set member by member (5.2.1). LAYER on a usage without a symbol is not an error but has no effect (as with SET-STYLE, 11.3); on points, nets, constraints and definitions, it fails.
- The members of a symbol have no layers: they are drawn in member order (9.2.1), together with their object.
- There is no numeric `z=`. Overriding (by the key of a fact) and covering (by layer and position) are separate.
- "Layer" here means the order of drawing; it has nothing to do with the levels of the host (2.6).

11.7 Coordinates and units

- **Axes.** x goes right and y goes up; a larger y is higher. So `"bottom-to-top"` of ORDER and DISTANCE goes upward (14.3, 14.5), and R90 is anticlockwise (11.2). SVG, other formats and some editors have y going down; the implementation flips y when drawing, importing and exporting.
- **Unit.** A unit has no physical size. How large a unit is drawn or printed is up to the viewer.
- **Decimals.** Coordinates and lengths may be written with decimals (3.7); they need not be integers, and there is no limit on the number of digits. Coordinates are drawn as written and are not snapped to integers.
- **Grid.** Pins should be on integer points (9.2.4); graphics in symbols are not limited. When an instance is placed on an integer point with one of the eight orientations, integer pin positions are still integers after the transform, so wires end on integer points too: the integer points are the usual grid. The language has no grid of its own. Whether the solver rounds free coordinates to integers is its own rule, ranked after all language descriptions (8.5); when constraints force a decimal, the decimal is used. The bend points that a routing mode computes are kept as computed, not rounded (13.1).
- **Tolerance.** When checking whether a constraint holds and when deciding conflicts (8.5), a difference within the tolerance counts as satisfied. The tolerance is an option of the solver that the user can set; this standard does not fix its value, but it is greater than 0, so that a value without a finite decimal form can be written back (below). A viewer that does not solve uses its own setting when it checks. So two tools may disagree on a case right at the edge of the tolerance. A solver writes back coordinates as decimals that read back as the same value, without further truncation; a value that has no finite decimal form, such as 1/3, is written with enough digits that the facts it satisfies still hold within the tolerance when the record is read back (chapter 17).
- **Symbol coordinates.** The local coordinates in a DEFINE-SYMBOL body have the unit and axes of the scene; their origin is the origin of the symbol (9.2.1, 6.5). There is no scaling: symbols take no scale parameter, and instances cannot be scaled. An importer converts the symbols of other tools to RNL units.
- **Other lengths.** All lengths of the language are in coordinate units and scale with the drawing: `min`, `max` and `exact` of DISTANCE, `radius=` of CIRCLE, `width=` and `height=` of RECTANGLE and ELLIPSE, the text size `size` (as the font-size of SVG), the line width `linewidth`, the arrow length `arrowsize` (9.3), and parameters of definitions used as lengths (such as `gap=3`). A line width of 0 is the exception: a hairline, equally thin at every zoom, which is also the default (9.3).
- **Origin and paper.** The origin of the scene has no special meaning. Each subcircuit is one drawing (6.1) with its own coordinates; the viewer frames the view by the extent of the content. RNL records neither a paper size nor the snap grid of an editor.

11.8 Bundles: BUNDLE

An iterated instance of Virtuoso, such as `I0<3:0>`, is one symbol but four lines in the netlist:

```
* A unit cell repeated four times
* @RNL(type="header") { @VERSION("0.3")}
.subckt unit in out vdd vss
R1 in out 1k
.ends unit
XI0<3> D<3> out vdd vss unit
XI0<2> D<2> out vdd vss unit
XI0<1> D<1> out vdd vss unit
XI0<0> D<0> out vdd vss unit
* @RNL {
*   #XI0<3:0> @BUNDLE => @PLACE(30, 10) => %x;    /* four instances drawn as one symbol */
*   &D<3:0> @WIRE {(20, 10) %x:in};               /* a bus to the in pin of the bundle */
*   &out @WIRE {%x:out (40, 10)};                 /* all four members are on out: an ordinary pin
*/
*   %x @TEXT("I0<3:0>", key="name")               /* the name, written out */
* }
.end
```

- **A new object.** BUNDLE takes a set of host instances and makes a bundle, which is drawn once in place of these instances. The members are in the order written, and the first member decides the pins and the symbol.
- **Pins and symbol.** The terminal names and the number of terminals are those of the first member. The appearance queues take the bundle as its first member: selections reach the bundle by the conditions of its first member (10.1), and the block symbol of the subcircuit applies. SET-SYMBOL, styles, PLACE, ORIENT, LAYER, constraints and texts written directly on the bundle belong to the bundle.
- **Terminals.** A terminal of the bundle, such as `%x:in`, stands for the terminals of the same name of all members: it is a bus pin.
  - `::net`: when all members are on the same net, that net, such as `&out` for `%x:out::net`; otherwise the set of their nets in member order, such as D<3>, D<2>, D<1>, D<0> for `%x:in::net`. So `?t::net @WIRE {…}` in a definition draws a bus when it gets a set, and an ordinary wire when it gets a single net (13.1).
  - A wire to a terminal of the bundle connects that terminal of every member (10.2).
- **Attributes** (13.6): when a value is the same for all members, that value; otherwise the values joined with `,`. `${w}` shows 1u when all are 1u; `${name}` is `XI0<3>,XI0<2>,XI0<1>,XI0<0>`, and to show `I0<3:0>`, write a text with the same key, as in the example. `${net}` of a terminal works the same way, such as `D<3>,D<2>,D<1>,D<0>`.
- **Selections.** Members cannot be selected; the bundle is selected instead (10.1). So a rule such as "a net name for each unwired pin" gives a bundle one label, which shows the whole bus. A role of a definition that takes instances also takes bundles, and one that takes terminals also takes the terminals of bundles (9.4).
- **Members are not drawn on their own.** A description about drawing written directly on a member or on a terminal of a member (a position, orientation, symbol, style, layer, constraint, wire or text) fails, as for a usage without a symbol (12.1 item 3). Descriptions that do not concern drawing work as usual, such as CURRENT and the roles of semantic usages.
- **All of one sort.** The members should have the same element letter, model or subcircuit name, parameters (compared as written) and number of terminals. When they do not, a warning is given; they are still bundled, with the pins and symbol of the first member.
- **One bundle for each instance.** An instance written in several BUNDLEs belongs to the one written later (1.5 item 2); the earlier one keeps its other members, and a hint may be given. An earlier bundle whose members have all been taken is empty and is not drawn, with a warning.
- BUNDLE with an empty set fails (5.2.1). A member that is not a host instance fails (a wrong kind). A member that cannot be found makes the whole statement fail (strict, 7.3).

12 Virtual parts

12.1 Concept

- A virtual part is a part that exists only in RNL, such as a ground, a supply, a port tag or a no-connect mark. It does not exist electrically: it is only drawn, and it neither creates nor changes electrical facts.
- A virtual part is a usage that has a symbol; there is no separate keyword or kind of object for it. `$GROUND @ATTACH &vss` makes a usage; a rule of the standard package gives the usages of `$GROUND` a ground symbol, so it is drawn.
- Rules:
  1. The terminals of a usage are the roles of its definition: `%g0:p` is a terminal, the position of pin p on the symbol; `%g0::p` is the member that fills role p, the net vss. This is the same as for instances: `#M5:3` is a terminal, and `#M5:3::net` is its net.
  2. A usage is drawn, and can be placed, only when it has been given a symbol; a usage without a symbol is not drawn. The standard package gives symbols only to the definitions that are meant to be drawn, so semantic usages such as differential pairs are not drawn.
  3. A usage without a symbol has no meaning for the drawing: PLACE, ORIENT, constraints and wires on it or on its terminals fail, as for terminals without a pin (9.2.4); this is checked after step 6 of 8.2. A style given to it is not an error: the style passes to the drawable objects created in its body (8.3), and has no visible effect when nothing is drawn in the body (11.3).
  4. A drawn usage draws only its own symbol. It neither contains nor moves its members, and it is not a visual container.
  5. **Where it is drawn.** The object that fills the first role is the main object of the usage. When the main object has a coordinate point (for a terminal, the position of its pin; for an instance, its origin; for a point, the point itself), the origin of the symbol is drawn at that point, and the usage has no position of its own: PLACE and constraints on it fail. Without ORIENT, the solver chooses its orientation and writes it back (14.8), and a renderer that has no solution for it draws `"R0"` (8.5); it does not follow the direction in which wires leave the pin. ORIENT may be written: it turns the usage about that point, with an orientation of its own that does not follow the main object (11.2). When the main object has no coordinate point (such as a net), the usage has a position of its own, given with PLACE or by the solver, and its `::position` is its own origin. A usage has no name of its own; texts refer to the main object through the role, such as `${p}` (13.6).
  6. The body of a definition is always expanded, so the definition of a virtual part may have a body.
- In terms of Principles 2.2, a virtual part is a usage with a symbol whose roles are filled with host objects; a standard virtual part is a definition of the standard package that comes with a rule giving it a default symbol; a user-defined virtual part is a definition of the user's own, together with a symbol rule.
- The appearance is given by rules for each definition, such as `# @SELECT-USAGE $GROUND => @SET-SYMBOL $GROUND_SYMBOL`. Changing the drawing does not change the terminals or the wires on them.
- Whether a usage is a virtual part depends only on whether it has a symbol.
- Note: a definition that should be drawn but is given no symbol is reported only when one of its usages is placed, constrained or wired.

12.2 Creating virtual parts

```
$GROUND @ATTACH &vss => @PLACE(24, 38) => %g0;
&vss @WIRE {#M5:3 %g0:p}
```

- In `$D @ATTACH &net`, the objects after the definition are matched to the role list of the definition. It is written with ATTACH and read as "add this on the net" (9.4). A virtual part with several terminals is like an instance line of the netlist: for a definition `$DIFF_PORT` with the roles a and b, `$DIFF_PORT @ATTACH &inp(a) &inn(b)`, or by order, `$DIFF_PORT @ATTACH &inp &inn`.
- The roles are usually filled with nets: the terminals of a virtual part are on nets, like those of an instance, and its relation to a terminal (where it is drawn, what it connects to) is given by wires and constraints (12.3). When

the first role is filled with a terminal, the virtual part is drawn right on the pin of that terminal (12.1 item 5), such as `$GROUND @ATTACH #M5:3`. The kinds of object a role takes can be declared in the definition (9.4), such as `roles=[["p", ["net", "terminal"]]]`.

- This association is descriptive only: it makes no electrical connection and is not checked (Principles 2.2).
- A net can have any number of virtual parts, such as five grounds and two probes. Usages have no key: each application makes one more usage (5.8).
- A single virtual part can be given another symbol, such as `%g1 @SET-SYMBOL $EARTH` with a symbol `$EARTH` of one's own.
- ATTACH has no parameters of its own other than `priority` (14.8). The direction of a port belongs to the content of the standard package (12.4); the name shown is that of the main object, and to show another text, write a label (13.3). The parameters of the definition are passed through ATTACH as usual (9.4).
- `%g0` is bound only once in a level (6.6); to change its position, write `%g0 @PLACE(…)`.

12.3 Positions relative to a terminal

Parameters cannot hold host references, so a position relative to a terminal is written as constraints:

```
{%g0:p #M5:3} @ALIGN("x");
%g0:p #M5:3 @DISTANCE("bottom-to-top", exact=2)    /* the pin of the ground 2 below the source of
M5 */
```

To draw a virtual part right on a pin, with no distance and no wire, fill its first role with the terminal (12.1 item 5).

12.4 Ports

```
$IO_PORT @ATTACH &inp => @PLACE(0, 20) => %p_inp
```

- The name of a port of a subcircuit is the name of the net inside it, so a port tag is associated with the net `&inp`.
- A port itself (`#:inp`) is not drawn; it is shown only by a virtual part. The standard package adds port tags through a set, leaving out the nets that already have a port tag written explicitly (10.5); the solver decides their positions.
- The direction of a port is only a matter of display: the netlist has no directions, and the user decides which symbol is drawn. Which port definitions and symbols there are belongs to the content of the standard package.
- **Orientation of port tags.** A port tag has a direction: the pin of `$IO_PORT_SYMBOL` in the standard package, for example, is at one end, and the tag extends the other way. Without ORIENT, the solver chooses the orientation from the directions of the wires, aiming at few bends. This is a goal of the solver, not part of the language (14.8), so the ORIENT of a port tag is usually not written by hand.

12.5 Standard virtual parts

The standard virtual parts belong to the standard package, which is not part of the language standard (16.3). The standard package provides, for example, a ground `$GROUND`, a supply `$SUPPLY`, a port tag `$IO_PORT` and a no-connect mark `$NO_CONNECT` (12.6), with their symbols, the rules that give these symbols, and the fill (10.5).

For one net vss, three ways of writing give three different drawings:

1. `&vss @ADD-TO $GROUND_NET_SET`: each terminal on vss without a wire gets a ground below it with a short wire (the fill of the standard package, 10.5). This is how textbooks usually draw it: a ground symbol under each grounded terminal.
2. `$GROUND @ATTACH &vss => %g`: one ground for the whole net, placed by the solver. The terminals are connected to its pin by wires, such as `&vss @WIRE {#M5:3 %g:p}`, or by ROUTE for the whole net.
3. `$GROUND @ATTACH #M5:3`: the ground is drawn right on the pin of the source of M5, with no wire (12.1 item 5).

- A supply is written the same way, with the set `$SUPPLY_NET_SET` and the definition `$SUPPLY`. Net 0 is in `$GROUND_NET_SET` by default (10.1).
- For the third way, the fill of the standard package takes the terminal out, so no second ground is added (10.5).
- Ports: the standard package adds a port tag for each port net of a subcircuit (10.5). To place it, constrain it or change the name it shows, write it out: `$IO_PORT @ATTACH &out => %pout`, with wires to `%pout:p`, and

`%pout @TEXT("V_{OUT}", key="name")` for the name (13.3). Nets that should have no port tag are taken out of `$PORT_FILL_SET`.

12.6 No-connect marks

```
$NO_CONNECT @ATTACH #M1:4    /* terminal 4 of M1 is left unconnected on purpose */
```

- A no-connect mark is the definition `$NO_CONNECT` of the standard package, whose role takes a terminal; there is no keyword for it. Its main object is a terminal, so the cross is drawn on the pin (12.1 item 5). The standard package gives it a cross symbol with its origin at the centre of the cross.
- The usage records that the terminal is left unconnected on purpose: `# @SELECT-USAGE $NO_CONNECT => @SELECT-TERMINAL` selects these terminals.
- Checks (other host terminals on the net, a wire touching the terminal) are not required (Principles 2.4); an implementation that knows the standard `$NO_CONNECT` may give a warning.
- A no-connect mark does not count as a wire (10.2): a terminal that has only the mark is still `wired=false`. For a fill by terminal to skip these terminals, subtract the selection above, as in the difference of 10.5.
- Note: a no-connect mark is not a keyword because it has no use for an implementation other than drawing: the solver does not read it, and how it is drawn can be written in a definition and a symbol (1.5 item 7).
- Net labels are not virtual parts; see 13.3.

13 Wires and text

13.1 Wires: WIRE

```
&x @WIRE(mode="straight") {#M1:1 (10, 37) #M3:1} => @SET-STYLE $signal => %w13;
&out @WIRE(mode="manhattan") {#M2:1 #M4:1};
&tail @WIRE {#M1:3 #M5:1}                    /* no mode: "auto", routed freely */
```

- WIRE takes a net (or a set of nets, which makes a bus; see below) and an ordered list of points, at least two. A point is a terminal (including a terminal of a virtual part) or a point object. Points may repeat; when the first and the last point are the same, the wire is a loop.
- **Routing mode**: the wire passes through every point of the list; how it runs between two neighbouring points is set by the parameter `mode=`. For neighbouring points A(x1, y1) and B(x2, y2), with dx = x2 − x1 and dy = y2 − y1:

`mode`	From A to B
`"auto"` (default)	Decided by the solver, which may optimize it for the situation and writes the route back (see below, 14.8); before the wire is solved, a viewer draws it by its own settings
`"straight"`	One straight segment
`"horizontal-vertical"`	A ⟶ (x2, y1) ⟶ B
`"vertical-horizontal"`	A ⟶ (x1, y2) ⟶ B
`"manhattan"`	If |dx| ≥ |dy|: half of dx across, all of dy up or down, then the other half across: A ⟶ (x1 + dx/2, y1) ⟶ (x1 + dx/2, y2) ⟶ B; otherwise the same with x and y exchanged
`"manhattan45"`	If |dx| ≥ |dy|: (|dx| − |dy|)/2 across, then a 45° diagonal (|dy| along each axis), then the rest across; otherwise the same with x and y exchanged

  - Every mode except `"auto"` is a strict rule, and the viewer draws it as defined. When dx or dy is 0, every mode gives one straight segment. When |dx| = |dy|, `"manhattan"` takes the case |dx| ≥ |dy|, and `"manhattan45"` is one diagonal segment. The bends that a mode computes keep their exact values and are not rounded; they may fall on half units (11.7).
  - The bends added by a mode are not objects: they cannot be referenced, and no junction can be put on them. To branch off at a bend or to draw a dot there, write the bend as a point. A free point `(_, _)` first gets its coordinates from the solver and is then connected according to the mode.
  - A `mode` value that is not in the table is an invalid parameter value: the wire fails.
- **Style parameters**: WIRE also takes the line style properties as parameters: `linecolor`, `linestyle`, `linewidth`, `cap`, `join` (9.3). They are the direct style of the wire (8.3), such as `@WIRE(linestyle="bold")` for a supply rail or `@WIRE(linestyle="dashed")` for a substrate connection.
- **Rewriting a wire.** Given a wire in place of a net, WIRE makes no new wire: it rewrites that wire. The statement is read as a new WIRE would be, and its list of points and its mode replace those of the wire; the net of the wire stays. Style parameters are direct style of the wire (8.3).
  - The rewrite has a key, the wire: a later one replaces the list and the mode of an earlier one (8.4); of two segments of one pipe, the later segment is the later one (5.5). The style parameters of every rewrite stay direct style of the wire and are resolved by 8.3, property by property. A rewrite is not a geometric fact (8.5).
  - A rewrite that fails, such as one that lists a terminal without a pin (9.2.4), is an error and has no effect, its style parameters included; the wire keeps its earlier list and mode: those of an earlier rewrite, or those it was made with (5.8). Each rewrite that fails is reported, also when a later one replaces it.
  - Like a new wire, a rewrite counts from step 3 of 8.2: `wired=` (10.2) reads the points of the last rewrite, also when that rewrite fails later, after step 6 (7.1).
  - The net slot takes a net, a set of nets or one wire; a set that contains a wire is of the wrong kind.
  - The solver writes the route that it chose for each `"auto"` wire this way (14.8). It should keep, in order, the points that the wire has in the authored records, after their last rewrite of it, add only bends written as coordinates between them, and write `mode="straight"`: the connections of a wire are written by its author, and the solver decides only its geometry.

