Layout vs. Schematic (LVS)

LVS is a verification step which checks whether a layout matches the circuit from the schematic. The LVS feature is described in the following topic chapters:

- Layout vs. Schematic (LVS) Overview
- LVS Introduction
- LVS Devices
- LVS Device Classes
- LVS Devices Extractors
- LVS Input/Output
- LVS Connectivity
- LVS Compare
- LVS Netlist Tweaks

A reference for the functions and objects available for LVS scripts can be found here: LVS Reference.

Table of Contents

Layout vs. Schematic (LVS) Overview

Basic usage of LVS scripts

Starting with version 0.26, KLayout supports LVS as a built-in feature. LVS is an important step in the verification of a layout: it ensures the drawn circuit matches the desired schematic.

The basic functionality is simply to analyze the input layout and derive a netlist from this. Then compare this netlist against a reference netlist (schematic). If both netlist are equivalent, the circuit is likely to work in the intended fashion.

Beside the layout, a LVS script will also need a schematic netlist. Currently, KLayout can read SPICE-format netlists. The reader can be configured to some extent, so the hope is that a useful range of SPICE netlists can be digested.

While the basic idea is simple, the details become pretty complex. This documentation tries to cover the solutions KLayout offers to implement LVS as well as the constraints imposed by this process.

KLayout's LVS is integrated into the Macro Development IDE the same way as DRC scripts. In fact, LVS is an add-on to DRC scripts. All DRC functions are available within LVS scripts. Netlist extraction is performed in the DRC framework which was given the ability to recognize devices and connections and turn them into a netlist. Although DRC does not really benefit from these extensions, they are still useful for implementing Antenna checks for example. As it happens, the majority of features required for LVS is documented in the DRC Reference, while the few add-ons required specifically for LVS are documented in LVS Reference.

LVS scripts are created, edited and debugged in the Macro Editor IDE. They are managed in the "LVS" tab. For more details about the IDE, see About Macro Development. For an introduction about how to work with DRC scripts see Design Rule Checks (DRC) Basics.

LVS scripts carry the ".lylvs" extension for the XML form (in analogy to ".lydrc" for DRC) and ".lvs" for the plain text form (same as ".drc"). Like DRC scripts, LVS scripts can be executed standalone in batch mode like DRC scripts. See "Using KLayout as a standalone DRC engine" in Design Rule Checks (DRC) Basics.

KLayout's LVS implementation

The LVS implementation inside KLayout is designed to be highly flexible in terms of connectivity, device recognition and input/output channels. Here are some highlights:

- **Agnostic approach:** KLayout tries to make as few assumptions as possible. It does not require labels (although they are helpful), a specific hierarchy, specific cell names or specific geometries. Netlist extraction is done purely from the polygons of the layout. Labels and the cell hierarchy add merely useful hints which simplify debugging and pin assignment, but no strict requirement.
- **Hierarchical analysis:** KLayout got a hierarchical layout processing engine to support hierarchical LVS. Hierarchical processing means that boolean operations happen inside the local cell environment as far as possible. As a consequence, devices are recognized inside their layout cell and layout cells are turned into respective subcircuits in the netlist. The netlist compare will benefit as it is able follow the circuit hierarchy. This is more efficient and gives better debugging information in case of mismatches. As a positive side effect of hierarchical layout processing the runtimes for some boolean and other operations is significantly reduced in most cases.
- **Hierarchically stable:** KLayout won't modify the layout's hierarchy nor will it introduce variants - at least for boolean and some other operations. This way, matching between layout and schematic hierarchy is maintained even after hierarchical DRC operations. Variants are introduced only for some anisotropic operations, the grid snap method and some other features which require differentiation of cells in terms of location and orientation.
- **Flexible engine:** The netlist formation engine is highly flexible with respect to device recognition and connectivity extraction. First, almost all DRC features can be used to derive intermediate layers for device formation and connectivity extraction. Second, the device recognition can be scripted to implement custom device extractors. Five built-in device extractors are available for MOS and bipolar transistors, resistors, capacitors and diodes.
- **Flexible I/O:** Netlists are KLayout object trees and their components (nets, devices, circuits, subcircuits ...) are fully mapped to script objects (for the main class see Netlist in the API documentation). Netlists can therefore be analyzed and manipulated within LVS scripts or in other contexts. It should be possible to fully script readers and writers for custom formats.