  - An editor in which the user reshapes a wire writes the same in its own record; the solver then keeps the wire as rewritten.
  - To fix the route in the statement itself, write the bends as points and use a mode other than `"auto"`; an exporter that keeps the exact shape writes `mode="straight"` and lists every bend.
- **Several terminals on one net** (Principles 2.6) are connected by several wires that share points. Wires are connected only when they share the same point object (a named point) or the same terminal; equal coordinates do not connect them (Principles 2.2). There is no statement that writes a tree, and no graph object.
- **T junction**: the through wire lists the junction point as one of its intermediate points and need not be split in two; the branch starts at that point (see the example in 13.2). There is no "point on a segment of a wire".
- **Pins that touch**: when two pins are drawn at the same point with no wire between them, a wire is still written to connect the two terminals, such as for a ground symbol drawn right on the pin of a capacitor: `&0 @WIRE(mode="straight") {#C1:2 %g3:p}`. Its two ends coincide and its length is 0, so nothing is drawn, but it states the connection: the terminals count as wired (10.2), and the standard package does not add another ground. Equal coordinates alone do not connect (Principles 2.2). An exporter that converts a schematic in which pins connect by position writes such wires; where a wire runs straight through a pin or a junction, it lists that point as an intermediate point (T junction); where a junction has only one wire and lies right on a pin, it connects the wire to that terminal directly.
- **The net is only stated**, and it must be written. The nets that the endpoints are actually on need not agree with it, and this is not checked (Principles 2.2); the net is not derived from the endpoints either, and the endpoints may be on different nets.
  - When a terminal in the list is not on the net of the wire (for a bus wire: not in its set of nets), this gives a warning: most likely the terminal is mistyped, such as `#R1:1` for `#R1:2`. The net of a terminal is known (a terminal of a virtual part is on the net of the object that fills its role); a point has no net and is not checked. The wire is valid and is drawn (Principles 2.2, 7.6).
- **Buses**: the net slot takes a net, or a set of nets as a whole. Given a set, the wire is a bus wire, also when the set has one member. A netlist has no buses (6.11); a bus is only a way of drawing several nets as one line.

```
* A 4-bit bus between two blocks, with two taps
* @RNL(type="header") { @VERSION("0.3")}
.subckt blk D<3> D<2> D<1> D<0>
RL D<3> D<0> 1k
* @RNL {
*   $BLK_SYM = @DEFINE-SYMBOL {
*     (-3, 0) @PIN(terminal=["D<3>", "D<2>", "D<1>", "D<0>"], label="D<3:0>", direction="left");
*     @RECTANGLE {(-3, -2) (3, 2)}
*   };
*   # @SET-SYMBOL $BLK_SYM                                        /* one bus pin for four
ports (9.2.4) */
* }
.ends blk
XA D<3> D<2> D<1> D<0> blk
XB D<3> D<2> D<1> D<0> blk
RT3 D<3> 0 1k
RT2 D<2> 0 1k
* @RNL {
*   %t3 = (20, 10); %t2 = (24, 10);
*   %bus = &D<3:0> @WIRE(linestyle="bold") {#XA:D<3> %t3 %t2 #XB:D<3>};   /* a bus between the
bus pins of XA and XB */
*   %bus @TEXT("D<3:0>");                                          /* the name of the bus,
written out */
*   &D<3> @WIRE {%t3 #RT3:1};                                      /* a tap: a single wire
that shares %t3 with the bus */
*   &D<2> @WIRE(mode="straight") {#RT2:1 (23, 9) %t2}              /* its last stretch
enters the bus at 45° */
* }
.end
```

  - The order of the set is the order of the bits, and duplicates are kept; sets written directly inside the set are taken down to their innermost members, in order. A member that is not a net makes the wire fail (wrong kind). An empty set fails too: a wire carries at least one net (5.2.1).
  - The net slot is strict (7.3): when one bit cannot be found, the whole wire fails, as when a single net cannot be found.
  - Points, `mode=`, style parameters, junctions and T junctions are as for other wires. There is no BUS keyword: an implementation draws a bus as it draws a wire, and drawing it bold is a matter of style (1.5 item 7). An importer may write `linestyle="bold"` directly; the standard package makes bus wires bold by default with the condition `bus=true` of SELECT-WIRE (10.1, 16.3).
  - **Taps** need no mechanism: a single wire that shares a point with a bus is a tap, and the bus lists the tap point as an intermediate point (T junction). A short 45° stretch into the bus is only the way the last stretch of the single wire runs. A tap symbol is a definition applied at the tap point and belongs to the standard package (16.3). The name of the bit is a label `${net}` on the single wire (13.3). Whether the net of the single wire is in the bus is not checked (Principles 2.2); an implementation may give a hint.
  - The attribute `net` of a bus wire is the names of its nets in order, joined with `,` and no spaces, such as `D<3>,D<2>,D<1>,D<0>`; to show `D<3:0>`, write that text. Implementations do not shorten the list: shortening depends on how the host writes names and has no unique result.
  - So `{&a &b} @WIRE {…}` is one bus wire, not two wires drawn on top of each other.
  - Several host instances drawn as one (an iterated instance) are written as a bundle (11.8).
- A terminal that a wire connects to must have a pin in its symbol; otherwise the wire fails (9.2.4).
- Lines that touch, cross or overlap are only geometry and make no connection (Principles 2.2). Wires that cross without sharing a point get no dot; small arcs at crossings and the like are a matter of style and of the implementation.
- Each WIRE given a net is a new object without a key: written twice, it is two wires (5.8). To refer to a wire from another statement, name it; the record of the solver refers to a wire without a name by its number (5.9). An endpoint that is a terminal moves with its device; a point written with coordinates does not move; a named point can be constrained.
- **Connections not drawn**: how a connection is shown that has no wire in the RNL (routed by the solver and written back as WIRE, a flight line, a label) is decided by the house style of the solver and the viewer and by the house-style packages in use; the language does not specify it. A house-style package can make labels or wires with rules, and these are RNL. To have a net drawn as wires with the routing left to the solver, write ROUTE (13.7).

13.2 Junctions

```
%j1 = (10, 30);
&x @WIRE {#M1:1 %j1 #M3:1};          /* the through wire passes %j1 */
&x @WIRE {%j1 (30, 30) #M4:2};       /* the branch starts at %j1 */
$JUNCTION @ATTACH %j1                /* draws the dot */
```

- A branch is several wires that share one point (13.1).
- Junction dots are written explicitly: a dot is drawn only where `$JUNCTION` is applied; a renderer does not add dots by itself.
- A dot is a usage, not a keyword. `$JUNCTION` is a definition of the standard package with one role, which takes a point or a terminal; the dot is drawn at the coordinate point of this main object (12.1 item 5), as for `$NO_CONNECT` (12.6) and the current arrow (15.2). It can be applied to a terminal when two wires meet at its pin. The standard package gives these usages a filled dot as their symbol: the size of the dot comes from the symbol, and its colour from the `linecolor` of the style of the usage; there is no separate setting for the size of dots.
  - Applying `$JUNCTION` twice to the same point gives two dots on top of each other, which look like one.
  - All junctions are selected with `# @SELECT-USAGE $JUNCTION`.
  - The drawing order follows the position of the rule that assigns the symbol (11.6). The rules of the standard package come first, so the dots are drawn before the wires; a dot has the colour of the wires and is wider than they are, so this makes no visible difference, and no LAYER is needed.

- Dots are not derived from the number of wires that meet at a shared point: real schematics have branch points with only two wires that are drawn with a dot, and a derived dot would need yet another way to remove it. An exporter writes the dots that the original shows. An implementation may give a hint where three or more wires meet at a shared point without `$JUNCTION`; this standard does not require it.
- **Dots added by the solver**: the output of the solver is RNL too and may contain usages of `$JUNCTION`. The solver writes the dots at the branch points of ROUTE (13.7); where wires written by the user meet three or more at a shared point without `$JUNCTION`, the solver may add one in its output as well. A shared point is always a named point or a terminal (equal coordinates do not count), so the dot can be written. Whether the solver adds dots is a setting of the solver, not a rule of the language.
- A dot is only drawing: it makes no connection and is not checked (Principles 2.2). A point with `$JUNCTION` and no wire still gets its dot.

13.3 Text: TEXT

```
(50, 18) @TEXT("5T OTA testbench");                          /* an annotation: attached to a point
*/
(1, 1) @TEXT("see here") => @PLACE(2, 2);                    /* attached to (1, 1); the text itself
is at (2, 2) */
#M1 @TEXT("W/L=${w}/${l}", key="wl") => @PLACE(12, 18);       /* a label: attached to M1, with the
key wl, so it is #M1::wl */
#R1 @TEXT("R_{D}", key="name");                              /* a key the symbol has: new content,
the position still from the symbol */
#M1::name @PLACE(13, 18);                                    /* the name of M1, dragged in an
editor */
%w13 @TEXT("${net}");                                        /* the net name on a wire; no key;
placed by the implementation */
&out @TEXT("${name}");                                       /* shows the name of the net out;
placed by the implementation */
# @SELECT-TERMINAL(wired=false) => @TEXT("${net}", key="netname")    /* net names on terminals
without wires */
```

- **Attachment**: TEXT takes a point, or an object that is drawn, and the text is attached to it. What it is attached to decides three things: whose attributes `${…}` reads (13.6); where the text goes when its position is not given; and, when it has a key, which object the text is a sub-object of.
  - A point (`(x, y)`, a named point, a `::position`): the text is an annotation (13.4). It can refer only to the attributes `x` and `y` of the point.
  - An object: the text is a label. The object can be an instance, a drawn usage, a bundle, a terminal that has a pin, a wire or a graphic. It can also be a net: a net itself is not drawn, and the implementation puts a text attached to a net next to one of its wires or terminals. This is how a file says which nets show their names.
  - A usage without a symbol, or a terminal without a pin, has no meaning for drawing: given one, the TEXT fails (12.1, 9.2.4).
  - Given a set, TEXT attaches a text to each member (5.2.1), as in the last line of the example.
  - In a symbol body, a text attached to a point is fixed text of the symbol; a text attached to a lone `?` is attached to the object drawn with the symbol (see below).
- **Content**: `text` is required, also when a key is given; without it the TEXT fails. It can be given by position, first, as in `@TEXT("${name}", key="name")`. There is no default content: to show the name, write `"${name}"`. Markup and attribute references are in 13.6.
- **Position**: a text has a position of its own, which is not the point or object it is attached to.
  - The position of a text starts out free, like `(_, _)`. When no PLACE, `dx`, `dy` or constraint fixes it, the solver decides and writes the position back (14.8); without a solution, the viewer decides: a text attached to a point usually goes at that point, and a text attached to an object next to the object; either may be moved away to avoid overlaps.
  - PLACE sets the position of the text itself. So `(1, 1) @TEXT(…) => @PLACE(2, 2)` is not a contradiction: the first part attaches the text to (1, 1), the second puts the text at (2, 2). To put an annotation exactly somewhere, write PLACE.

- `dx`, `dy`: offsets, taken only by TEXT. When what the text is attached to has a coordinate point (a point, the origin of an instance, the point where a usage is drawn, the pin of a terminal), the text goes at that point plus (dx, dy). Wires and nets have no coordinate point: an offset on a text attached to one makes the TEXT fail.
  - The text still has one position, the one with the offset; constraints and the solver use it, and there is no position "before the offset". When the attached object is moved or placed by the solver, the text moves with it; when other constraints move the text, the attached object moves with it. An offset is the same as one exact DISTANCE for x and one for y, both in the increasing direction (`"left-to-right"`, `"bottom-to-top"`; 14.5), written on the text.
  - Offsets go along the axes of the scene and do not turn with the orientation of the attached object, since texts in the scene do not turn with objects. In a symbol body they can be written too; there they are in local coordinates and transform with the instance, like a local position written with `=> @PLACE(…)`.
  - **Offsets and PLACE replace each other**: on each axis, the position of a text is either absolute (the x or y of PLACE) or an offset (`dx` or `dy`), never both. On the same axis of the same text, the kind written later (by the position of 8.5) takes effect, and the kind written earlier has no effect at all: after an offset, the absolute position on that axis is free and follows from the offset; after a PLACE, the offset on that axis no longer applies. This is replacement, not a conflict: priority plays no part, and no hint is given. For example, after `dx` and `dy`, a later `#M1::name @PLACE(x=13)` makes x the absolute 13, `dx` no longer applies, and y still follows `dy`; an editor writes this when a text placed by offset is dragged. A position given in the symbol comes first (see below), and a PLACE or an offset in the scene replaces it.
  - Whichever takes effect is an ordinary geometric fact: the x and the y of an offset are one unit each, their position is that of the TEXT statement that writes them, and their priority is the default of the record head (14.8); they are resolved with other facts by 8.5. Several PLACEs on the same axis are resolved among themselves by 8.5. For a text with a key, `dx` and `dy` are overridden separately (8.4).
- A text can be a member of constraints by its own position (14.1). The `::position` of a text is its own position: `%n1::position @PLACE(…)` is the same as `%n1 @PLACE(…)`.
- Texts in the scene do not turn with the orientation of what they are attached to; to turn one, write `rotate=`. When the scene supplies both axes of a text's position, by PLACE or offsets, neither axis uses the symbol's local position: the text does not inherit the attached object's orientation, and `halign` and `valign` are interpreted in the scene's directions.
- The solver writes back the position it chose (14.8): by the key (below) or the name of the text, or otherwise by its number (5.9). A text without a key that a generation rule makes directly (13.3, 10.5) cannot be referred to: the solver leaves its position to the viewer. A text in the body of a usage that a rule makes is referred to by number once the solver writes the usage as an explicit statement (10.5, 5.9).

- **Key `key=`**:
  - Without a key, the text is a text of the attached object that has no key; written again, it is another text, as with wires (13.1). It can be named with `%x =` and referred to and changed in its level. Texts made by rules without a key have no handle, so the solver cannot write back their positions.
  - With a key, the text is the sub-object `object::key` of the attached object, such as `#M1::wl`. The same object with the same key is the same text, and later statements override it parameter by parameter: the size, the alignment, the rotation and so on are each overridden on their own, and parameters not written stay as they were (8.4). The content is written in every statement (`text` is required), and the later content always replaces the earlier; to change only the alignment or the rotation, write the content again, as in `#M1 @TEXT("${name}", key="name", halign="end")`. The position follows the rules of PLACE (8.5). Other records, editors and the solver find the text by this handle.
  - A text attached to a point takes no `key=`, since a point has no sub-objects: with a key, the TEXT fails.
  - The names after `::` are looked up in a fixed order, and keys come last (6.5): the role names and local names of a usage, then their `definition=` aliases, then the standard sub-object names and their aliases, then the keys of texts. So a key must not be spelled like any name that comes before it: a standard sub-object name (`position`, `net`) or an alias of one declared for the file (such as `位置` of `rnl-zh`); a role name or local name of the usage the text is attached to; or a `definition=` alias of one of these. Such a key is an error, like the name clashes of 6.5. For a text written in a symbol, the aliases are those of the file in which the symbol is written. `name` and `value` are not standard sub-object names, only common keys.

- **Texts in symbols**:
  - In a symbol body, a lone `?` is the object drawn with the symbol: M1 when M1 is drawn, `%g0` when the usage `%g0` is drawn. It can only be the object of TEXT; anywhere else it fails, such as `? @PLACE(…)` in a symbol, which has no meaning.
  - `? @TEXT("${name}", key="name") => @PLACE(1.5, 1.5)`: the text is attached to the object drawn with the symbol, and `${…}` reads its attributes. This PLACE is in local coordinates and transforms with the instance (see the direction of texts in 9.2.5). Without PLACE, the implementation puts the text next to the object.
  - A text with a key is a sub-object of the object drawn with the symbol, such as `#M1::name`. The texts in the symbol come first, and a text with the same key in the scene overrides it parameter by parameter: `#R1 @TEXT("R_{D}", key="name")` changes only the content and keeps the position from the symbol; `#M1::name @PLACE(13, 18)` changes only the position and keeps the content from the symbol. A position given in the scene is in scene coordinates and no longer turns with the instance.
  - The keys that a symbol gives exist only once symbols are assigned (8.2 steps 4 and 6). When the symbol of M1 has no text with the key `name` and the scene writes none, `#M1::name` fails. To be independent of the symbol, write a TEXT with that key in the scene.
  - A text attached to a point in the symbol is fixed text, such as `(-2, 2) @TEXT("+")` in an op-amp symbol, and cannot be changed for one instance.
- **Rules**: a rule such as `# @SELECT-TERMINAL(wired=false) => @TEXT(…)` selects terminals by their symbols, so it is evaluated in step 5 with the one-by-one generation (8.2). When an implementation adds flight lines or labels by itself, it skips terminals that already have a label.
- Which net names are shown and where, and whether connections without wires are shown with labels or flight lines, are decided by the implementation by default: they are part of its house style (Principles 2.7). Showing labels instead of wires changes the layout, so the solver and the renderer use the same setting. RNL writes these only when needed: to state an intent (without positions), or to keep positions already decided.
- There are no positions along a wire: the coordinates of a wire are not fixed themselves.
- Parameters: `text` (required; may be given by position), `key`, `dx`, `dy`, `halign`, `valign`, `rotate`, and the text style properties `textcolor`, `size`, `font`, `bold`, `italic` (9.3); a text attached to a point has no `key`. For `halign`, `valign` and `rotate`, see 9.2.5. The style parameters are the direct style of the text (8.3); how a text gets its style is in 11.3.

13.4 Annotations: graphics in the scene

```
(50, 18) @TEXT("5T OTA testbench") => @SET-STYLE $note => %n1;
%f1 = @RECTANGLE {(0, 0) (40, 30)};
%n1 @PLACE(60, 18)                          /* moves the text; may also be written in another
record */
```

- LINE, RECTANGLE, CIRCLE, ELLIPSE, ARC, BEZIER and TEXT can be written directly in the scene, and so also in the body of a general definition. In a symbol body their coordinates are local; in the scene they are scene coordinates. Where they are written decides this; the keyword is the same.
- The note objects of Principles 2.2 (free-form text, graphics, frames and other objects that are only drawn, with no terminals and no electrical meaning) are the annotations of this standard: graphics written in the scene and TEXTs attached to points.
- PIN can be written only in the body of DEFINE-SYMBOL: a pin binds a terminal of the object drawn with the symbol, and the scene has no such object.
- A TEXT attached to a point is an annotation (13.3): it can refer only to the attributes `x` and `y` of the point, and it takes no `key=`, since a point has no sub-objects (with a key, the TEXT fails). To show the name or value of an object, attach the TEXT to the object, which makes it a label.
- The position of a graphic is given by its points. A point can be any point object: a named point, a free point with constraints, or `#M1::position`, which follows the device. PLACE does not take a graphic itself: to move, from another record, a RECTANGLE, CIRCLE or ELLIPSE written with a centre, move its `::position`; LINE, ARC, BEZIER and the graphics written with two points have several points, so name the points and move them. TEXT is the exception: it has a position of its own, the point it is attached to only gives its default place, and PLACE sets the position of the text directly (13.3).

- Appearance: graphics can take style parameters directly (9.2.5) and also take SET-STYLE, with the two tiers of 8.3; a property given by neither has the default of the renderer.
- A frame is a RECTANGLE, or a LINE whose last point is its first. There is no graphic that encloses a group of objects: draw a RECTANGLE and constrain its corners to lie outside the group, such as `%tl %stage1_set @DISTANCE("left-to-right", exact=1)` (a set counts by its extent, 14.5); what is enclosed is the points of the members plus the margin. A viewer may draw a GROUP as a frame; that is up to the viewer (11.5). There is no `::box`.
- The drawing order follows layers and positions (11.6).
- **Class**: whether something is an annotation follows from its kind of object: graphics in the scene are annotations, so a renderer can show or hide them as a class. SELECT-ANNOTATION selects all graphics in the scene, including those expanded from the bodies of general definitions (such as frames drawn by a house-style package); graphics in symbols, labels, wires and usages (including junctions) are not annotations. Finer classes (explanations, title blocks, decorations drawn by a package) are named sets, and a graphic can be in several sets. Showing and hiding is up to the viewer; the language has no `visible`.
- A text on a device is a TEXT attached to the device, which is a label (13.3). A TEXT attached to `#M1::position` is an annotation; by default it also follows the device, but it can refer only to the coordinates of that point, and it has no key.

13.5 (Not used)

This section number is not used.

13.6 Text markup and attribute references

```
#R1 @TEXT("R_{D}", key="name");                              /* subscript */
(50, 18) @TEXT("\textbf{5T OTA} testbench\\VDD = 1.8 V");    /* bold, two lines */
#M1 @TEXT("W/L=${w}/${l}", key="wl");                        /* host parameters */
$NMOS_M = @DEFINE-SYMBOL {
  $NMOS;
  ? @TEXT("${model}", key="model") => @PLACE(1.5, -1.5)      /* the model name, in a symbol */
}
```

- **Markup** is a subset of the text mode of LaTeX, not full TeX:

Markup	Meaning
`_{…}`, `^{…}`	Subscript, superscript
`\textbf{…}`, `\textit{…}`	Bold, italic
`\overline{…}`	Overline, such as the active-low `\overline{RST}`
`\frac{…}{…}`	Fraction: numerator above, denominator below, a bar between, such as `\frac{${w}}{${l}}`; a viewer that cannot stack them shows "numerator/denominator"
`\alpha` … `\omega`, `\Gamma` … `\Omega`	Greek letters, with the names of LaTeX
`\pm`, `\times`, `\cdot`, `\infty`, `\approx`, `\neq`, `\leq`, `\geq`, `\ll`, `\gg`, `\propto`, `\parallel`, `\angle`, `\circ`, `\to`, `\uparrow`, `\downarrow`	Common symbols
`\\`	Line break
`\_`, `\{`, `\}`, `\$`, `\%`, `\&`, `\#`, `\textbackslash`	The characters `_`, `{`, `}`, `$`, `%`, `&`, `#`, `\` themselves

- Greek letters: lower case `\alpha` to `\omega` (LaTeX has no `\omicron`; write o), and the variants `\varepsilon`, `\vartheta`, `\varpi`, `\varrho`, `\varsigma`, `\varphi`; upper case only the letters that differ from Latin ones: `\Gamma`, `\Delta`, `\Theta`, `\Lambda`, `\Xi`, `\Pi`, `\Sigma`, `\Upsilon`, `\Phi`, `\Psi`, `\Omega`. A symbol not in the table is written directly as a Unicode character (strings may hold non-ASCII characters, 3.7).
- Markup can be nested, such as `\textit{V}_{\textit{DD}}` and `I_{D_{1}}`. How the scripts of `V_{in}^{+}` are set (stacked or one after the other) is up to the viewer.

- **Subscripts and superscripts need braces**: a lone `_` or `^` is an ordinary character. This differs from LaTeX, where the m of `g_m` is a subscript: names and descriptions often contain `vin_1` or `bias_ctrl`, which that rule would spoil. In return, `g_m` is shown as written.
- Other characters are shown as written, including a lone `{`, `}`, `$`, `~`, `%`, `&` or `#`, which have other meanings in LaTeX.
- **Backslash**: `\\` is a line break, as in LaTeX; a backslash itself is written `\textbackslash`. A backslash that is not followed by a known command is shown as written and gives a warning, such as `C:\temp` (`\temp` is an unknown command; see the next item); to avoid the warning, write `C:\textbackslash{}temp`. `\textbackslash` is needed only where the backslash is directly followed by `\`, `{`, `}`, `_`, `^`, `$`, `%`, `&`, `#` or the name of a known command. RNL strings have no backslash escapes (3.7): `"\textbf{V}"` is written just so, not as `\\textbf`.
- An unknown command (`\` followed by letters that are not in the table, such as `\sqrt` or `\alpah`) and unbalanced braces are shown as written and give a warning; the text remains valid.
- **After a command name**: a command name is the run of letters directly after `\`. Spaces after the name of a known command are dropped and not shown, and an empty `{}` right after it only separates and is not shown either. So `10 \mu A` and `10 \mu{}A` both show 10 µA; to keep the space, write `\mu{} A`; `\textbackslash{}temp` shows \temp. Spaces after the escaped characters such as `\_` and `\{` are kept.

  Note: support for more of TeX in a later version only allows more. Math mode would then be written `\(…\)`, not `$…$`, since `$` is used by attribute references.
- **Fractions**: `\frac` takes two groups in braces directly after it; when one is missing, it is unknown markup, shown as written with a warning. Numerator and denominator may contain markup and `${…}`.
- How far scripts and fractions are raised or lowered and how much they are made smaller is up to the viewer. The font, and bold and italic for the whole text, come from the style or from the parameters `font`, `bold`, `italic` (9.3).
- Markup is read in the `text` of TEXT and in the `label` of PIN (such as `\overline{CS}`). Other string parameters (`key`, enumeration values, selection conditions) do not read markup.
- Note: in import and export, the `_{…}` and `^{…}` of KiCad are the same, and its `~{…}` is `\overline{…}`; an overline that Altium writes with `\` after each character is rewritten as `\overline{…}`.
- **Attribute references** `${name}` show an attribute of the object the text is attached to. To write `${` itself, write `\${`; everything from `\${` to its matching `}` is shown as written, and that `}` does not count as an unbalanced brace. The attributes are:
  - Instances: `name`; `value`, the value written on the host line, as written, empty if there is none (it does not fall back to the model name); `model` and `type`, as in selection conditions (10.1), so that `${model}` of an X instance shows the subcircuit name; and the host parameters, such as `w`. A host parameter with the name of a standard attribute is written `${#…}` (see below).
  - Terminals and wires: `net`, the net they are on, shown by its name. The `net` of a bus wire is the names of its nets in order, joined with `,` and no spaces (13.1). For a terminal of a bundle, it is the name of the net when all members are on the same net, and otherwise the names joined with `,` (11.8).
  - Nets: `name`.
  - Usages: the roles and the parameters of their definition. `${p}` is the object that fills the role p, shown by the rules below. A usage has no name of its own: the symbols of grounds, supplies and ports write `${p}`, which shows the name of the net when p is a net; when the usage is drawn on a pin (12.1 item 5), p is a terminal, which also shows the name of its net.
  - Bundles: an attribute that is the same for all members has that value; otherwise the values are joined with `,` (11.8).
  - Points: `x`, `y`, the final coordinates, that is, after solving.
  - Graphics: no attributes.
- **Attribute paths**: in `${a::b::c}`, the first name takes an attribute of the attached object, and each following `::name` takes an attribute of the value before it, such as `${p::x}` (the x of the role p when it is a point) and `${pair::in_p::w}`.

  - The value of an attribute can be an object (a role of a usage; the `net` of a terminal or wire) or text (`name`, `value`, a host parameter and so on). Text has no attributes, so no `::` may follow it.
  - A value on the path can also be a set, such as a set role. The rest of the path is then taken from each of its members, a set among them by its own members, in the order of the set, and the values are shown as a set is (below); when one member lacks the attribute, the text fails. So for a usage of `$CASCADE` whose first stage is two differential pairs, `${first::in_p::w}` shows `w` for both `in_p`, such as `1u,2u`.
  - In `${…}`, `::` takes attributes only, unlike `::` in statements, which takes sub-objects (6.5): `${m::name}` is the name of the instance m, while `#M1::name` in a statement is the text with the key `name` on M1 (13.3).
  - `${m::#type}` looks only for a host parameter, as `${#type}` does (see below).
- **Display**: when the last value of the path is text, it is shown as written. When it is an object: instances and nets show their names; terminals and wires show the name of their net, and a bus wire the names of its nets joined with `,`; bundles as described above; a set (such as a set role) shows its members, each in this way, joined with `,`. Points, usages and graphics cannot be shown: the text fails. So when the role p is a point, write `${p::x}`, not `${p}`.
- References give the text as written in the netlist, without evaluating it: `w=1u` shows 1u. What each dialect provides is defined by its host adapter; in Spectre, for example, the value of a resistor is the parameter `r`.
- The attributes listed above always exist; one that has no value is empty. A reference to an attribute name that does not exist makes the text fail.
- The referenced content is shown as written and is not read as markup: host names and values are not written in this markup.
- References go only down from the attributes of the attached object; they cannot reach an arbitrary object. There is no reference to another object such as `${R3:VALUE}` of KiCad.
- **Aliases**: attribute names in `${…}` also accept the aliases declared for the file, looked up as standard attribute first, then alias, then host parameter; `${#name}` looks only for a host parameter (16.5). The role and parameter names of a usage are RNL names: they accept only the `definition=` aliases of its definition, looked up at each step in the definition of the usage of that step, such as `${输入对::正输入::w}` (16.5).
- **Host parameters with standard names**: a standard attribute comes first, and a host parameter with the same name is written `${#type}` (`#` stands for the host in RNL). Any host parameter can be written this way; `${w}` is short for `${#w}` where no standard attribute and no alias has that name. Standard attribute names are compared exactly as written, host parameter names by the rules of the host: in a SPICE host, `${Type}` is not a standard attribute, so it is the host parameter type.
- **Formatting names**: the language has no way to format individual characters of referenced content. To show M1 as M_1, write the text itself, `#M1 @TEXT("M_{1}", key="name")`, which does not follow a renaming. A uniform convention such as "the digits at the end of a device name are subscripts" is part of the house style of the viewer.

13.7 Drawing whole nets as wires: ROUTE

```
&x @ROUTE;                                  /* x is drawn as wires; the solver decides the
branches and paths */
%jo = (_, _);
&out @WIRE {#M2:1 %jo};
&out @ROUTE;                                /* this wire is drawn as written; the solver adds the
rest */
$ROUTED_NET_SET = # @SELECT-NET;
$GROUND_NET_SET @REMOVE-FROM $ROUTED_NET_SET;
$ROUTED_NET_SET @ROUTE                      /* as a house-style package would write it: every net
except the ground nets */
```

- ROUTE lets a file ask for a whole net to be drawn as wires without saying where it branches and where the dots go.
- **Signature**: one slot, which takes nets; a set is taken member by member (5.2.1), and is not a bus. Parameters: `mode` (a hint for the solver, with the values of WIRE, default `"auto"`) and the style parameters of wires (13.1).

- **Meaning**: the net is drawn as wires in this level. All terminals on the net that have a pin (of devices, and of virtual parts attached to the net or to its terminals) are joined into one tree; where it branches and how it runs is decided by the solver. Wires already written are part of the tree, and the solver adds the rest.
- Note: a WIRE with an empty list of points does not mean this. A list of points may be computed, and a selection that happens to be empty would then silently connect everything; so ROUTE is a keyword of its own; this version has no optional slots either (5.3).
- **Written back by the solver**: concrete WIREs (with their bends and `mode="straight"`), the named points where they branch, and the junctions (13.2), with the style parameters of the ROUTE, go into the solver's own record. The ROUTE itself is the intent of the user and stays; the next solve computes it again (chapter 17).
- **Viewers that do not solve** draw flight lines or nothing, as the viewer decides (Principles 2.7).
- ROUTE is not a geometric fact and takes no part in conflicts (8.5). Terminals without pins are not in the tree. When a net has fewer than two points to connect, ROUTE does nothing.
- A house-style package chooses which nets are drawn as wires with sets and rules, as in the example; there is no keyword for how a net is shown.

14 Layout intent

14.1 Common rules

- A constraint makes a constraint object. A named constraint can be referred to, which helps to find and edit it, such as `%row1 = {#M1 #M2} @ALIGN("y")`; a constraint can also be anonymous.
- **Vocabulary**: the atomic constraints on positions are only ALIGN, ORDER, DISTRIBUTE and DISTANCE, besides PLACE, ORIENT and GROUP. Other relations are written as general definitions: symmetry (14.7), centring (a DISTRIBUTE of three members), rows and columns (ALIGN with ORDER), signal flow (14.6), mirrored pairs, spacings in proportion (several exact DISTANCEs). There is no LEFT or ABOVE (they are an ORDER or a DISTANCE with a direction), and no RATIO, NEAR or INSIDE. Relative orientation (the same orientation, mirror images) is ORIENT-RELATION (14.9); whether the gates of a pair face each other or face outwards is still fixed with an ORIENT, or with an ORDER on terminals (14.3). That symbols do not overlap, spacing constants and grids belong to the solver (11.7) and are not part of the vocabulary of constraints.
- **No positions relative to the whole drawing**: there is no keyword for "leftmost", "rightmost" or "topmost", and no extent of the whole drawing or of symbol outlines. Write an ORDER or a DISTANCE relative to a group of objects, and the solver finds the layout from it. For a port at the far right: `{#M6 #M7 #CL} %pout @DISTANCE("left-to-right", min=2)`, or `{#M7 %pout} @ORDER("left-to-right")` when M7 is the rightmost already.
- **Members** are objects that have a position: instances, virtual parts with a position of their own (12.1), bundles, points, TEXT (by its own position, 13.3), terminals, the centre of a RECTANGLE, CIRCLE or ELLIPSE written with a centre, the `::position` of a set, and sets (see below). Wires, nets, constraints, definitions, usages without a symbol and usages drawn at the coordinate point of their main object cannot be members: given one, the constraint fails (12.1). A position on a wire must first be written as a point (13.1).
- **Which point counts**: an instance counts by its origin, the origin of its symbol, which is also what PLACE places; a terminal by the position of its pin, which is the position of the device plus the rigid transform of the pin position in the symbol and cannot be moved on its own; a point by itself; a virtual part by its own position (12.1); a TEXT by its own position; a RECTANGLE, CIRCLE or ELLIPSE written with a centre by its centre. The centre of the box of a symbol is not used: the boxes of three-terminal and four-terminal MOS symbols differ, and a row of them would not line up. To align by pins, write terminals. Visual symmetry requires the origin to lie on the axis of symmetry of the symbol, which is up to the symbol library.
- **Sets as members**: a member of ORDER, ALIGN or DISTRIBUTE that is a set counts as one member, by its centre, which is its `::position`. The two slots of DISTANCE take objects or sets, and a set counts by its extent (14.5). Centres and extents are computed from the points of the members, nested sets down to the innermost objects, regardless of symbols. An empty set is not a member; a set that contains an object without a position makes the constraint fail. References are still spread one level (5.2): `{%a_set %b_set} @ALIGN("y")` joins the members of both sets and aligns them one by one. To count each set as one member, write `{{%a_set} {%b_set}} @ALIGN("y")`, or use `%a_set::position` and `%b_set::position`.

  Note: writing `{%a_set %b_set}` where `{{%a_set} {%b_set}}` is meant is a common mistake (5.2).
- Projections are only `"x"` and `"y"`.
- Constraints do not change the netlist. In a conflict, an atomic constraint is kept whole or discarded whole (8.5).
- The four atomic constraints and ORIENT-RELATION take the parameter `priority=` (14.8).

14.2 ALIGN

`{…} @ALIGN("y")`: the members are equal in the projection `axis`: `"y"` gives them the same y (a row), `"x"` the same x (a column). `axis` is required and can be given by position. With fewer than two members, the constraint holds trivially and does not fail: how many members a selection or a set has is not known when the statement is written.

14.3 ORDER

```
%rc_set = {#R2 #C2};
{#M1 #M1:2 #M2:2 #M2} @ORDER("x");              /* the two gates face each other */
{#R1 {%rc_set} #M1} @ORDER("left-to-right")     /* left to right; R2 and C2 count as one member,
by their centre */
```

- `{…} @ORDER(direction)`: the members are strictly monotonic in the projection of the direction. With fewer than two members, it holds trivially.
- **Direction**: the parameter `direction`, required, given by position first. Its values are:
  - `"left-to-right"`, `"right-to-left"`: increasing x, decreasing x;
  - `"bottom-to-top"`, `"top-to-bottom"`: increasing y, decreasing y (y goes up, 11.7);
  - `"x"`, `"y"`: only the axis is fixed; the order may be ascending or descending.

  DISTANCE takes the same values.
- A member that is a set counts by its centre (14.1). Only centres are compared, so the members of two sets may still interleave; to keep two sets apart, use DISTANCE (14.5).
- ORDER is strict: neighbouring members differ by at least one coordinate unit (11.7). For equal coordinates, use ALIGN.

14.4 DISTRIBUTE

`{…} @DISTRIBUTE("x")`: the differences between neighbouring members in the projection `axis` are equal; the step may be positive, negative or zero. The steps are between the points of the members (a set by its centre), not gaps between boxes. `axis` is required and can be given by position. With fewer than three members, the constraint holds trivially.

14.5 DISTANCE

```
#M1:2 #M1 @DISTANCE("left-to-right", min=1);      /* the gate is on the left */
#R1 #R2 @DISTANCE("left-to-right", exact=8)       /* R2 exactly 8 units to the right of R1 */
```

```
%stage1_set = {#M1 #M2 #M5};
%stage2_set = {#M3 #M4};
%stage1_set %stage2_set @DISTANCE("x", min=1);               /* the two sets do not interleave in
x; either may be on the left */
%stage1_set %stage2_set @DISTANCE("left-to-right", min=1)    /* all of stage 2 is to the right of
all of stage 1 */
```

- `A B @DISTANCE(direction, min=…, max=…)`: A fills the slot `from` and B the slot `to`; each is an object or a set.
- **Direction**: the parameter `direction`, required, given by position first, with the values of ORDER (14.3). Only that axis is constrained; this is not the Euclidean distance.
  - An increasing direction (`"left-to-right"`, `"bottom-to-top"`) constrains B − A;
  - a decreasing direction (`"right-to-left"`, `"top-to-bottom"`) constrains A − B;
  - `"x"` and `"y"` constrain |B − A|, and the bounds cannot be negative.
- **Sets count by their extent**: when A and B are sets, an increasing direction constrains the minimum of B minus the maximum of A, which is positive when all of B comes after A; a decreasing direction the reverse, the minimum of A minus the maximum of B; `"x"` and `"y"` constrain the gap between them, with either one first, which is negative when they interleave. A single object is an extent of one point and gives the same as the item before. Extents are computed from the points of the members (14.1). When one side is an empty set, the constraint holds trivially. So two sets that are not to interleave are written with DISTANCE, and nothing else is needed.
- At least one of `min`, `max` and `exact` must be written; `min` must not be greater than `max`.
- `exact=` is an exact distance, short for equal `min` and `max`: B − A = exact for an increasing direction, A − B = exact for a decreasing one, and |B − A| = exact for `"x"` and `"y"`, with either one first (it cannot be negative). It cannot be written together with `min` or `max`: that is an invalid parameter, and the statement fails. Tools write exact distances with `exact` (4.3).

14.6 Signal flow

Signal flow (Principles 2.4) is written as the usage of a general definition; there is no keyword for it. Whether the standard package provides such a definition, and under what name, is up to the standard package (16.3); the examples use `$FLOW`, which orders a path, given as a set, in a direction given as a parameter.

```
%rc_set = {#R2 #C2} @GROUP;
{#R1 {%rc_set} #M1} @AS $FLOW          /* the signal goes from R1 through the block of R2 and C2
to M1, left to right */
```

- Signal flow is a soft notion: not every circuit has a strict direction. So it is written as a constraint of low priority (14.8).
- A stage of the path that is made of several objects is written as a `{…}` of its own, such as `{%rc_set}`, and counts by its centre (14.1). A GROUP (11.5) only makes the solver treat it as one block, and may be left out.
- The direction is given as a parameter when the definition is applied (9.4): `{…} @AS(direction="top-to-bottom") $FLOW` runs from top to bottom (y goes up, 11.7).

14.7 Symmetry

Symmetry is written as the usage of a general definition. The axis of symmetry is only a free point, and no keyword is needed. The standard package provides `$MIRRORED`, with the roles `a`, `b` and `axis`: a and b at the same height and symmetric about the axis point, a on the left, with b oriented as a flipped left-right. As in every definition of the standard package, these constraints are of priority `"low"` (9.5).

```
%ax = (_, _);
%sym12 = #M1(a) #M2(b) %ax(axis) @AS $MIRRORED
```

- `$MIRRORED` does not fix the actual orientation: whether the gates face inwards or outwards still needs an ORIENT on one side (14.9). For a symmetric pair with the same orientation on both sides, write another general definition, without its ORIENT-RELATION.

14.8 Priority: PRIORITY

```
$DP_LAYOUT = @DEFINE(roles=["in_p", "in_n"]) {
  {?in_p ?in_n} @ALIGN("y", priority="low")                    /* a default layout, the first to
give way */
};
%dp2 = #M1(in_p) #M2(in_n) @AS(priority="high") $DP_LAYOUT;     /* this pair keeps the default
layout strictly */
#M1 @PLACE(12, 20, priority="high")                             /* fixes the position of M1 */
```

```
# @SELECT-USAGE $DP_LAYOUT => @PRIORITY("disabled")    /* a house-style package turns the default
layout off, to apply its own */
```

- **Four values**: `"disabled"`, `"low"`, `"normal"`, `"high"`. A fact without a priority is `"normal"`, whether it is written in a record of the user, in a package or in the body of a definition.
- **Ranked, not weighted**: in a conflict, priorities are compared first, and the higher one is kept; with equal priorities, the one written later is kept (8.5). Constraints are not summed with weights.