Netlists plus the corresponding layout elements (sometimes called "annotated layout") can be persisted in a KLayout-specific, yet open format. SPICE format is available to read and write pure netlist information. The SPICE reader and writer is customizable through delegate classes which allow tailoring of the way devices are read and written.

- **User interface integration:** KLayout offers a browser for the netlist extraction results and LVS reports (cross-reference, errors).
-

Terminology

KLayout employs a specific terminology which is explained here:

- **Circuit:** A graph of connected elements as there are: devices, pins and subcircuits. The nodes of the graph are the nets connecting at least two elements. If derived from a layout, a circuit corresponds to a specific layout cell.
- **Abstract circuits:** Abstract circuits are circuits which are cleared from their inner structure. Such circuits don't have nets and define pins only. Abstract circuits are basically "black boxes" and LVS is required to consider their inner structure as "don't care". Abstract circuits are useful to reduce the netlist complexity by taking out big IP blocks verified separately (e.g. RAM blocks).
- **Pin:** A point at which a circuit makes a connection to the outside. Circuits can embed other circuits as "subcircuits". Nets connecting to the pins of these subcircuits will propagate into the subcircuit and connect further elements there. Pins are usually attached to one net - in some cases, pins can be unattached (circuits abstracts). Pins can be named. Upon extraction, the pin name is derived from the name of the net attached to the pin.
- **Subcircuit:** A circuit embedded into another circuit. One circuit can be used multiple times, hence many subcircuits can reference the same circuit. If derived from a layout, a subcircuit corresponds to a specific cell instance.
- **Device:** A device is a n-terminal entity describing an atomic functional unit. Devices are passive devices (resistors, capacitors) or active devices such as transistors.
- **Device class:** A device class is a type of device. Device classes are of a certain kind and there can be multiple classes per type. For example for MOS transistors, the kind is "MOS4" (a four-terminal MOS transistor) and there is usually "NMOS" and "PMOS" classes at least in a CMOS process. A device class typically corresponds to a model in SPICE.
- **Device extraction:** Device extraction is the process of detecting devices and forming links between conductive areas and the device bodies. These links will eventually form the device terminals.
- **Device combination:** Device combination is the process of forming single devices from combinations of multiple devices of the same class. For example, serial resistors can be combined into one. More importantly, parallel MOS transistors ("fingered" transistors) are combined into a single device. Device combination is a step explicitly requested in the LVS script.
- **Terminal:** A "terminal" is a pin of a device. Terminals are typically named after their function (e.g. "G" for the gate of a MOS transistor).
- **Connectivity:** The connectivity is a description of conductive regions in the technology stack. A layer has intra-layer and intra-layer connectivity: "Intra-layer connectivity" means that polygons on the same layer touching other polygons form a connected - i.e. conductive - region. "Inter-layer connectivity" means that two layers form a connection where their polygons overlap. The

sum of these rules forms the "connectivity graph".

- **Netlist:** A hierarchical structure of circuits and subcircuits. A netlist typically has a top circuit from which other circuits are called through subcircuits.
- **Extracted netlist:** The extracted netlist is the netlist derived from the layout. Sometimes, "extracted netlist" describes the netlist enriched with parasitic elements such as resistors and capacitors derived from the wire geometries. In the context of KLayout's LVS, "extracted netlist" is the pure connectivity without parasitic elements.
- **Schematic:** The "schematic" is a netlist taken as reference for LVS. The "schematic" is thought of the "drawn" netlist that is turned into a layout by the physical implementation process. In LVS, the layout is turned back into the "extracted netlist" which is compared to the schematic.
- **Annotated layout, Net geometry:** The collection of polygons belonging to the individual nets. Each net inside a circuit is represented by a bunch of polygons representing the original wire geometry and the device terminals. As nets can propagate to subcircuits through pins, nets and therefore annotated layout carries a per-net hierarchy. The per-net hierarchy consists of the subcircuits attached to one net and the nets within these subcircuits that connect to the outer net. Subcircuits can instantiate other subcircuits, so the hierarchy may extend over many levels.
- **Layout to netlist database (L2N DB):** This is a data structure combining the information from the extracted netlist and the annotated layout into a single entity. The L2N database can be used to visualize nets, probe nets from known locations and perform other analysis and manipulation steps. An API for handling L2N databases is available.
- **Cross reference:** The cross reference is a list of matching objects from the two netlists involved in a LVS netlist compare ("pairing"). The cross-reference also lists non-matching items and inexact pairs. "Inexact pairs" are pairs of objects which do not match precisely, but still are likely to be paired. The cross reference also keeps track of the compare status - i.e. whether the netlists match and if not, where a mismatch originates from.
- **LVS database:** The "LVS database" is the combination of L2N database, the schematic netlist and the cross-reference. It's a complete image of the LVS results. An API is available to access the elements of the LVS database.
- **Labels:** "Labels" are text objects drawn in a layout to mark certain locations on certain layers with a text. Typically, labels are used to assign net names - if included in the connectivity, nets formed from such labels get a name according to the text string of the label.