  Note: with weights, ten constraints of low priority could outweigh one of high priority, and solvers differ in how they measure how far a constraint is violated.
- **Every fact that is kept is hard**: `"low"` means "discarded first in a conflict", not "as close as possible". The solver must satisfy every fact that is kept, whatever its priority; the implementation's own aims for a good-looking drawing are optimized only within the facts kept, after all descriptions of the language.
- **`"disabled"`**: the fact is removed before the ordering and is not given to the solver. This is not a failure: the object still exists, descriptions that depend on it are not affected, no warning is given, and a priority written later can turn it on again.
- **How it is written**:

  - PLACE, ORIENT, the four atomic constraints, ORIENT-RELATION, AS and ATTACH take the parameter `priority=`, and the priority goes with the fact. `priority` is the only standard parameter of AS and ATTACH; a parameter of a definition cannot have this name.
  - The PRIORITY statement changes a priority afterwards. It takes only objects with a handle of their own: constraints, usages, and sets with GROUP (on and off only, 11.5). A set with GROUP is taken as the group itself; any other set is taken member by member (5.2.1). Given an instance or another object, it fails (wrong kind): `#M1 @PLACE(…) => @PRIORITY(…)` is an error, because PLACE passes on the handle of M1; write the priority as a parameter of PLACE.
  - The record head `@RNL(priority=…)` gives a default to the statements of the record that take this parameter and do not write it, and to the `dx` and `dy` of TEXT (13.3). To AS and ATTACH it gives only `"disabled"`, so that a disabled record turns off its usages with their bodies, like its other statements. Under a head with `"high"`, `"normal"` or `"low"`, a usage has a priority only when its own statement or a PRIORITY statement gives it one, and otherwise keeps the priorities of its body, as when a rule makes it, so that a usage that the solver writes out (10.5) means what the generated one meant. A priority that a usage writes still overrides its body (below).
- **Order of overriding**, from low to high: the default of the record head; the statement's own parameter; PRIORITY statements (among several, the later wins; for those in packages, by the position of the USE-PACKAGE); the priority of an enclosing usage; PRIORITY statements written from outside a usage, through `::`, directly on one of its local objects (among several, the later wins). When none of these is given, the priority is `"normal"`.
- **The priority of a usage goes down**: the priority of a usage (its own parameter, or a PRIORITY statement on it) applies to all facts that it expands to, including those of nested usages, and overrides their own priorities; with nesting, the outermost usage that has a priority decides. To adjust only a part inside, write directly on that local object from outside, through `::` (9.4): this is more specific than the priority of the usage and overrides it: the more specific one wins. Such a statement written in the body of an enclosing definition is still subject to the priority of the enclosing usage. Every statement of a record can be changed directly, so the record head gives only a default.
  - In full: the priority of a usage governs every PRIORITY statement written in the body of its definition, which then has no effect. A PRIORITY written from outside through `::` directly on a local usage governs in the same way: that usage counts as the outermost one with a priority, and the statement decides for everything that the usage expands to. Among the statements that nothing governs, the one written on the innermost object wins; on the same object, one written from outside the body that makes the object wins over one written in that body; then the later one wins.
  - For example, with `%o1` disabled, `%o1::in @PRIORITY("high")` at the top and `%in::row @PRIORITY("low")` in the body of the definition of `%o1`, the facts of `row` are `"high"`: the statement in the body is governed by the priority of `%o1`, which the statement at the top overrides for `in`. If `%o1` had no priority, they would be `"low"`, written on the more inner object. To have `"low"` in both cases, write `%o1::in::row @PRIORITY("low")` from outside.
- **The output of the solver**: the solver writes its results as PLACE, ORIENT and so on, with `"high"`, in the last record of the level's own text, such as `@RNL(type="generated", priority="high") {…}`. For a subcircuit, this is inside its `.subckt … .ends`, even when that is in an included file; for the top level, it is in the file level of the main file (chapter 17). When the solver solves, the level is interpreted without the records with `type="generated"` (2.8), so old results do not constrain the next solve; after solving, it removes them and writes the new results at the end.
  - Between solves, a new statement of normal priority that conflicts with the results of the solver is discarded and takes effect only after the next solve. A viewer may suggest solving again, by the hash in the record head; such discards need not be reported one by one.
  - A drag in an editor is written as a PLACE with `"high"`, also in the level's own text, after the record of the solver. This fixes the position: every later solve takes it as a hard constraint. To release it, delete that statement.
- **What the solver writes.** The record of the solver must hold every position, orientation and route that the solver decided and that the text leaves free: by name or key for the objects that have one, by number for the others (5.9), with WIRE for the route of a wire (13.1), and as explicit statements for the objects that rules generate (10.5). A decision that is not written there is not a result: the object stays free, as if the level had not been solved. So a viewer that does not solve draws the level exactly as the solver drew it. The solver does not decide what it cannot

refer to, such as the position of a text without a key that a generation rule makes directly; the viewer places such a text (13.3).

- **The results of a solver are RNL.** When a solver passes its results to other tools in another form, such as the messages between a solver and a viewer, that form must hold only what a record of the solver can hold, so that every result can be saved as such a record.
- **The objectives of a solver** choose only among the layouts that satisfy every fact that is kept; a fact that is kept is hard, whatever its priority. The objectives are not part of the language: the kinds of objectives and their order differ between solvers, and this standard cannot define a common vocabulary for them. Solver options are part of the input that the hash covers (chapter 17), and keeping them in the file would help to reproduce a result; this version has no extension mechanism (16.4), so a solver keeps its own settings in files of its own.

14.9 Orientation relation: ORIENT-RELATION

```
#M1 #M2 @ORIENT-RELATION("MY");                  /* M2 is drawn as M1 flipped left-right */
{#M1 #M3} {#M2 #M4} @ORIENT-RELATION("MY");      /* M1 and M3 alike, M2 and M4 alike, the two
pairs mirror images */
#M5 {#M7 #M8} @ORIENT-RELATION("R0")             /* M5, M7 and M8 have the same orientation */
```

- `A B @ORIENT-RELATION(relation)`: two slots, `from` and `to`, each an object or a set. The relation is one of the eight orientation codes of ORIENT (11.2), given by position. The orientation of B is the orientation of A followed by the transform of the relation. The transform is taken in the coordinates of the drawing, so `"MY"` always flips left-right about a vertical line, whether or not A itself is rotated.
- When a side is a set, its members all have the same orientation, and the relation holds between the two sides. To give a whole set one orientation, take one member out, as in `#M5 {#M7 #M8} @ORIENT-RELATION("R0")`, or write the set twice: `{#M5 #M7 #M8} {#M5 #M7 #M8} @ORIENT-RELATION("R0")`.
- There is no form with a single set whose members all have the same orientation: the keyword names a relation between two things, and with one set it is not clear what the relation applies to. A single object does not match the signature, which is an error. A single set is first split one level, as 5.2.1 says, so `{#M1 #M2} @ORIENT-RELATION("MY")` is `#M1 #M2 @ORIENT-RELATION("MY")`; when the split does not give two objects, it is an error.
- Members are objects that have an orientation, as for ORIENT (11.2): instances, bundles and usages with a symbol. A terminal, point, net, wire or usage without a symbol is an error: the constraint fails. When one side is an empty set, the constraint holds trivially.
- It is a constraint: it makes a constraint object, can be named, takes `priority=` and can be changed with PRIORITY (14.8); in conflicts it is kept or discarded as one unit (8.5). A member with an ORIENT has its orientation fixed already; for the other members, the solver chooses among the orientations that meet the relation.
- It does not fix the actual orientation: when two MOS transistors are mirror images, gates facing inwards and gates facing outwards both meet `"MY"`. To decide, write one ORIENT as well, or an ORDER on terminals (14.3).

15 Electrical semantics

15.1 Applying semantic definitions

```
%dp1 = #M1(in_p) #M2(in_n) #M5(tail) @AS $DIFFERENTIAL_PAIR;
%cm1 = {#M3 #M9}(ref) {#M4 #M6}(out) @AS $CURRENT_MIRROR;
%in = {#M8 #M5} @AS $MATCHED
```

- What objects mean is marked by applying a semantic definition with AS (9.5): the application makes a usage, and the objects fill the roles of the role list of the definition. Marking a meaning and stating a layout relation are the same kind of statement.
- A usage can be a member of another usage (nesting), as in the examples of 9.4 and 10.1.
- A set role, written `"{name}"` in the role list (such as `"{members}"` of `$MATCHED`, `"{tail}"` of `$DIFFERENTIAL_PAIR`, and `"{ref}"` and `"{out}"` of `$CURRENT_MIRROR` in the example), takes one set, for a number of members that is not fixed (9.4).
- A role can declare the kind of object it takes, such as `roles=[["in_p", "instance"], ["in_n", "instance"]]`; an object of the wrong kind makes the whole application fail (9.4). A definition with an empty body checks nothing of what fills the roles, so without such declarations a wrong object cannot be detected there.
- The role names of a usage are its sub-objects, such as `%dp1::in_p`.
- Applying a definition expands its body, and the body may contain constraints, such as a default drawing at priority `"low"`, which the priority of the usage turns off (9.5, 14.8).
- Not checked: whether the meaning agrees with the connections, types or parameters of the netlist is not checked (Principles 2.4). For example, `#M3(ref) #M4(out) @AS $CURRENT_MIRROR` does not check whether the gate and the drain of M3 are connected.
- The usages of a given meaning are selected with SELECT-USAGE, and the usages that contain a given object with SELECT-ANCESTOR-USAGE; the members of usages are reached with a selection whose scope is the usages (10.1).
- The vocabulary of semantic definitions belongs to the standard package (9.5), which provides definitions such as `$DIFFERENTIAL_PAIR`.
- The basic validity of one description (Principles 2.4) is: its references can be found, its roles receive objects of the right kinds, its single roles are filled, and each statement it expands to is valid itself (9.5). Whether two roles are filled with the same object is not checked.

15.2 Current direction: CURRENT

```
#M5:1 @CURRENT("in");                /* drain: the current flows in */
#M5:3 @CURRENT("out")                /* source: the current flows out */
```

- CURRENT sets the direction of the static current at a terminal: `"in"` (it flows in), `"out"` (it flows out) or `"unspecified"` (used to override a value given by a rule of a package). The parameter is `direction`.
- It is a fact about the terminal. Like ORIENT, it is a modifying statement and passes on its input handle. It has a key, one per terminal, and the later one wins. A terminal for which it is never written is unspecified.
- It is given only on terminals. The current inside a device with several terminals cannot be split up clearly: for a three-terminal device, "a flows to b, b flows to c" leaves it unclear whether the current at b flows in or out, while the direction of the static current at a terminal always has a definite meaning.
- It takes only terminals of the host, including the ports of this level: `#:inp @CURRENT("in")` means that the current flows into the subcircuit there. Terminals of virtual parts do not exist electrically: given one, CURRENT fails. A terminal without a pin is allowed (unlike for wires).
- Not checked: neither against the netlist, nor whether the directions of the terminals agree with each other (Principles 2.4).
- Use: the current at the terminal runs in the direction in which a wire leaves the pin (9.2.4), turned with the orientation of the symbol, when it flows out, and the other way when it flows in. A solver can use this to choose

among the layouts, such as preferring the current to run downwards (14.8); up or down is a matter of house style. A terminal without CURRENT, with `"unspecified"`, or whose pin does not write `direction=` (9.2.4) gives no direction; a direction that an implementation infers from the graphics does not count here. Rules can select terminals by the direction of their current, with the condition `current` of SELECT-TERMINAL (10.1).

- **Arrows**: CURRENT draws nothing. Whether and how arrows are drawn from CURRENT is a preference of the viewer (part of its house style), like which net names are shown (13.3). A viewer that draws them itself skips terminals that already have a usage of the standard current arrow, as it recognizes the standard `$NO_CONNECT` (12.6). Arrows to be kept (from an imported schematic, or decided by the author) are written as explicit objects:

```
$CURRENT_ARROW @ATTACH #M5:1 => @ORIENT("R270") => @TEXT("I_{SS}", dx=1);   /* on the drain pin,
with I_SS beside it */
$CURRENT_ARROW @ATTACH (12, 20) => @ORIENT("R270")                           /* in the middle of
a wire, at absolute coordinates */
```

  - An arrow drawn on a terminal is a usage whose main object is the terminal (12.1). It does not turn with the pin, so its orientation is written; the text beside it is attached to the usage (13.3).
  - An arrow in the middle of a wire is not anchored to the wire: it is drawn at a point, with absolute coordinates.
  - How "in" and "out" are told apart, and how the definition and the symbol of the arrow are drawn, belong to the standard package. The symbol can be just a line with an arrowhead at its end (9.2.5).
  - Drawn arrows are kept apart from CURRENT: if they were one thing, exporting an arrow would change what the layout aims for.

16 Files, versions, packages and aliases

This chapter covers the language version of a file, packages and package files, the standard package, unknown content, and aliases.

16.1 Versions

```
* @RNL(type="header") { @VERSION("0.3")}
```

- **One language version per file.** RNL of different versions is written in different files, joined with include (6.1).
- **Declared in the header record.** VERSION may be written only in the header record (`type="header"`), which is the first record of the file (2.8). The version may be declared only once, so a file never has two version values. A VERSION written anywhere else is an error and is ignored.
- **Order of reading:**
  1. the host adapter gives the comments;
  2. all records are cut out by their outer form, and their record heads are read; this step does not depend on the version;
  3. when the first record is the header record, its VERSION is read;
  4. the other records are parsed in that version.
- **Parts that do not depend on the version.** The comment carriers, the `@RNL` entry, the form of the record head, the matching of braces, and the form of VERSION in the header record do not change in any later version; otherwise a parser could not find the version. Support for a new host only adds carriers and does not change existing ones (2.2). The rest of the header record is read in the declared version.
- **No header record.** The header record may be left out, and so may VERSION in it. An included file then takes the version of the file that includes it; a main file takes the default version of the tool, with a hint. Tools must write the header record and the version when they save a file, so hand-written files may leave them out and files written by tools have them. A package file must have a header record (16.2); without VERSION it takes the default version of the tool, as a main file does. An inline package has the version of the file it is written in (16.2).
- **Unsupported version.** When a tool does not support the version of a file, none of the RNL of that file is interpreted, an error is reported, and the file is kept as it is when saved; other files are interpreted as usual.
- Implementations must declare the versions they support (Principles 2.3). The language version, the version of a package (`version=` of USE-PACKAGE) and the version of the host are three different things. How package versions are written and matched is in 16.2. A package file is read in the language version it declares itself, whatever the version of the file that uses it.
- **Version numbers.** A version is written `"major.minor"`: two unsigned decimal integers, such as `"1.0"` or `"1.2"`. There are no patch numbers, dates or profiles. Versions before `"1.0"` are drafts, written `"0.minor"`; versions 0.x promise no compatibility with each other, and each counts as a major version of its own. A malformed version, such as `"1"` or `"v1.2"`, is handled as an unsupported version, not as a missing one, so that the file is not misread in another version.
- **A file is read in the version it declares.** Tools must know in which version each standard word was added: in a file that declares an earlier version, a word added later is not a standard word. So a standard sub-object name added later does not affect a key of the same spelling in an older file (6.5), a new attribute name does not affect a host parameter of the same name (13.6), and a new standard parameter of AS and ATTACH does not affect a definition parameter of the same name (9.4).
- **Within one major version, a new minor version only adds and does not change meaning.**
  - Allowed: adding keywords, parameters, values, standard words, sub-object names and attribute names; giving a meaning to a form that used to fail. The items of Annex C, and whatever this standard calls a pure relaxation, are of this kind, and so is making the cases where the whole interpretation fails (1.5 item 3) tolerant.
  - Not allowed without a new major version: changing the meaning of a valid form; removing anything; changing a default value; turning a valid form into an error.
  - An old file keeps its meaning in a newer tool. In an older tool, the parts of a newer file that the older version has are read as usual, the new parts are unknown content (16.4), and a warning says that the version of the

file is newer than the tool supports. The tool may also choose not to draw the file, but it must not pretend to understand it.

- **A different major version.** A file of a major version that the tool does not support is not interpreted and is kept as it is (see above). The standard package, used implicitly at the root, is not applied either (16.3): no RNL of the file is interpreted, and a viewer draws the netlist with its own default appearance and reports that the version is not supported. A tool may support several major versions. Moving a file to a new major version is an explicit migration, in which the tool rewrites the file and its header record; a tool must not silently read an old file by the rules of a new major version. Each new major version of this standard comes with migration notes.
- **Supported versions.** A tool states, for each major version, up to which minor version it supports, such as `"1.0"` – `"1.3"`. Supporting `"1.3"` means being able to read `"1.0"` to `"1.2"` as well.
- **Raising the version.** When a tool writes into a file a form that only a newer version has, it raises the version in the header record. Before that, it checks whether the standard words added in between have the spelling of names already in the file (keys of texts, role names, definition parameters, host parameters) and gives a warning for each such name, as raising the version can change its meaning; the version is raised all the same.
- **Version 0.2.** This version replaces version 0.2. A tool may still read files of version 0.2, by the rules of 0.2, but writes only 0.3: when it writes into a file of 0.2, it raises it to 0.3 and writes `@USE-PACKAGE(name="rnl-std", version="0.2")` in its header record, unless the file or a file that includes it pins the standard package already, so that the file keeps its names and its drawing (16.3).

16.2 Packages and USE-PACKAGE

```
@USE-PACKAGE(name="acme-style", version="1.2");                          /* looked up in the
package directories of the tool */
@USE-PACKAGE(name="acme-style", version="1.2", path="lib/acme-style.rnl");   /* reads this
package file */
@USE-PACKAGE(name="acme-style", version="1.2") {…}                       /* an inline package
*/
```

Where a package takes effect.

- The `$` definitions and the rules of a package take effect in the level where its USE-PACKAGE is written and in the levels below. This version has no USE-PACKAGE that applies to its own level only. Written in a subcircuit, a package applies to that subcircuit only; written at a file level, to the subcircuits of the file and to the files it includes (6.1); written in the header record, to the same levels as at the file level, but placed at the root (16.3).
- The rules of a package apply in every level that uses the package (10.4).
- **Place in the order.** The content of a package is placed at the position of its USE-PACKAGE, and within the package in the order of its text, just as the facts expanded from a definition take the position of the AS or ATTACH statement. Conflict resolution (8.5) and the appearance queues (8.3) use this position. A package used before the statements of a level acts as a set of defaults; used after them, its rules can override earlier statements. A USE-PACKAGE in the header record is placed at the root (16.3).

Names in a package. Only the place follows the USE-PACKAGE; the names of a package stay its own.

- A name in a package is looked up first among the definitions of the package itself, then in the packages it references (see "Packages that reference packages" below), then in the standard package in effect at the level of the USE-PACKAGE. It is not looked up among the user's definitions of that level or of the levels above, nor in other packages; as in a definition body (6.9), a definition of the same name in the level of the USE-PACKAGE does not capture it. So an inline package means the same as a package file with the same content and can be moved out without change, and a package file can be checked on its own, with the default standard package of the tool. The standard package and the packages it references, directly or not, see only themselves and the packages they reference, not the standard package, so that a lookup never comes back to them.
- The `$` definitions of a package are public, and they count as belonging to the level just above the level of the USE-PACKAGE: a definition of the same name in that level hides the one of the package and gives a hint (6.8); it is not a second binding in one level. When the `$` definitions of a package hide definitions of the levels above, or of the packages at the root, a hint is given too (6.8).
- The `%` definitions of a package are private to the package.

What a package contains.

- A package holds definitions, `$` sets and rules. The header record of a package file holds VERSION and PACKAGE, and may also hold USE-PACKAGE (references to other packages, see below), DEVICE (6.12), and ALIAS and USE-ALIASES, which apply to that file only (16.5). An alias package is a package file with only a header record, loaded with USE-ALIASES (16.5).
- A package has no circuit of its own, like the file level of an included file (6.1):
  - statements that name particular host objects (`#M1`, `&vss`, `#:inp`) fail; a lone `#` is still the level in which a rule is evaluated;
  - a statement that is not a rule and draws a single scene object, such as `(0, 0) @RECTANGLE …`, has no level to be drawn in and is an error;
  - `scope=` and `type="generated"` in a record head are errors (2.8);
  - USE-PACKAGE in a record of a package other than its header record is an error: references to other packages are written in the header record only.
- The body of an inline package follows the same rules but has no header record: the name and version of the package are the parameters of its USE-PACKAGE, and DEVICE is written in the header record of the file itself. An inline package cannot reference other packages; to freeze a package collection into a file, its expanded content is written into the body in order (16.3).

Package files. How a package file is read is given in 2.7, with an example. Its header record declares with `@PACKAGE(name=…, version=…)` which package and which version the file is; a file without PACKAGE is not a package, and a USE-PACKAGE of it is an error. One package is one file: a package split over several files would need an order among them, which this version does not provide. A package file is read in the language version it declares itself (16.1).

Lookup.

- A USE-PACKAGE with a body is an inline package and is not looked up.
- With `path=`, that file is read. The path is relative to the directory of the file in which the USE-PACKAGE is written, with `/` as the separator; it names a local file, not a URL. When the file does not exist, or its PACKAGE does not match, the package is looked up by name, with a warning.
- Otherwise the package is looked up by the name and version of its PACKAGE in the package directories set in the tool. The standard package comes with the tool or is installed in a package directory. The package directories are a setting of the tool and are not written in the file; when there are several copies of the same version, the order of the directories in that setting decides.
- Diagnostics say which file and which version were used.
- When the package is not found, the USE-PACKAGE fails (an error), and the statements that use names from the package fail with it (7.3); the diagnostic gives the missing package as the cause.
- `name=` is required. It is compared as a whole string and is case-sensitive; package names should use lower-case letters, digits and `-`.
- With `path=`, a netlist and its package files can be kept together and moved as a whole; this version has no manifest format for them.

Package versions.

- A package version is written like a language version, `"major.minor"`; each 0.x counts as a major version of its own (16.1).
- When the version written exists, it is used. Otherwise the smallest of the newer versions within the same major version is used, with a warning that the version of the package differs and the drawing may differ; when the major version has only older versions, or none, the package is not found. An older version is not used, because it may lack names. A file named by `path=` is checked by the same rule.
- Without `version=`, the newest version found is used, with a hint. The `version=` of an inline package only identifies the version and is not looked up.
- A package found by name and version is read only once in a document: the same version is the same set of definitions wherever it is used. Each inline package is a copy of its own. Definitions of the same name in different versions are two definitions (9.5); see the example in 16.3.

- Package authors should follow the rule of the language (16.1): within one major version, do not remove or change existing names, roles, parameters or pins; the drawing may change. This is a convention that tools cannot check.
- This version has no content digest for packages (such as a `digest=` parameter of USE-PACKAGE). To keep a drawing exactly the same, pin the version, or write the package inline in the file. How the `hash` of a record head is computed is up to the tool (chapter 17).

Packages that reference packages. A package can reference other packages, so that a large package can be split into parts and put together again in several sizes; the standard package is built this way (16.3). A package whose file has only a header record and only references other packages is called a *package collection*; this standard usually just calls it a package.

- **Form.** In the header record of a package file, write `@USE-PACKAGE(name=…, version=…)` for each package referenced. A `path=` is relative to the directory of that package file; lookup and versions are as above. A reference has no body; one with a body is an error, because inline content has no version and cannot take part in choosing the newest version (below).
- **Order.** When a package is used, the packages its header record references are expanded first, in the order of the references, and placed before it; a referenced package that references others is expanded the same way first. Then comes the content of the package itself. Within one expansion each package is placed once, at its first place. So the content of a package collection is the content of the packages it references, one after another.
- **Names.** The `$` names of a referenced package are visible in the package that references it, and are passed on to its users; like the `$` names of the package itself, they count as belonging to the level just above the level of the USE-PACKAGE. When several referenced packages define `$X`, the later reference hides the earlier one, and a definition of the package itself hides them all; each such hiding gives a hint, as for several packages used in one level. This version does not distinguish references for the package's own use from references passed on to its users. `%` names stay private to each package.
- **Only one in effect.** Packages are counted by name, as in "Which USE-PACKAGE takes effect" below, including those brought in through other packages, at the place where they are expanded.
  - For a package that a file uses directly, the USE-PACKAGE written in a file that is nearest to the level takes effect, as in "Which USE-PACKAGE takes effect" below, with its version and inline content, also where other packages reference the package. When its major version differs from the one that a package referencing it asks for, a warning is given: the referencing package may use names it does not have.
  - For a package that is brought in only through other packages, the newest version within one major version among all the references on one chain of levels is used, with a hint when the versions differ; within one major version a newer version only adds, so it serves the older references too. References of different major versions are an error: the reference expanded later fails, and the statements that use names from it fail with it (7.3).
- **Replacing or dropping one of them** follows from "only one in effect"; there is no further rule. An empty inline package of the same name drops it, such as `@USE-PACKAGE(name="rnl-std-fill") {}`, which turns off the automatic fill; another version or other inline content under the same name replaces it. When a dropped package is used by other packages, the statements in them that use its names fail.
- **Cycles.** References that form a cycle are an error, and the reference that closes the cycle fails. Which reference that is depends on the place where the expansion starts; when several packages of a cycle are used directly, the cycle is reported once for each of them.
- **Where diagnostics are reported.** The hint about different versions and the warning about a different major version are reported on the USE-PACKAGE written in the file (hints and warnings inside packages are not shown to the user, 7.6). The error of a failing reference is reported on that reference and says which USE-PACKAGE in the file brought it in.
- An alias package cannot be referenced (16.5). USE-ALIASES in the header record of a package file applies to that file only, as usual.

The same `$` name in several packages.

- When several packages used in one level define `$X`, the one used later hides the earlier one, with a hint.
- A name is looked up in this order: this level → the packages used in this level (the later first) → the level above → the packages used in the level above → … → the packages at the root (the one placed later first).

- This version has no package-qualified names, other names for packages or level tags (such as `$acme.DIFFERENTIAL_PAIR` or `as=`). Note: a later version that adds them needs another form than a `.` after the name, which is common at the end of a sentence (`align #M1 and #M2.`).

Which USE-PACKAGE takes effect. On the chain of levels from the root down to a level, a package is in effect only once. The USE-PACKAGE written in the level nearest to this level takes effect, with its version and inline content; within one level the later one counts as nearer, and one in the header record counts as written at the very start of the file level. A package brought in through other packages counts as written at the USE-PACKAGE that brings it in, with its version chosen as in "Packages that reference packages" above. The package is placed at the position of that USE-PACKAGE, or at the root when that USE-PACKAGE is in the header record.

Note: otherwise the generating rules of a package, such as the port fill, would run once more in a level below and generate their objects twice.

Using a package again. A subcircuit or a library file may use the standard package again, at its file level or in a subcircuit. The appearance rules of the standard package are then placed after the rules that flow down from the levels above (such as those of the main file) and override them again, and the `$` names of the package, counted as belonging to the level just above that USE-PACKAGE, hide the definitions of the same name from the levels above. The symbols used in this level are then not affected by the appearance rules of the levels above (6.1).

- This affects only symbols and names. The geometric and generating rules that flow down from the levels above still apply; when a solution becomes stale because of them, a hint to solve again is given (chapter 17).
- To pin only the version, and still let the style of the levels above override the package, write the USE-PACKAGE in the header record (16.3).

Inline packages. USE-PACKAGE may have a body, which is the content of the package. The file then carries the package, and nothing is looked up:

```
@USE-PACKAGE(name="acme-style", version="1.2") {
  $NMOS_ACME = @DEFINE-SYMBOL {$NMOS; (-1, 0) @CIRCLE(radius=2.5)};
  # @SELECT-INSTANCE(type="nmos") => @SET-SYMBOL $NMOS_ACME
}
```

- An inline package means the same as a USE-PACKAGE without a body: the name and version of the package, its place at the USE-PACKAGE, its names looked up as in "Names in a package", its `$` names counted as the level above, its `%` names private, and "only one in effect" counted by package name all hold as above. The body is consumed by enclosing (5.4). An inline package has the language version of the file it is written in (16.1).
- An inline package differs from a package file only in that its content is written in the netlist and that it has no header record: its language version is that of the file, its `version=` only identifies the version, it cannot declare DEVICE or ALIAS, and it cannot reference other packages. Each inline package is a copy of its own.
- Inline packages are normally used for two things only: to drop a package, with an empty inline package of the same name (16.3), and to freeze a whole set of packages into the file, with a body that holds their full expanded content and is complete in itself.
- To extend the standard package, or to replace one part of it, write a package file whose header record references the packages it uses (see "Packages that reference packages" above), and point to it from the netlist with `path=`. An inline package with content that replaces a sub-package of the standard package counts as part of the standard package and sees only itself: it cannot use names from the other sub-packages of the standard package.

Other rules.

- Processing a package must not execute code or access the network implicitly (Principles 2.7).
- How a viewer displays a drawing, and whether it can change the appearance by the preferences of the user (such as a dark background or printing), is outside this standard. A property that no style or parameter gives a value takes the default of the renderer (9.3, Principles 2.7).

16.3 The standard package

- The standard package is not part of this standard. It is written in RNL like any other package and published separately; its semantic definitions, virtual parts and other content are defined by the package itself.

- The names in the standard package have no mandatory prefix, such as `$DIFFERENTIAL_PAIR`. A user may write a definition of the same name, which replaces the standard one by hiding it (6.8).
- **A package collection.** The standard package `rnl-std` has only a header record. It references the smallest collection, `rnl-std-core`, and then two sub-packages:
  - `rnl-std-core` references `rnl-std-style` (styles and the default style rules), `rnl-std-basic` and `rnl-std-semiconductors` (the symbols and the definitions, including the virtual parts, such as grounds, supplies and port tags), and the rule packages of the two dialects, which give instances their symbols: `rnl-std-spice` by element letter and model type, `rnl-std-spectre` by master and model type (see the dialects below). It only draws the netlist and adds nothing.
  - `rnl-std-fill` holds the sets for the fill and the rules that add grounds, supplies and port tags; it references `rnl-std-basic`.
  - `rnl-std-structures` holds the definitions of circuit structures, such as the differential pair and the current mirror, and of layouts, such as `$MIRRORED` and `$FLOW`; the constraints in their bodies are all of priority `"low"` (9.5, 14.8).
  - The names in the sub-packages do not repeat, so a user need not know which sub-package a name comes from.
  - The fuller collection `rnl-std-full` references `rnl-std` and the packages of particular fields: `rnl-std-functional` (amplifiers, comparators, converters and the like), `rnl-std-rf`, `rnl-std-logic` and `rnl-std-signal-flow`. The optional `rnl-std-no-labels` turns off the default labels of the standard symbols, for files that write all their labels themselves; it is in no collection. Further collections and packages may be added later. The whole standard series is released together under one version number, and a collection pins the versions of its sub-packages.
  - This is the structure of the standard series 0.3.
- **Implicit loading.** The standard package `rnl-std` is used at the root, also when no file writes it, except for a file whose version is not supported (16.1). Its version is the default of the tool for the language version of the file: the newest version that the tool has of the standard series written in that language version. So a file of version 0.3 uses the series 0.3, as does a main file without a header record that the tool reads as 0.3 (16.1); a hint may be given. The version used therefore changes when the tool is upgraded; a USE-PACKAGE in the header record of the file, or of a file that includes it, pins it, and the pinned version comes first (below).
  - Note: so a netlist without any RNL is drawn with standard symbols, and a writer can use the standard definitions without first writing a USE-PACKAGE. The standard package is a package, not code in the viewer, so viewers need no symbols of their own for instances.
- **The rule package of the other dialect.** The names of instances mean different things in the two dialects: in SPICE, `I0` is a current source; in Spectre, the name of an instance says nothing, and Virtuoso names the instance of a subcircuit `I0`. In a file of version 0.3, the rule package of the other dialect is therefore dropped implicitly: a file in the SPICE dialect does not use `rnl-std-spectre`, and a file in the Spectre dialect does not use `rnl-std-spice`. This is as if the header record of the file wrote an empty inline package of that name, but only for the levels of that file.
  - The dialect of a file is the language mode in which its first statement of the host is read (2.2): the first line of the file that is not a comment, not a blank line, not the title of the main file and not a `simulator lang=` line. A `.scs` file starts in the Spectre mode and any other file in the SPICE mode, and a `simulator lang=` line before the first statement changes the mode in which it is read (2.2). So in a Spectre netlist, a `.scs` file is in the Spectre dialect unless its first statement comes after `simulator lang=spice`, and another file is in the SPICE dialect unless its first statement comes after `simulator lang=spectre`; the files of a SPICE netlist are in the SPICE dialect. A `simulator lang=` line after the first statement does not change the dialect of the file.
  - Writing a header record into a file does not change its dialect, since comments are skipped. A tool writes the header record with a comment marker of the language mode where it writes it (2.2). When the first statement of a file comes after a `simulator lang=` line, the header record may go right after that line, in the mode of the rest of the file, or before it, such as at line 2, in the mode that the file starts in. In either place it must be the first record of the file (2.8).
  - Each file drops by its own dialect, also when another file includes it: a SPICE file included by a Spectre file uses `rnl-std-spice`, not `rnl-std-spectre`.

  - What is written comes first: a level does not drop a package implicitly when its chain of levels (16.2) holds a USE-PACKAGE of that package, in this file or in a file that includes it, whether of the package itself or of an inline package of its name; that USE-PACKAGE takes effect as 16.2 says. So a USE-PACKAGE written in a subcircuit keeps the package for that subcircuit and the levels below it, and the other levels of the file still drop it. A Spectre file that also wants the rules of SPICE writes `@USE-PACKAGE(name="rnl-std-spice", version="0.3")` in its header record.
  - When no standard package is used, or the standard series in use has no such packages, nothing is dropped. An inline package that freezes the standard package holds no packages of these names either (16.2), so nothing is dropped from it; a tool that freezes the standard package into a file of version 0.3 writes the content in effect for that file, which has the rules of its own dialect only.
- **Packages at the root.** The header record of a main file or of an included file may hold USE-PACKAGE:

```
* @RNL(type="header") { @VERSION("0.3"); @USE-PACKAGE(name="rnl-std", version="0.2")}
```

  - Its range is that of a USE-PACKAGE at the file level: the file that writes it and the files it includes. One written in an included file applies to that file and the files it includes in turn, not to the file that includes it (6.1).
  - It is placed at the root: before the rules of all levels, and also before block symbols (8.3, group 1). Its `$` names are at the root too, so a definition of the same name in a level, such as in the main file, still hides it. A USE-PACKAGE at the file level is placed at its own position instead: it uses the package again (16.2), so it overrides the rules that flow down, and its `$` names count as belonging to the level above. Written in the header record, a package says "this file depends on this version, and the style of the levels above may still override it"; written at the file level, it says "this file uses the package to set its own drawing".
  - Which USE-PACKAGE takes effect is as in 16.2: the one written in the level nearest to this level, where one in the header record counts as written at the very start of the file level.
  - The order of the packages at the root: the standard package first, then the others in the order of the first file that writes each (from the outermost file inwards) and of its header record. When the header record of an inner file writes the same package again, this only changes the version or the inline content used in the levels of that file; the place stays, so pinning a version does not change the order.
  - A package in a header record that defines a `$` name of the standard package, or of a package placed at the root before it, hides that definition and gives a hint (6.8).
  - Pinning a version is the author's choice. A main file should pin it. An included file that does not pin it follows the file that includes it; it should pin a version only when it depends on one. Tools must not add versions to included files by themselves, except for the pin of the series 0.2 that raising a file of 0.2 writes (16.1).
  - A version should not be pinned with a USE-PACKAGE at the file level of the main file: there the package is placed after the block symbols, and when the file is included elsewhere, the USE-PACKAGE uses the package again (16.2) and changes the meaning of the file.
- **Example.** The header record of `tb.sp` writes `@USE-PACKAGE(name="rnl-std", version="1.2")`, and that of `ota.sp`, which `tb.sp` includes, writes version `"1.0"`.
  - The top level of `tb.sp` and the subcircuits defined in `tb.sp` use `"1.2"`; the subcircuits in `ota.sp` use `"1.0"`. The style rules of `tb.sp` still flow into `ota.sp` and override the defaults of `"1.0"`.
  - Default choices are made by rules in the standard package, such as the rule that gives the usages of `$GROUND` a ground symbol. Each version selects its own `$GROUND` in the levels that use it, so `ota.sp` is drawn as usual.
  - Definitions of the same name in the two versions are two definitions (16.2). So a rule of `tb.sp` that selects by definition, such as `# @SELECT-USAGE $GROUND => @SET-SYMBOL $MY_GND`, finds only the usages of the `$GROUND` of `"1.2"`; `ota.sp` follows it only if it does not pin a version of its own. Rules that select by properties of the host, such as `# @SELECT-INSTANCE(type="nmos") => …`, are not affected.
  - For its symbols to be free from the rules of `tb.sp`, `ota.sp` uses the package at its file level (16.2).
- **More or less of the standard package.** Only the package name `rnl-std` is loaded implicitly.
  - For more, write a fuller collection or single sub-packages in the header record. A sub-package of the same name is in effect only once, never twice.

- For less, drop the unwanted sub-packages with empty inline packages of the same name: `@USE-PACKAGE(name="rnl-std-fill") {}` turns off the automatic fill. For the smallest collection only, drop the default with an empty `rnl-std` and use `rnl-std-core`:

```
* @RNL(type="header") {
*   @VERSION("0.3");
*   @USE-PACKAGE(name="rnl-std") {};
*   @USE-PACKAGE(name="rnl-std-core", version="0.3")
* }
```

- **Extending the standard package or replacing a part of it.** Write a package file whose header record references the standard sub-packages it uses, and add its own content (16.2, inline packages). Definitions and rules that only add to the standard package may also be written directly in the netlist; rules that the standard package has can be removed or replaced only by a package of the same name.
- **Freezing or dropping the whole standard package.** Write an inline package of the same name in the header record. This follows from "only one in effect", counted by package name; there is no further rule:
  - `@USE-PACKAGE(name="rnl-std", version="1.2") {…}` freezes the standard package into the file, making the file self-contained; the body is the content of the sub-packages after expansion, one after another in the order of the expansion (16.2);
  - `@USE-PACKAGE(name="rnl-std") {}` uses no standard package; the appearance is then left to the defaults of the renderer (Principles 2.7).
- Conventions such as bold buses, tap symbols for buses and marks for the width of a bus are content of the standard package too. Without a standard package, the renderer decides the default symbols and drawing style (Principles 2.7).

16.4 Extensions and unknown content

- **Unknown content** (Principles 2.3). Unknown keywords, parameters and values are errors: the statement fails, the statements that depend on it fail with it (5.5, 7.3), and the rest is interpreted as usual (7.6). Unknown fields of a record head are the exception and give only a warning (2.8). The drawing may then be incomplete, and the diagnostics must say so.
- **Kept when saving.** Principles 2.3 recommends keeping such content unchanged; this standard requires it:
  - records that are not the tool's own are not changed (chapter 17), and the host text and ordinary comments are left as they are;
  - when a tool rewrites its own records, the statements in them that it does not know or that fail are kept as they are, with their free text, in the same order;
  - a file whose version is not supported is kept entirely as it is (16.1);
  - the records that a tool rewrites need not stay the same byte for byte.
- **No extension mechanism in this version.** New semantics, virtual parts and ways of drawing are written as definitions, symbols and styles in packages (1.5 item 7), not as new keywords; a new mechanism that implementations must understand waits for a new version of this standard. Data private to a tool, such as the locks or the view state of an editor, is not written in RNL records; it goes in files of the tool, or in ordinary comments outside records.
  - Note: this version first makes all tools read the same RNL; vendor dialects this early would split the language, and new semantics can already be written in packages.
  - Note: a later version may add extensions as keywords with a vendor prefix and a `.`, such as `@ACME.ROUTE`, declared in the header record: a required extension that a tool does not know would leave the whole file uninterpreted, and an optional one would be skipped with a hint. Such a keyword is an unknown keyword now, an error, so adding this later is a pure relaxation (16.1); a vendor prefix would not be allowed to have the name of a standard keyword.
- No form is reserved for extensions: a `.` right after a standard keyword is free text, so `@BUNDLE.LOCK` is BUNDLE followed by the free text `.LOCK`, and `{#M1 #M2} @GROUP.` at the end of a sentence is valid (3.5).
- **Recognized but not interpreted.** In this version the only content of this kind is an unknown field of a record head (2.8). Optional extensions would be a second kind, if a later version adds extensions.

16.5 Aliases

```
* @RNL(type="header") {
*   @VERSION("0.3");
*   @USE-ALIASES(name="rnl-zh");
*   @ALIAS("@at", "@PLACE")
* }
* @RNL {#M1 @位于(10, 20); {#M1 #M2} @对齐("y"); #M3 @at(x=4); #M1(正输入) #M2(负输入) @视作 $"差分对"}
```

- Aliases give standard words other spellings, for compatibility with another use of upper and lower case, or for writing RNL in the style of another natural language, such as Chinese.
- **Where aliases come from:**
  - ALIAS statements written directly in the header record, for a few aliases of one's own;
  - alias packages, loaded with `@USE-ALIASES(name=…, version=…, path=…)` in the header record. The aliases of one language are made into one package, so that files need not repeat them and the wording stays the same everywhere. The standard Chinese alias package is `rnl-zh`; it is published with the standard package and comes with the tools (16.3).
  - Both may be used together, and several alias packages may be loaded; together they are checked by the rules of correspondence below.
  - Aliases apply only to the file that declares them and are not inherited through include; an included file or a package file that uses aliases declares them in its own header record. The same standard word may have different aliases in different files. The keys of the texts written in a symbol are checked against the aliases of the file in which the symbol is written (6.5, 13.3).
  - The header record itself is always read in the standard spelling and does not recognize aliases: until USE-ALIASES and ALIAS have been read, the aliases are not known, and header records are mostly written by tools. So aliases of the keywords that may be written only in the header record (VERSION, ALIAS, USE-ALIASES, DEVICE, PACKAGE) are of no use.
- **Alias packages:**
  - An alias package is a package file (16.2) whose header record holds only VERSION, PACKAGE and a list of ALIAS statements, and which has no other records. It is looked up, and its versions are chosen, as in 16.2.
  - USE-ALIASES of a file that is not an alias package, or USE-PACKAGE of an alias package, is an error. When an alias package is not found, its USE-ALIASES is an error, and the statements that use its aliases fail with it (7.3); the diagnostic gives the missing alias package as the cause.
  - Note: alias packages are not loaded with USE-PACKAGE because USE-PACKAGE may be written in any level, deals with definitions and rules, and is ordered by position, while aliases deal with how the text of a file is read and can only apply per file. Kept apart, a house-style package cannot change how the text is read.
  - The content of an alias package follows what it refers to: when a name in the standard package changes, `rnl-zh` changes with it and its version is raised.
- **Form.** `@ALIAS("alias", "original")`: the alias first. The two parameters are `alias` and `original`, normally given by position; the third parameter, `definition`, is given by name only (below).
- **Originals:**
  - A keyword or standard word defined by this standard, in its standard spelling. A keyword is written with its `@`, such as `"@PLACE"`; other words without it, such as `"axis"` or `"low"`. The alias is written the same way: the alias of a keyword has `@`, other aliases do not. The `@` tells a keyword from a standard word of the same spelling, and an alias appears in the text exactly as declared.
  - A `$` name. Both the alias and the original have the `$`, such as `@ALIAS("$接地补位集", "$GROUND_FILL_SET")`; in the text it is written `$"接地补位集"` (a non-ASCII name after `$` needs quotes, 3.6) and read as `$GROUND_FILL_SET`. `%` names have no aliases: writers can give their own names in another language directly, and the `%` names in packages are private.
  - A name in a definition, declared with `definition=`, such as `@ALIAS("正输入", "in_p", definition="$DIFFERENTIAL_PAIR")` or `@ALIAS("间距", "gap", definition="$GROUND_STUB")`. Such an alias is looked up only within that definition: role names when it is applied (`#M1(正输入) #M2(负输入) @AS $DIFFERENTIAL_PAIR`), names of parameters passed to it by AS and ATTACH (`$GROUND_STUB @ATTACH(间距=3) …`), and local names taken from its usages through `::`

(`%c1::"中点"`). The value of `definition` may itself be written as an alias. Aliases of standard words are not used in these places, except for `priority`, the parameter of AS and ATTACH themselves.
  - The aliases go by the name: they follow the `$` name, and also apply where a lower level hides the definition with one of the same name, just as an alias of a `$` name replaces the spelling.
  - When the alias is declared, the original is not checked against the definition; when the definition is applied, a name that is not found is an error as usual (such as "no role in_p").
  - Names after `::` are looked up in the order of 6.5: the names of the usage itself, their `definition=` aliases, the standard sub-object names and their aliases, the keys of texts.
  - The value of the selection condition `role=`, and a path in `${…}` that goes down along roles, also recognize these aliases, looked up in the definition of the usage at each step, such as `%c1 @SELECT-INSTANCE(role="第一级")` or `${第一级::正输入::w}`, for a usage `%c1` of `$CASCADE` whose first stage is a differential pair. When the scope holds usages of several definitions, each is looked up in its own definition; the originals still work.
- **What can have aliases:** all the words of the word list except those listed in the next item: keywords, slot names of built-in keywords, parameter names and values, kind names, sub-object names, attribute names, style properties, `true` and `false` (5.3); and the `$` names and names in definitions above.
- **What cannot have aliases:** the outer form, that is, `@RNL`, the field names of a record head and the values of `type`, and `@VERSION`; and `@ALIAS` and `@USE-ALIASES` themselves.
- **Other names.** A name is already another name of an object: after `$"差分对" = $DIFFERENTIAL_PAIR`, `$"差分对"` is the same definition. Such a name is part of the text, cannot be taken out again, and does not reach role or parameter names; for a full set of names in another language, use aliases.
- **Correspondence:**
  - An original may have several aliases; within one scope an alias stands for one original only. The `definition=` aliases of each definition are one scope.
  - An alias must not have the spelling of any standard word of the version that the file declares, and must not be declared again for another original. A `definition=` alias must not have the spelling of a role, parameter or local name that the definition has, nor of an alias of `priority`, the parameter of AS and ATTACH themselves.
  - An alias cannot be used as an original, and cannot be the same as its original. When the version of a file is raised and a newly added standard word has the spelling of an alias, a warning is given and the version is raised all the same (16.1); in the new version the alias then has the spelling of a standard word, which the rule above makes an error.
  - An alias package is read in the language version it declares itself (16.1); an original that is not a standard word in the version of the file that uses the package is an error.
- **Aliases only add spellings.** The standard spellings always work, and both may be mixed in one file.
- **Aliases replace whole tokens only:**
  - an alias replaces a whole token; it is not a text macro, so positions and diagnostics stay right;
  - standard words without `@` get aliases by their spelling, so words of the same spelling (such as the parameter name `x` and the value `"x"`) share their aliases;
  - an alias of a `$` name replaces the spelling: a writer who writes `$"接地补位集" = {…}` defines `$GROUND_FILL_SET` in that level, which hides the one of the standard package with a hint (6.8).
- **Lexical rules:**
  - An alias of a keyword is the letters (of any script), digits and underscores right after `@`, without `-`; they form one token, and the `@` must be preceded by a space (2.3), also after Chinese punctuation: `对 &b，@选出元件` is an error, and must be written `对 &b， @选出元件`. When free text in Chinese follows, a space or a punctuation mark (`，`, `。`) separates it: `@对齐两个器件` is one unknown keyword, an error, and its diagnostic must ask whether a space is missing after `@对齐`. Tokens are not split by the declared aliases, so that reading the tokens needs no declarations.
  - Unquoted non-ASCII words occur only in two places: after `@`, and inside brackets. After `@`, the word must be a declared alias; otherwise it is an error ("a non-ASCII name needs quotes"). Inside brackets (parameter names, slot names of built-in keywords, `true` and `false`, role names), a declared alias is an alias; in the place of a role name or of a parameter name of a definition, a word that is not an alias is the name itself (3.6); in other places, a word that is not an alias is an error. Elsewhere 3.6 applies: host names, RNL names, terminal names and names

after `::` have only ASCII characters when they are not quoted, and an alias used after `::` is quoted too, such as `#M1::"位置"`.

- An unquoted alias in brackets that has the spelling of a parameter name or role name of a definition usable in the same place is an error, because its meaning is not unique. A quoted word in brackets is always a name, never an alias. Aliases of `$` names and aliases after `::` are written in quotes anyway (3.6).
- Aliases of enumeration values are written inside the strings and work only in parameters whose values are an enumeration. Attribute names in `${…}` recognize aliases too (13.6). Other strings, such as the `text` of TEXT, are not replaced.

- **Diagnostics and translation:**
  - Diagnostics show the words as written in the file, together with their standard spelling.
  - A translator works mechanically in both directions. Translating into aliases, it writes USE-ALIASES in the header record; for a self-contained file it writes the aliases used as ALIAS statements instead. When an original has several aliases, the first declared one is taken. Translating back into the standard spelling removes USE-ALIASES and ALIAS and loses nothing.
  - Free text, comments and strings are not translated, except enumeration values and attribute names in `${…}`.
  - LLMs, solvers and editors write the standard spelling. When they change a file written with aliases, the statements they add are also in the standard spelling; mixing is valid.

17 Modification model

- RNL describes only the current state, not the editing that led to it:
  - there are no undo or delete operations; deleting is editing the text;
  - there are no relative moves; an editor writes new absolute values.
- **Who changes which records.** The solver replaces the records with `type="generated"`. Other tools must change only the records whose `author` is themselves, such as an editor tidying up the changes it wrote, or a new import replacing what the previous import wrote. Tools must not touch authored records without an `author` (2.8).
- Descriptions appended later override earlier ones, in order (8.4); for geometric facts, priority is compared first (8.5). `type` and `author` do not affect the order (8.3).
- **Output goes with the subcircuit.** The output of the solver and the changes that a user makes in an editor are written into the level's own text: for a subcircuit, inside its `.subckt … .ends`, even when it is in an included file; for the top level, at the file level of the main file. The solver writes its output with `priority="high"` at the end of the level's own text; when it solves, the records with `type="generated"` are left out of the interpretation (2.8), and afterwards it replaces them (14.8). It must write back every decision it makes (14.8). `scope=` is only for statements that the author chooses to write outside (2.6). Whether a read-only file is saved is up to the tool.
- **A stale solution only gives a hint.** When an upper level or a package changes, the solution of a subcircuit in a library may become stale, for instance when the main file changes a symbol. The old solution still takes effect; a viewer that finds that the `hash` in the record head does not match the current input gives a hint to solve again (2.8).
  - The hash is computed from the actual input of the level after inheritance: the rules that flow down and take effect, the chosen symbols, the versions of the packages and the options of the solver, without the records with `type="generated"` and without the metadata fields. It is computed per level, not per file, and not from the subcircuit's own text only, or changes in the main file would go unnoticed.
  - The hash does not change the interpretation. Its algorithm belongs to the tools; the value should carry a tag for the algorithm, such as `"sig1:…"`, and a tool that does not know the algorithm does not compare it.
  - Note: if a hash that does not match meant that the old solution is not used, the algorithm would become part of the language, every tool would have to compute it the same way, and a viewer that does not solve could not draw the level.
  - Numbers are the exception: when the statements that make objects without a name change, the check of the numbers no longer matches, and the statements that use numbers take no effect until the level is solved again (5.9). The check covers only the text of the level and of the definitions it uses, so every tool computes it the same way, and the rest of the old solution is still drawn.
- **Generated objects.** When the solver has placed and wired objects that rules generate, it writes them as explicit statements into its own record, together with the wires. The explicit object takes precedence, and the fill rules no longer generate the same object (10.5). Objects in their bodies that have no local name are referred to by number (5.9).
- **Changing what the solver made.** A name that the solver gives to an object it writes out is bound only in its own record, which is left out when the level is solved (2.8). So a statement of another record that uses this name finds nothing when solving, and the solver cannot take it into account. To describe such an object on its own, an editor writes it by hand in its own record, as 10.5 says, and adds the change, such as `%g_r4 = $GROUND_STUB @ATTACH #R4:2; %g_r4::g @PLACE(30, -4, priority="high")`: the wire of the stub makes the terminal wired, the fill skips it, and the next solve no longer writes that object; until then, the object in the record of the solver is drawn as well. When the object changed is the generated usage itself, such as a port tag, a selection from the host object reaches it both when solving and when drawing, such as `&out @SELECT-ANCESTOR-USAGE $IO_PORT => @PLACE(40, 20, priority="high")`; the local objects of a usage cannot be selected (9.4).
- The fields of a record head apply to every statement in the record (2.8).

- **Example: a change in the editor.** Below is a complete fragment, with the comment prefixes of the host. After the header record, the designer wrote the first record and the solver the second; the user dragged VIN to the left in the editor, and the editor appended the third:

```
* Common-source stage
* @RNL(type="header") { @VERSION("0.3")}
.model nch nmos level=1
M1 out in 0 0 nch W=4u L=1u
RD vdd out 5k
VDD vdd 0 1.8
VIN in 0 DC 0.6 AC 1
* @RNL {
*   {#RD #M1} @ALIGN("x");
*   #M1 #RD @DISTANCE("bottom-to-top", min=2);
*   &vdd @ADD-TO $SUPPLY_NET_SET
* }
* @RNL(type="generated", priority="high", author="rnl-solver 0.3", hash="sig1:5be1c09a") {
*   #M1 @PLACE(10, 0); #RD @PLACE(10, 8); #VIN @PLACE(0, -2); #VDD @PLACE(20, 4)
* }
* @RNL(author="rnl-editor 0.1") { #VIN @PLACE(x=-4, priority="high")}
.end
```

  - The drag is written as a new record, appended after the record of the solver, with priority `"high"`. Both are high, so the later one is kept and the x that the solver wrote is discarded, with a hint (8.5); whenever the level is solved again, the new x is a fixed position (14.8).
  - Writing it into the designer's record does not work. The statements there are `"normal"` by default and cannot override the `"high"` of the solver; written as `"high"`, they still lose, because the record of the solver comes later, and within one priority the later fact is kept. This mistake is not an error; it only gives a hint about the resolution. The position takes effect the next time the level is solved, when the designer's statement is one of the constraints; until then the drawing does not change.
  - Parameters given by position come before those given by name (4.2 item 9): `@PLACE(-4, -2, priority="high")` is the same as `@PLACE(x=-4, y=-2, priority="high")`, and `@PLACE(x=-4, -2)` is an error.
  - To change what the designer wrote (change a symbol, rename, delete a constraint), edit the designer's record directly. The record of the solver is left alone, and is replaced as a whole the next time the level is solved.
- To disable a statement for a while, wrap it in `/* … */`.
- **Round trips.** RNL does not guarantee that a drawing stays the same after any export to RNL and import back (Principles 1); an exporter may choose to write exact RNL at the level of coordinates (Principles 3). This version does not define what exact RNL is, nor criteria for round trips.

Annex A Keyword signatures (normative)

This annex is generated from the word list `rnl-words-0.3.json` (5.3); where the two differ, it is an erratum. It lists what each keyword takes and gives; what it means is in the section named in the last column.

- **Slots** are listed in the standard order, with their names (which can be written as roles, 5.3) and what they take: *one*: a single object; given a set, the keyword is applied to each member (5.2.1), except that PRIORITY takes a set marked with GROUP as the group itself (14.8); *set*: a set; loose objects written for the slot form one set (5.2.1); *any*: one object, or a set taken as a whole; *shape*: a point, or a set of points (two forms of a graphic, 9.2.5); *body*: statements in braces, enclosed (5.4); *definition*: a definition: a `$` name, a `%` name bound to a definition, or an anonymous definition (5.6, 6.6). Absorbed input is strict; enclosed input is tolerant (5.4, 7.3).
- **Kinds** of slots are kind names (`instance`, `terminal`, `net`, `point`, `wire`, `usage`, `bundle`, `text`, `annotation`, `label`) or: *any*, any object; *level*, `#` alone: the level itself; *owner*, `?` alone in a symbol body: the object drawn with the symbol; *position*, the sub-object `::position` of an object (its origin); *locatable*, an object with a position: instance, bundle, terminal with a pin, point, text, the centre of a graphic written with a centre, a usage drawn at a point of its own, or a set of them (14.1); *orientable*, an object with an orientation: instance, bundle, usage with a symbol, or a set of them (14.9); *constraint*, an object made by a constraint keyword; *set*, a set, written as a name.
- **Parameters** marked *position 1*, *position 2* can be given by position, in that order; all others only by name (4.2). `priority` always takes `"disabled"`, `"low"`, `"normal"` or `"high"`; without it the priority of the record head applies, except that AS and ATTACH take only `"disabled"` from it (14.8).
- **Result**: the object the statement gives (5.5) and how its facts override (5.8). Without an override, facts accumulate.
- Unless noted, a keyword can be written in any record of a level; *symbol bodies* means it can also be written in a DEFINE-SYMBOL body.

A.1 The record

`@RNL(…) {…}` commits its statements to the level (5.4). Record head fields (2.8), all given by name:

Field	Values	Default	Affects interpretation
`type`	`"authored"`, `"generated"`, `"header"`	`"authored"`	yes
`scope`	string	—	yes
`priority`	`"disabled"`, `"low"`, `"normal"`, `"high"`	`"normal"`	yes
`author`	string	—	no
`hash`	string	—	no
`numbering`	string	—	yes
`modified`	string	—	no

A.2 Header records and packages

Keyword	Slots	Parameters	Result	Section
`@VERSION` (header records only)	—	`version` (position 1; required)	—	16.1
`@PACKAGE` (header record of a package file only)	—	`name` (required), `version` (required)	—	2.7
`@USE-PACKAGE` (also in header records)	`body`: body, optional	`name` (required), `version`, `path`	—	16.2
`@ALIAS` (header records only)	—	`alias` (position 1; required), `original` (position 2; required), `definition`	—	16.5
`@USE-ALIASES` (header records only)	—	`name` (required), `version`, `path`	—	16.5
`@DEVICE` (header records only)	—	`name` (required), `model`, `terminals` (required; array)	—	6.12

A.3 Definitions

Keyword	Slots	Parameters	Result	Section
`@DEFINE-SYMBOL` (also in symbol bodies)	`body` : body	`box` (array)	a definition (symbol)	9.2
`@DEFINE-STYLE`	`extends` : body (a style)	style properties (9.3)	a definition (style)	9.3
`@DEFINE`	`body` : body	`roles` (required; array), `parameters` (array)	a definition (general)	9.4

A.4 Symbol content and graphics

Keyword	Slots	Parameters	Result	Section
`@PIN` (symbol bodies only)	`position` : one (point)	`terminal` (required; terminal, or a non-empty array of terminals), `label`, `direction` (`"right"`, `"up"`, `"left"`, `"down"`)	a new pin	9.2.4
`@LINE` (also in symbol bodies)	`points` : set (point, terminal)	`arrow` (`"none"`, `"start"`, `"end"`, `"both"`; default `"none"`), `fill` (`"none"`, `"linecolor"`, `"fillcolor"`; default `"none"`), line style properties	a graphic of the symbol; in the scene, a new annotation (13.4)	9.2.5
`@RECTANGLE` (also in symbol bodies)	`points` : shape (point, terminal)	`width` (number), `height` (number), `fill` (`"none"`, `"linecolor"`, `"fillcolor"`; default `"none"`), area style properties; `width` and `height` together or not at all	a graphic of the symbol; in the scene, a new annotation (13.4)	9.2.5
`@CIRCLE` (also in symbol bodies)	`points` : shape (point, terminal)	`radius` (number), `fill` (`"none"`, `"linecolor"`, `"fillcolor"`; default `"none"`), area style properties	a graphic of the symbol; in the scene, a new annotation (13.4)	9.2.5
`@ELLIPSE` (also in symbol bodies)	`points` : shape (point, terminal)	`width` (number), `height` (number), `fill` (`"none"`, `"linecolor"`, `"fillcolor"`; default `"none"`), area style properties; `width` and `height` together or not at all	a graphic of the symbol; in the scene, a new annotation (13.4)	9.2.5
`@ARC` (also in symbol bodies)	`points` : set (point, terminal)	`arrow` (`"none"`, `"start"`, `"end"`, `"both"`; default `"none"`), `fill` (`"none"`, `"linecolor"`, `"fillcolor"`; default `"none"`), line style properties	a graphic of the symbol; in the scene, a new annotation (13.4)	9.2.5
`@BEZIER` (also in symbol bodies)	`points` : set (point, terminal)	`arrow` (`"none"`, `"start"`, `"end"`, `"both"`; default `"none"`), `fill` (`"none"`, `"linecolor"`, `"fillcolor"`; default `"none"`), line style properties	a graphic of the symbol; in the scene, a new annotation (13.4)	9.2.5
`@TEXT` (also in symbol bodies)	`attach` : one (point, instance, usage, bundle,	`text` (position 1; required), `key`, `dx` (number), `dy`	a new text; with `key`, a sub-object of the attached	13.3

Keyword	Slots	Parameters	Result	Section
	terminal, wire, annotation, net, `?`)	(number), `halign` (`"start"`, `"middle"`, `"end"`; default `"start"`), `valign` (`"baseline"`, `"top"`, `"middle"`, `"bottom"`; default `"baseline"`), `rotate` (`0`, `90`, `180`, `270`; default `0`), text style properties	object; overrides by key (key: attached object and key, when key is given)	

A.5 Placement and appearance

Keyword	Slots	Parameters	Result	Section
`@PLACE` (also in symbol bodies)	`target`: one (instance, usage, bundle, point, text, `::position`)	`x` (position 1; number), `y` (position 2; number), `priority`; at least one of `x`, `y`	the input object; geometric fact (key: object and axis)	11.1
`@ORIENT` (also in symbol bodies)	`target`: one (instance, usage, bundle)	`code` (position 1; required; `"R0"`, `"R90"`, `"R180"`, `"R270"`, `"MX"`, `"MY"`, `"MXR90"`, `"MYR90"`), `priority`	the input object; geometric fact (key: object)	11.2
`@LAYER`	`target`: one (instance, usage, bundle, wire, annotation, text)	`layer` (position 1; required; `"bottom"`, `"middle"`, `"top"`)	the input object; overrides by key (key: object)	11.6
`@SET-SYMBOL` (also in symbol bodies)	`target`: one (instance, usage, bundle, `#`); `symbol`: definition (a symbol)	—	the input object; appearance queue	11.3
`@SET-STYLE`	`target`: one (instance, usage, bundle, wire, annotation, text); `style`: definition (a style)	—	the input object; appearance queue	11.3

A.6 Selections

Keyword	Slots	Parameters	Result	Section
`@SELECT-INSTANCE`	`scope`: any (`#`, any object)	`name`, `model`, `type`, `role`, `recursive` (bool; default `true`)	a new set of instances	10.1
`@SELECT-TERMINAL`	`scope`: any (`#`, any object)	`name`, `model`, `type`, `terminal` (terminal), `net`, `wired` (bool), `current` (`"in"`, `"out"`, `"unspecified"`), `role`, `recursive` (bool; default `true`)	a new set of terminals	10.1
`@SELECT-NET`	`scope`: any (`#`, any object)	`name`, `port` (bool), `wired` (bool), `global` (bool), `role`, `recursive` (bool; default `true`)	a new set of nets	10.1
`@SELECT-WIRE`	`scope`: any (`#`, any object)	`net`, `bus` (bool), `role`, `recursive` (bool; default `true`)	a new set of wires	10.1
`@SELECT-LABEL`	`scope`: any (`#`, any object)	`key`, `role`, `recursive` (bool; default `true`)	a new set of labels	10.1
`@SELECT-ANNOTATION`	`scope`: any (`#`, any object)	`role`, `recursive` (bool; default `true`)	a new set of annotations	10.1

Keyword	Slots	Parameters	Result	Section
`@SELECT-USAGE`	`scope` : any (`#` , any object); `definition` : definition (a general definition), optional	`recursive` (bool; default `true`)	a new set of usages	10.1
`@SELECT-ANCESTOR-USAGE`	`scope` : any (any object); `definition` : definition (a general definition)	`recursive` (bool; default `true`)	a new set of usages	10.1

A.7 Applying definitions

Keyword	Slots	Parameters	Result	Section
`@AS` (also in symbol bodies)	the roles of the definition, then the definition (`definition` , written last)	the parameters of the definition (by position in the order of its parameter list, or by name; 9.4), `priority`	a new usage	9.4
`@ATTACH` (also in symbol bodies)	the definition, then its roles (`definition` , written first)	the parameters of the definition (by position in the order of its parameter list, or by name; 9.4), `priority`	a new usage	9.4

A.8 Wires

Keyword	Slots	Parameters	Result	Section
`@WIRE`	`net` : any (net); `points` : set (terminal, point)	`mode` (`"auto"` , `"straight"` , `"horizontal-vertical"` , `"vertical-horizontal"` , `"manhattan"` , `"manhattan45"` ; default `"auto"`), wire style properties	a new wire; given a wire, rewrites it, overriding by key (key: that wire)	13.1
`@ROUTE`	`net` : one (net)	`mode` (`"auto"` , `"straight"` , `"horizontal-vertical"` , `"vertical-horizontal"` , `"manhattan"` , `"manhattan45"` ; default `"auto"`), wire style properties	a new route	13.7

A.9 Sets, groups and bundles

Keyword	Slots	Parameters	Result	Section
`@BUNDLE`	`members` : set (instance)	—	a new bundle	11.8
`@GROUP`	`members` : set (any object)	—	the input object	11.5
`@ADD-TO`	`items` : any (any object); `target` : any (set)	—	the target set	5.2.2
`@REMOVE-FROM`	`items` : any (any object); `target` : any (set)	—	the target set	5.2.2
`@UNION`	`sets` : set (any object)	—	a new set	5.2.2

A.10 Constraints and priority

Keyword	Slots	Parameters	Result	Section
`@ALIGN`	`members` : set (locatable)	`axis` (position 1; required; `"x"` , `"y"`), `priority`	a new constraint; geometric fact	14.2
`@ORDER`	`members` : set (locatable)	`direction` (position 1; required; `"left-to-right"` , `"right-to-left"` , `"bottom-to-top"` ,	a new constraint; geometric fact	14.3

Keyword	Slots	Parameters	Result	Section
		`"top-to-bottom"`, `"x"`, `"y"`), `priority`		
`@DISTRIBUTE`	`members`: set (locatable)	`axis` (position 1; required; `"x"`, `"y"`), `priority`	a new constraint; geometric fact	14.4
`@DISTANCE`	`from`: any (locatable); `to`: any (locatable)	`direction` (position 1; required; `"left-to-right"`, `"right-to-left"`, `"bottom-to-top"`, `"top-to-bottom"`, `"x"`, `"y"`), `min` (number), `max` (number), `exact` (number), `priority`; at least one of `min`, `max`, `exact`; `exact` not with `min`; `exact` not with `max`	a new constraint; geometric fact	14.5
`@ORIENT-RELATION`	`from`: any (orientable); `to`: any (orientable)	`relation` (position 1; required; `"R0"`, `"R90"`, `"R180"`, `"R270"`, `"MX"`, `"MY"`, `"MXR90"`, `"MYR90"`), `priority`	a new constraint; geometric fact	14.9
`@PRIORITY`	`target`: one (constraint, usage, set)	`priority` (position 1; required)	the input object; overrides by key (key: object)	14.8

A.11 Electrical semantics

Keyword	Slots	Parameters	Result	Section
`@CURRENT`	`terminal`: one (terminal)	`direction` (position 1; required; `"in"`, `"out"`, `"unspecified"`)	the input object; overrides by key (key: terminal)	15.2

A.12 Other standard words

- Kinds: `instance`, `terminal`, `net`, `point`, `wire`, `usage`, `bundle`, `text`, `annotation`, `label` (5.3, 9.4).
- Sub-objects after `::`: `position`, `net` (6.5).
- Attributes read by `${…}`: `name`, `value`, `model`, `type`, `net`, `x`, `y` (13.6).
- Truth values: `true`, `false` (3.7).

Style properties (9.3):

Property	Type and values	Default
`linecolor`	color	—
`linestyle`	`"solid"`, `"dashed"`, `"dotted"`, `"dash-dot"`, `"double"`, `"bold"`, `"none"`	`"solid"`
`linewidth`	number	`0`
`cap`	`"butt"`, `"round"`, `"square"`	—
`join`	`"miter"`, `"round"`, `"bevel"`	—
`fillcolor`	color	—
`textcolor`	color	—
`size`	number	—
`font`	string	—
`bold`	bool	—
`italic`	bool	—
`arrowsize`	number	—

Style properties that keywords take as parameters (9.2.5): *line*: `linecolor`, `linestyle`, `linewidth`, `cap`, `join`, `fillcolor`, `arrowsize`; *area*: `linecolor`, `linestyle`, `linewidth`, `cap`, `join`, `fillcolor`; *wire*: `linecolor`, `linestyle`, `linewidth`, `cap`, `join`; *text*: `textcolor`, `size`, `font`, `bold`, `italic`.

Annex B Standard sentences (informative)

A standard sentence is a statement in the standard form with free text added between its tokens, so that the statement reads like a sentence. The free text does not change the meaning (3.3); the rules for the wording are in 4.5. In the tables, `…` at the start of a sentence stands for the segments before it in a pipe, and `{…}` or a final `…` for content that is left out.

B.1 Placement and orientation

Keyword	Case	Sentence
PLACE	two axes	`#M1 is at @PLACE(10, 20)`
	one axis	`the x of #M1 is at @PLACE(x=10)`
	written by an editor	`the x of #X2 is at @PLACE(x=3, priority="high")`
	in a pipe	`… => it is at @PLACE(40, 0)`
ORIENT	rotated	`#M2 @ORIENT("R90")`
	mirrored	`#M2 @ORIENT("MX"), i.e., flipped upside down`
		`#M2 @ORIENT("MY"), i.e., flipped left-right`
		`#M3 @ORIENT("MXR90"), i.e., flipped upside down, then rotated 90°`
		`#M3 @ORIENT("MYR90"), i.e., flipped left-right, then rotated 90°`
	in a pipe	`… => they @ORIENT("R90")`
ORIENT-RELATION	two objects	`from #M1 to #M2, they have @ORIENT-RELATION("MY"), i.e., flipped left-right`
	sets	`from {#M1 #M3} to {#M2 #M4}, they have @ORIENT-RELATION("MY"), i.e., flipped left-right`
	the same orientation	`from #M5 to {#M7 #M8}, they have @ORIENT-RELATION("R0"), i.e., the same orientation`
LAYER		`#X is on @LAYER("bottom")`
	in a pipe	`… => they are on @LAYER("top")`
GROUP		`{#M1 #M2 #M5} as a @GROUP`
	named	`%s1_set = {#M1 #M2 #M5} as a @GROUP`

B.2 Layout constraints

Keyword	Case	Sentence
ALIGN	same y	`{#M1 #M2} @ALIGN("y"), i.e., same y`
	same x	`{#M1 #C1} @ALIGN("x"), i.e., same x`
	low priority	`{#M7 #M5} @ALIGN("y", priority="low"), i.e., same y`
	in a pipe	`… => they @ALIGN("y")`
ORDER	with a direction	`{#M1 #M2 #M3} are in @ORDER("left-to-right")`
	either direction	`the x of {#M1 #M1:2 #M2:2 #M2} are in @ORDER("x"), either ascending or descending`
	in a pipe	`… => they are in @ORDER("bottom-to-top")`
DISTRIBUTE		`the x of {#R1 #R2 #R3} @DISTRIBUTE("x"), i.e., in equal steps`
	in a pipe	`… => the x of them @DISTRIBUTE("x"), i.e., in equal steps`
DISTANCE	with a direction	`from #M1 to #M2, they have @DISTANCE("bottom-to-top", min=4)`
	from a set	`from {#M6 #M7 #CL} to %pout, they have @DISTANCE("left-to-right", min=2)`
	either direction	`between %a_set and %b_set, they have @DISTANCE("x", min=1)`
PRIORITY		`%c1 has @PRIORITY("disabled")`
	in a pipe	`… => they have @PRIORITY("low")`

- `the x of` (or `the y of`) is written only before ORDER and DISTRIBUTE whose direction or axis is `"x"` (or `"y"`), and before a PLACE that gives one axis (B.1); ALIGN takes the gloss `, i.e., same x` instead.

- The x is that of the position of each member: for an instance, the point that PLACE sets (the origin of its symbol), not the centre of its outline; a set counts by its centre (14.1).

B.3 Appearance

Keyword	Case	Sentence
SET-SYMBOL	one object	`#M1 @SET-SYMBOL to $NMOS`
	a set	`{#M1 #M2} @SET-SYMBOL to $NMOS`
	the block symbol of a subcircuit	`this subcircuit # @SET-SYMBOL to $NAND2_SYM`
	in a pipe	`… => they @SET-SYMBOL to $PMOS`
SET-STYLE		`{#CC #RZ} @SET-STYLE to $orange`
	in a pipe	`… => they @SET-STYLE to $grey`
DEFINE-STYLE		`$orange = @DEFINE-STYLE(linecolor="#e07000", textcolor="#e07000") {}`
	merging another style	`$wide = @DEFINE-STYLE(linewidth=2) based on {$orange}`

B.4 Symbols and graphics

Keyword	Case	Sentence
DEFINE-SYMBOL		`$TRI = @DEFINE-SYMBOL {…}`
PIN		`at (-4, 0) there is a @PIN(terminal="in", direction="left")`
LINE		`a @LINE running through {(0, 0) (4, 0)}`
ARC		`an @ARC running through {(0, 2) (2, 0) (0, -2)}`
BEZIER		`a @BEZIER with control points {(0, 0) (1, 1) (2, 1) (3, 0)}`
CIRCLE		`centered at (0, 0), a @CIRCLE(radius=1)`
RECTANGLE	centre and size	`centered at (0, 0), a @RECTANGLE(width=4, height=2)`
	two corners	`a @RECTANGLE with corners {(0, 0) (4, 2)}`
ELLIPSE	centre and size	`centered at (0, 0), an @ELLIPSE(width=4, height=2)`
	corners of its box	`an @ELLIPSE with box corners {(0, 0) (4, 2)}`
TEXT	in a symbol	`on ? there is a @TEXT("${name}", key="name") => it is at @PLACE(-2, 2.5)`
ATTACH	in a symbol	`$ARROW is @ATTACH to (0, -1.5) => …`

- Graphics written in the scene, as annotations (13.4), read the same.

B.5 Wires and text

Keyword	Case	Sentence
WIRE		`net &n1 has a @WIRE connecting {#R1:2 %j #R2:1}`
	a bus	`net &D<3:0> has a @WIRE connecting {%p1 %p2}, which is a bus`
	a routing mode	`net &in has a @WIRE(mode="vertical-horizontal") connecting {#VIN:1 #R1:1}`
ROUTE		`net &n1 is fully @ROUTE`
	a set	`nets {&a &b} are fully @ROUTE`
	in a pipe	`… => it is fully @ROUTE`
TEXT	on an object	`on #M1 there is a @TEXT("W/L = ${w}/${l}", key="wl", dy=-3)`
	at a point	`at (_, _) there is a @TEXT("RC low-pass") => named %title`
	in a pipe	`… => on them there is a @TEXT("${w}", key="size")`

- Naming a point, such as `%j = (_, _)`, takes no added words.

B.6 Definitions and applications

Keyword	Case	Sentence
DEFINE		`$PAIR = a definition, @DEFINE(roles=["a", "b"], parameters=[["gap", 2]]) {…}`
AS	two roles	`#M1(in_p) and #M2(in_n) are treated @AS $DIFFERENTIAL_PAIR`
	a role that takes a set	`{#M5 #M7 #M8} are treated @AS $MATCHED`
	one object	`#X1:3 is treated @AS $SUPPLY_HIGH`
	three roles and a parameter	`#M1(a), #M2(b) and %ax(axis) are treated @AS(priority="high") $MIRRORED`
	a parameter of the definition	`{#R1 {%rc} #M1} are treated @AS(direction="top-to-bottom") $FLOW`
	a usage in a role	`%dp(pair) and %cm(load) are treated @AS $OTA`
	named	`%dp = #M1(in_p) and #M2(in_n) treated @AS $DIFFERENTIAL_PAIR`
	named in a pipe, with another object	`#M1(in_p) and #M2(in_n) are treated @AS $DIFFERENTIAL_PAIR => named %dp => it and %cm are treated @AS $OTA`
ATTACH	on a net	`$GROUND is @ATTACH to &vss`
	on a terminal	`$NO_CONNECT is @ATTACH to #M1:4`
	on a point	`$JUNCTION is @ATTACH to %j`
	two roles	`$DIFF_PORT is @ATTACH to &inp(a) and &inn(b)`
	followed by other segments, named	`$GROUND is @ATTACH to &vss => it is at @PLACE(24, 38) => named %g0`
	on each of several objects	`$SIZE_LABEL is @ATTACH to {#M1 #M2 #M3}`
	the fill, on a set	`$GROUND_STUB is @ATTACH to $GROUND_FILL_SET`
	the fill, in a pipe	`among the terminals at each level in # and below, @SELECT-TERMINAL(net="vss", wired=false) => $GROUND_STUB is @ATTACH(gap=3) to them`

- AS is used when the objects together make up the thing, such as a circuit structure, a placement, or a local way of drawing such as `$RC_STAGE`; ATTACH when the thing is added on the objects, such as a ground, a port tag, a no-connect mark, a junction dot, a label for each instance, or the fill. The two keywords do the same thing; only the place of the definition differs (9.4).
- `gap` is a parameter of `$GROUND_STUB`, the least distance from the terminal down to the ground symbol (2 by default). Like the role names `in_p` and `axis`, it is a name given in a definition, not a standard word.

B.7 Selections

A selection is written as a choice among candidates: `among <candidates>, @SELECT-…(conditions)`. The wording of the candidates depends on the kind of the scope and on the kind selected:

Scope	Selecting	Candidates
`#`	any kind	`the <kind> at each level in # and below`
`#`, with `recursive=false`	any kind	`the <kind> in # only`
a set of the kind selected (a filter)	that kind	`{…}`
net	instances	`the instances connected to &en`
net	terminals	`the terminals connected to &en`
net	wires	`the wires on &en`
instance	terminals	`the terminals of #R1`
instance	nets	`the nets connected to #M1`
terminal	instances	`the instances owning #M1:g`
terminal	nets	`the nets connected to #M1:g`
wire	nets	`the nets of %w13`
instance, terminal, wire, net, graphic	labels	`the labels attached to #M1`
usage (no usages among its members)	any kind	`the members of %dp`
usage (usages among its members, or not known)	any kind	`all members of %st1 at any depth`

Scope	Selecting	Candidates
usage	usages (SELECT-USAGE)	`%st1 itself and the usages inside it`
any object	the usages that contain it (SELECT-ANCESTOR-USAGE)	`the usages containing #M1`
a set of several kinds (kind not known)		write `for %mixed_set, @SELECT-…` instead

- `<kind>` follows the keyword: instances, terminals, nets, wires, labels, annotations, usages.
- In the middle of a pipe, the scope in the candidates is written `them`, such as `=> among all members of them at any depth,`. Whether the usages selected before have usages among their members is not known when the sentence is written, so `at any depth` is used.
- For the members of usages the wording is `at any depth`, not `at any level`: a level is a level of the circuit, as in `at each level in # and below`.
- A rule in a package applies at every level that uses the package, so in a package `at each level in # and below` becomes `at every level that uses this package, #`.

Keyword	Case	Sentence
SELECT-INSTANCE	this level and below	`among the instances at each level in # and below, @SELECT-INSTANCE(model="nch") => they @SET-SYMBOL to $NMOS`
	this level only	`among the instances in # only, @SELECT-INSTANCE(type="pmos", recursive=false) => …`
	a filter	`among {#M1 #M2 #R1}, @SELECT-INSTANCE(name="M*") => they @ORIENT("MY")`
	on a net	`among the instances connected to &en, @SELECT-INSTANCE => they @SET-STYLE to $grey`
	owning a terminal	`among the instances owning #M1:g, @SELECT-INSTANCE => they are on @LAYER("top")`
	members of a usage	`among all members of %st1 at any depth, @SELECT-INSTANCE(role="pair")`
	members of a usage without nested usages	`among the members of %dp, @SELECT-INSTANCE`
	in the middle of a pipe	`among the usages at each level in # and below, @SELECT-USAGE of $DIFFERENTIAL_PAIR => among all members of them at any depth, @SELECT-INSTANCE => they @SET-STYLE to $signal`
SELECT-TERMINAL	this level and below	`among the terminals at each level in # and below, @SELECT-TERMINAL(net="vss", wired=false) => $GROUND_STUB is @ATTACH(gap=3) to them`
	of an instance	`among the terminals of #R1, @SELECT-TERMINAL => on them there is a @TEXT("${net}")`
	on a net	`among the terminals connected to $GROUND_NET_SET, @SELECT-TERMINAL(wired=false) => $GROUND_STUB is @ATTACH to them`
	a filter	`among {#M1:s #M2:s}, @SELECT-TERMINAL(wired=false) => $GROUND is @ATTACH to them`
SELECT-NET	this level and below	`among the nets at each level in # and below, @SELECT-NET(port=true) => $IO_PORT is @ATTACH to them`
	of an instance	`among the nets connected to #M1, @SELECT-NET => named %m1_net_set`
	of a terminal	`among the nets connected to #M1:g, @SELECT-NET => they are fully @ROUTE`
	of a wire	`among the nets of %w13, @SELECT-NET => they are fully @ROUTE`
	a filter	`among {&a &b}, @SELECT-NET(wired=false) => they are fully @ROUTE`
SELECT-WIRE	this level and below	`among the wires at each level in # and below, @SELECT-WIRE(bus=true) => they @SET-STYLE to $bus`
	on a net	`among the wires on &clk, @SELECT-WIRE => they @SET-STYLE to $red`

Keyword	Case	Sentence
SELECT-LABEL	this level and below	`among the labels at each level in # and below, @SELECT-LABEL(key="wl") => they @SET-STYLE to $grey`
	attached to an object	`among the labels attached to #M1, @SELECT-LABEL => they @SET-STYLE to $grey`
SELECT-ANNOTATION	this level and below	`among the annotations at each level in # and below, @SELECT-ANNOTATION => they @SET-STYLE to $base`
	a filter	`among {%n1 %f1 #M1}, @SELECT-ANNOTATION`
SELECT-USAGE	this level and below	`among the usages at each level in # and below, @SELECT-USAGE of $DIFFERENTIAL_PAIR => they have @PRIORITY("disabled")`
	without a definition	`among the usages at each level in # and below, @SELECT-USAGE => they @SET-STYLE to $base`
	a usage and the usages inside it	`among %st1 itself and the usages inside it, @SELECT-USAGE of $DIFFERENTIAL_PAIR`
SELECT-ANCESTOR-USAGE		`among the usages containing #M1, @SELECT-ANCESTOR-USAGE of $DIFFERENTIAL_PAIR => named %dp_set`
	a terminal	`among the usages containing #M1:3, @SELECT-ANCESTOR-USAGE of $GROUND => they @ORIENT("R180")`
any selection	kind of the scope not known	`for %mixed_set, @SELECT-TERMINAL => …`

B.8 Sets and other keywords

Keyword	Case	Sentence
ADD-TO		`&vdd @ADD-TO $SUPPLY_NET_SET`
	in a pipe	`… => they @ADD-TO $SUPPLY_NET_SET`
REMOVE-FROM		`{&vdd &vss} @REMOVE-FROM $PORT_FILL_SET`
UNION		`%u_set = %a_set and %b_set @UNION`
BUNDLE		`#XI<3:0> is drawn as a @BUNDLE => named %xi`
CURRENT	in	`#M5:1 has @CURRENT("in")`
	out	`#:iout has @CURRENT("out")`
	unspecified	`#M5:2 has @CURRENT("unspecified")`

B.9 Header record and packages

No words are added. The header record is read in the standard spelling and does not use aliases (16.5).

Keyword	Case	Sentence
VERSION		`@VERSION("0.3")`
ALIAS		`@ALIAS("@位于", "@PLACE")`
USE-ALIASES		`@USE-ALIASES(name="rnl-zh", version="0.2")`
DEVICE		`@DEVICE(…)`
PACKAGE		`@PACKAGE(name="acme-style", version="1.2")`
USE-PACKAGE	in a header record; also a reference in the header record of a package file	`@USE-PACKAGE(name="rnl-std", version="0.2")`
	in a level	`@USE-PACKAGE(name="acme-style", version="1.2")`

Annex C Reserved for later versions (informative)

This annex lists what this version of the standard leaves out and says may come in a later version, with the sections that mention it. This version reserves no keywords or forms of writing for these items (5.3, 16.4), and a later version need not add them.

Left for a later version	See
Carriers for further hosts, including block comments (such as `#com … #endcom` of Eldo), and the handling of a comment terminator of the host that this brings	2.2, 2.5
Record IDs; tools find their records by `type`, `author` and the keys of statements	2.8
Expressions	3.7
Standard sentences that say whether a definition is applied to a whole set or once for each member (`each`)	4.5
Optional slots; without them a selection always has a scope, if only `#`	5.3, 10.1, 13.7
More than one subscript expanded in one reference, and further forms of subscripts, such as `<*2>A`, `A<0:2*2>` and `D[0..7]`	6.11
Standard diagnostic codes	7.6
A way to suppress diagnostics in a file, and a strict mode, for instance as fields of the record head	7.6
Units of several facts: a definition or a usage whose facts are kept or discarded together in conflict resolution. Still open: the priority of such a unit, nested usages, and priorities written from outside through `::`	8.5
Scaling of symbols and of instances	9.2.1, 11.7
Arrows at the ends of wires	9.2.5
More line styles	9.3
Colour names and `rgb()`	9.3
Pattern fills, gradients and shadows	9.3
Assertion statements, for checks such as two roles not being filled with the same object	9.5
The intersection of sets in one statement	10.1
Choosing the symbol of a primitive by its number of terminals or by its parameters. For a few primitives of each dialect, the simulator decides the kind by the line, and the conditions `name`, `model` and `type` cannot tell the kinds apart: in Spectre, a `switch` with two terminals is single-throw and one with three double-throw; in SPICE, a B source written with `V=` is a voltage source and one with `I=` a current source. Subcircuits and models do not vary in this way, since their ports and model types are fixed. One way is general conditions of SELECT-INSTANCE, such as `terminals=` (the number of terminals) and `param=` (the line has this parameter). Another is a type that the adapter of the dialect gives these primitives (6.12), such as `"current"` for B and `"spdt"` for a switch with three terminals, selected with `type=`, which then also compares the type that the adapter gives a primitive. Until then, the standard package draws both kinds by its present rules	10.1, 6.12
Choosing a symbol by the language mode in which an instance is written. In a Spectre file, a section after `simulator lang=spice` may hold instances to draw, such as `I1 a b 1m`; the file is in the Spectre dialect (16.3), so they find their symbols by the rules of Spectre and may be drawn as boxes. Using the rules of SPICE with a USE-PACKAGE would also draw the Spectre instances of the whole file that are named like `I0` as current sources. To tell the sections apart, a rule would have to see in which syntax an instance is written. This is the same kind of question as choosing by the number of terminals or by parameters, above, and is to be considered with it. The devices that an included file of the other dialect writes outside subcircuits belong to the level that includes it (6.1), and are treated the same way	16.3, 10.1
Rotating a whole set as one block with ORIENT	11.2
Several drawings of one host object, as separate objects with names of their own	11.4
A graphic that frames a group of objects	13.4
Annotations anchored on wires: positions along a wire (such as `t` or `offset`), and symbols or arrows anchored in the middle of a wire	13.3, 15.2
More of TeX in texts, with a math mode written `\(…\)`	13.6
References in `${…}` to the attributes of other objects, such as `${R3:VALUE}` of KiCad	13.6

Left for a later version	See
The diagonal projections `"x+y"` and `"x-y"` for constraints	14.1
A place in the file for the settings of an implementation, such as those of a solver, as content that is recognized but not interpreted	14.8, 16.4
A USE-PACKAGE that applies to its own level only	16.2
A package split over several files	16.2
A manifest format for a netlist and its package files	16.2
A content digest for packages, such as a `digest=` parameter of USE-PACKAGE	16.2
References that a package uses for itself only, apart from those it passes on to its users	16.2
Package-qualified names, other names for packages and level tags, such as `$acme.DIFFERENTIAL_PAIR` or `as=`, in a form other than a `.` after the name	16.2
An extension mechanism: keywords with a vendor prefix, such as `@ACME.ROUTE`, declared in the header record as required or optional extensions	2.8, 16.4
Exact RNL, that is, the conditions under which a file can be drawn without a solver, and criteria for round trips; these may instead be agreed between tools	17

Annex D Changes from version 0.2 (informative)

Version 0.3 lets the solver write back every decision it makes, so that a viewer that does not solve draws what the solver drew. Versions 0.x make no promise of compatibility with each other (16.1); this annex lists what changed and how a file moves to 0.3.

Change	See
Objects that cannot be reached by a name have numbers, such as `%#3`, used only in records of the solver. The record head field `numbering` holds a check of the numbers; when it does not match, the statements that use numbers take no effect	5.9, 2.8, 3.2, 5.7, 7.6, 9.4, 13.1, 13.3
Given a wire in place of a net, WIRE rewrites that wire; the solver writes the routes of `"auto"` wires this way	13.1, 5.1, 5.3, 5.8, 10.2
The solver must write back every decision it makes, and passes its results to other tools only in a form that a record of the solver can hold	14.8, 1.6, 8.2, 8.5, 12.1, 13.3, 17
When the solver solves, the records with `type="generated"` are left out of the interpretation; how else the solver uses them is up to it. Version 0.2 said that the solver must not read them	2.8, 14.8, 17
Clarified: the priority of a usage governs the PRIORITY statements in the body of its definition; a PRIORITY written from outside through `::` on a local usage governs what that usage expands to; a group in a disabled usage can be turned on again from outside	14.8, 11.5
Clarified: the names bound in a record with `scope` are published to the target level, where binding one again to the same object is allowed	2.6
A note on how conflicts may be resolved	8.5
Clarified: facts have a solution when real coordinates satisfy them within the tolerance, and integer points are only an aim; a value without a finite decimal form is written back with enough digits	8.5, 11.7
An implementation may limit the time it spends on checking for a solution; a fact whose check does not finish is discarded with a warning. This limit and the tolerance are options of the solver that the user can set; the tolerance is greater than 0	8.5, 7.6, 11.7
Clarified: an object that the solver writes back takes the place of the one that a rule generates only when the rule leaves out what is written, as the fill of the standard package does	10.5
The hint of a discarded fact names every set of facts that it conflicts with, and each statement in these sets is told which facts were discarded because of it; for a fact of priority `"low"` these hints may be left out	8.5
Clarified: a fact that has no solution on its own contradicts itself; it fails and takes no part in resolution, as 9.5 of version 0.2 already shows	8.5, 9.5
Clarified: each segment of a pipe has a place of its own in the text, so a later segment counts as written later wherever the place of writing matters, such as in override by key, the appearance queues and the order of resolution	5.5, 1.5, 8.3, 8.4, 8.5, 13.1
How an editor changes an object that the solver wrote out	17
In a file of version 0.3, the rule package of the other dialect is dropped implicitly: a file in the SPICE dialect does not use `rnl-std-spectre`, and a file in the Spectre dialect does not use `rnl-std-spice`. The dialect of a file is the language mode of its first statement; a USE-PACKAGE of either package on the chain of levels keeps it there	16.3
Terminal names by default for T and O (`p1`, `n1`, `p2`, `n2`), U (`a`, `b`, `ref`) and Z (`d`, `g`, `s`), by the version of the file in which the instance line stands	6.12
The default terminal names of Spectre are listed: of its primitives, of `switch`, and of the instances of model cards by the family of the model, with the type that the condition `type` compares	6.12, 10.1
Clarified: a section of `.lib` that is read counts as a file, and its header record is the first record in the section	2.8
Clarified: when a name of a file has the spelling of a standard word that a newer version adds, raising the version gives a warning and is done all the same	16.1
The priority of the record head is no default for AS and ATTACH, except `"disabled"`: a usage keeps the priorities of its body unless it is given one; a disabled record still turns off its usages	14.8, 2.8, 10.5

Change	See
with their bodies. The solver writes out a usage that a rule made with the priority that the rule writes on it, as written, or with the `"disabled"` of a disabled record, and otherwise with none	
The implicit standard package follows the language version of the file: the series 0.2 for a file of version 0.2, the series 0.3 for a file of version 0.3; a pinned version comes first	16.3, 16.1
Version 0.3 replaces 0.2, which tools may still read: they write 0.3, and raise a file of 0.2 that they write into, with a pin of the standard series 0.2, so that the file keeps its names and its drawing	16.1, 16.3

Moving a file from 0.2 to 0.3. Raise the version in the header record (16.1); a tool that writes into a file of 0.2 does so itself, and pins the series 0.2 (16.1). Version 0.3 adds one standard word, the record head field `numbering`; in 0.2 it was an unknown field. Raising the version warns about the names that have its spelling (16.1), and in 0.3 an alias of that spelling is an error (16.5). A record of the solver written for 0.2 uses no numbers and stays valid; the next solve also writes back the decisions that 0.2 could not hold. Some results can differ from what a tool made of 0.2: a priority written from outside on a local usage decides for everything that the usage expands to (14.8); a file no longer uses the rule package of the other dialect, so that in a Spectre netlist an instance such as `I0` is no longer drawn as a current source (16.3); and T, O, U and Z instances have terminal names by default, which a DEVICE of the file still overrides (6.12). Raised to 0.3, a file that does not pin the standard package also moves from the series 0.2 to the series 0.3, which renames some definitions and symbols; to keep the names of 0.2, pin it with `@USE-PACKAGE(name="rnl-std", version="0.2")` in the header record (16.3); see the standard series 0.3 below.

The standard series 0.3. The standard package is not part of this standard (16.3), and the README of the standard packages lists in full what its series 0.3 changes. What follows is what matters to a file that moves from the series 0.2 to the series 0.3.

Renamed. The names of 0.3 are written out in full, and the symbol of a virtual part is the name of its definition followed by `_SYMBOL`. The roles and parameters keep their names, but for the role `series` of `$CHAIN` (below); only these names change:

Series 0.2	Series 0.3	Kind
`$GND`	`$GROUND`	definition
`$IOPORT`	`$IO_PORT`	definition
`$NC`	`$NO_CONNECT`	definition
`$DIFFPAIR`	`$DIFFERENTIAL_PAIR`	definition
`$DIFFPAIR_TAIL`	`$DIFFERENTIAL_PAIR`, with its `tail` filled	definition
`$MIRROR_PAIR`	`$MIRRORED`	definition
`$GND_STUB`	`$GROUND_STUB`	definition
`$GND_FILL_SET`	`$GROUND_FILL_SET`	set
`$LABEL_TERM_SET`	`$LABEL_TERMINAL_SET`	set
`$GROUND`	`$GROUND_SYMBOL`	symbol
`$SUPPLY_BAR`	`$SUPPLY_SYMBOL`	symbol
`$PORT_TAG`	`$IO_PORT_SYMBOL`	symbol
`$NC_CROSS`	`$NO_CONNECT_SYMBOL`	symbol
`$JUNCTION_DOT`	`$JUNCTION_SYMBOL`	symbol
`$CURRENT_ARROW_SYM`	`$CURRENT_ARROW_SYMBOL`	symbol
`$ARROW_HEAD`	`$ARROW_SYMBOL`	symbol
`$RES`	`$RESISTOR`	symbol
`$CAP`	`$CAPACITOR`	symbol
`$IND`	`$INDUCTOR`	symbol
`$VSRC`	`$VOLTAGE_SOURCE`	symbol
`$ISRC`	`$CURRENT_SOURCE`	symbol

Series 0.2	Series 0.3	Kind
`$NMOS4`	`$NMOS_4TERM`	symbol
`$PMOS4`	`$PMOS_4TERM`	symbol

`$GROUND` is in both series with another meaning: in 0.2 it is the symbol of a ground, in 0.3 the definition of a ground, which 0.2 calls `$GND`. So a file renames `$GROUND` first and `$GND` after it, and only whole names: the `$GND` in `$GND_STUB` is not a name of its own.

Same name, other drawing or meaning.

- `$VCVS` and `$VCCS` draw only the two output terminals. The symbols with all four are `$VCVS_4TERM` and `$VCCS_4TERM`, which E and G instances, and the masters `vcvs` and `vccs` of Spectre, get by default. In a file that gives `$VCVS` or `$VCCS` to an instance itself, the instance now has two pins, and the wires to its terminals 3 and 4 fail (9.2.4); the file gives it the symbol with `_4TERM` instead.
- The symbols are drawn anew, after those of Analog Canvas, and many have their pins at other points: the pins of a two-terminal part such as `$RESISTOR` are 4 apart, not 6; the diode lies across; the + input of `$OPAMP`, terminal 1, is at the top. Lines are 0.16 wide, where 0.2 draws hairlines. Positions and routes written back for the symbols of 0.2 may therefore no longer meet the pins; such a file is solved again, or pins the series 0.2.
- `$DIFFERENTIAL_PAIR` has a third role, the set role `tail`, which may be left out, so a pair written as in 0.2 means the same.
- Both roles of `$CURRENT_MIRROR` are set roles: a mirror with several reference or output transistors is one usage, where `#M3(ref) {#M4 #M6}(out)` made two usages in 0.2.
- The set role `series` of `$CHAIN` is called `chain`.
- A constraint of priority `"low"` in the body of `$DIFFERENTIAL_PAIR` puts the tail under the middle of the pair (9.5), so a file that does not place its tail is drawn otherwise than in 0.2; `%dp1 @PRIORITY("disabled")` turns it off for one pair (14.8).
- Every constraint in the body of a definition of the standard package is of priority `"low"` (9.5). `$MIRRORED` puts a on the left of b (14.7); `$BRANCH` puts its members in one column, and `$CHAIN` its source, its chain and its load from left to right.
- The four roles of `$CASCADE` are set roles: a stage of several objects is one set, where `{#RZ #CC}(feedback)` made one usage for each member in 0.2.
- `$OPAMP` is no longer in `rnl-std` but in `rnl-std-functional`, which a file names in its header record, alone or through `rnl-std-full` (16.3).
- `$GROUND_STUB` and `$SUPPLY_STUB` give only the least distance `gap` from the pin, no longer an exact one, so that a solver may draw the wire longer, and point the ground down and the supply up, `@ORIENT("R0")`, all at priority `"low"` (10.5).
- The fill puts no port tag on a ground net or a supply net, and draws the grounds of a ground net other than 0, such as vss added to `$GROUND_NET_SET`, with `$NAMED_GROUND_SYMBOL`, which shows the name of the net (10.5).
- Instances that the series 0.2 drew without a symbol of its own now get one by default: S with `$VOLTAGE_CONTROLLED_SWITCH`, W with `$SWITCH`, B with `$BEHAVIORAL_SOURCE`, and F, H, J, Z, T, O and U of SPICE with theirs, as do the masters of Spectre for these elements, such as `relay` and `switch`.

Removed. `$INPUT_STAGE`: its tail goes into the role `tail` of its differential pair, as in `#M1(in_p) #M2(in_n) #M5(tail) @AS $DIFFERENTIAL_PAIR`. `$TAIL_CENTER`: the standard package centres the tail of each differential pair itself; to centre other objects, a file writes a definition of its own, such as `$TAIL_MID` of 9.4. `rnl-std-semantics` and `rnl-std-layout`: they are one package, `rnl-std-structures` (16.3); a file that dropped `rnl-std-layout` to do without the default layouts writes, for each definition, a rule such as `# @SELECT-USAGE $MIRRORED => @PRIORITY("disabled")` (14.8).

New. `rnl-std-basic` and `rnl-std-semiconductors` add symbols, such as switches, current-controlled sources, transmission lines and JFETs. The series also adds packages that `rnl-std` does not bring in: `rnl-std-functional`, `rnl-std-rf`, `rnl-std-logic` and `rnl-std-signal-flow`, the collection `rnl-std-full` of these with `rnl-std`, and the optional `rnl-std-no-labels` (16.3). A name that a file defines itself and that the series 0.3 adds, such as `$SWITCH`, hides the standard one with a hint (6.8), so the file keeps its own.

Revision 2. An editorial revision; the language does not change. The examples of 1.3 and 5.9 write the model card of their transistors, `.model nch nmos level=1`, so that the standard package can tell their polarity and give them a symbol.

Later editorial clarification. Section 13.3 makes explicit that a text whose two position axes are supplied by the scene does not inherit its attached object's orientation; its alignment is interpreted in the scene's directions. The language does not change.