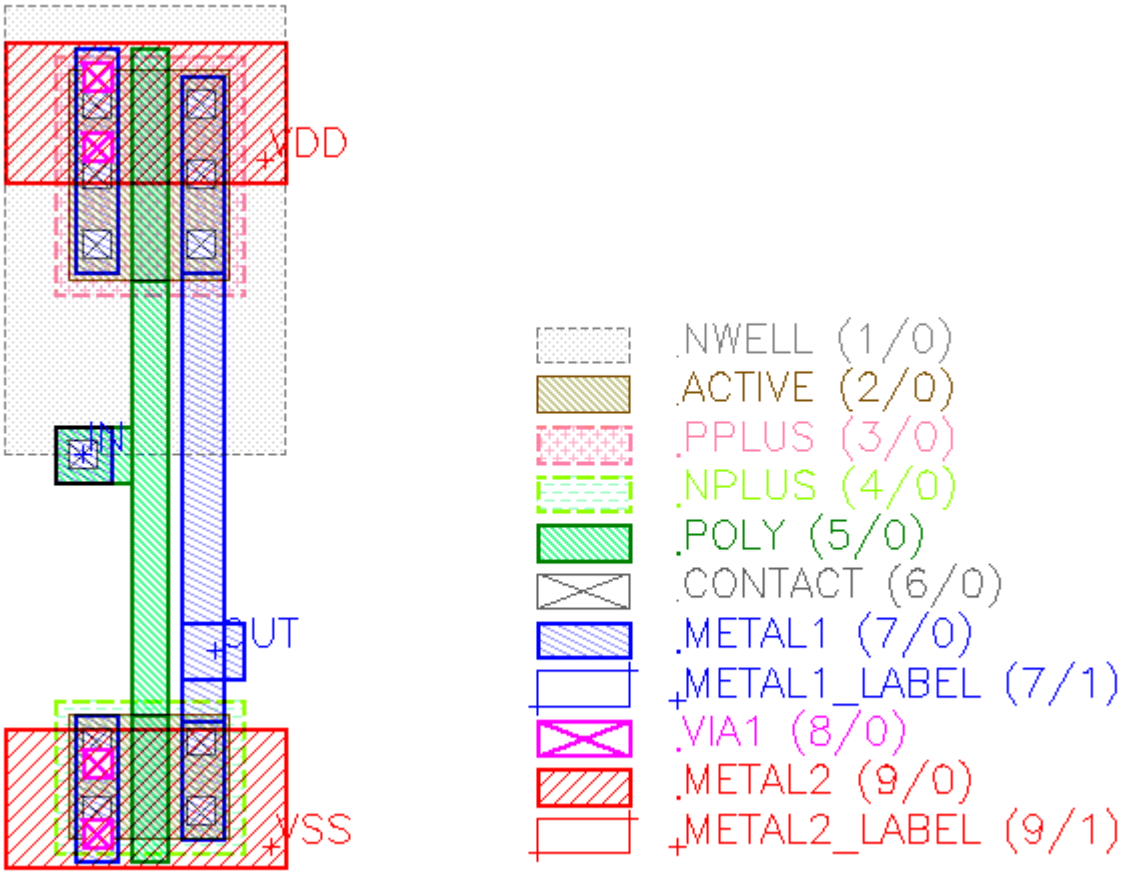
LVS introduction

For introducing the LVS feature we consider the most simple CMOS structure there is: the two-transistor inverter.

Layout

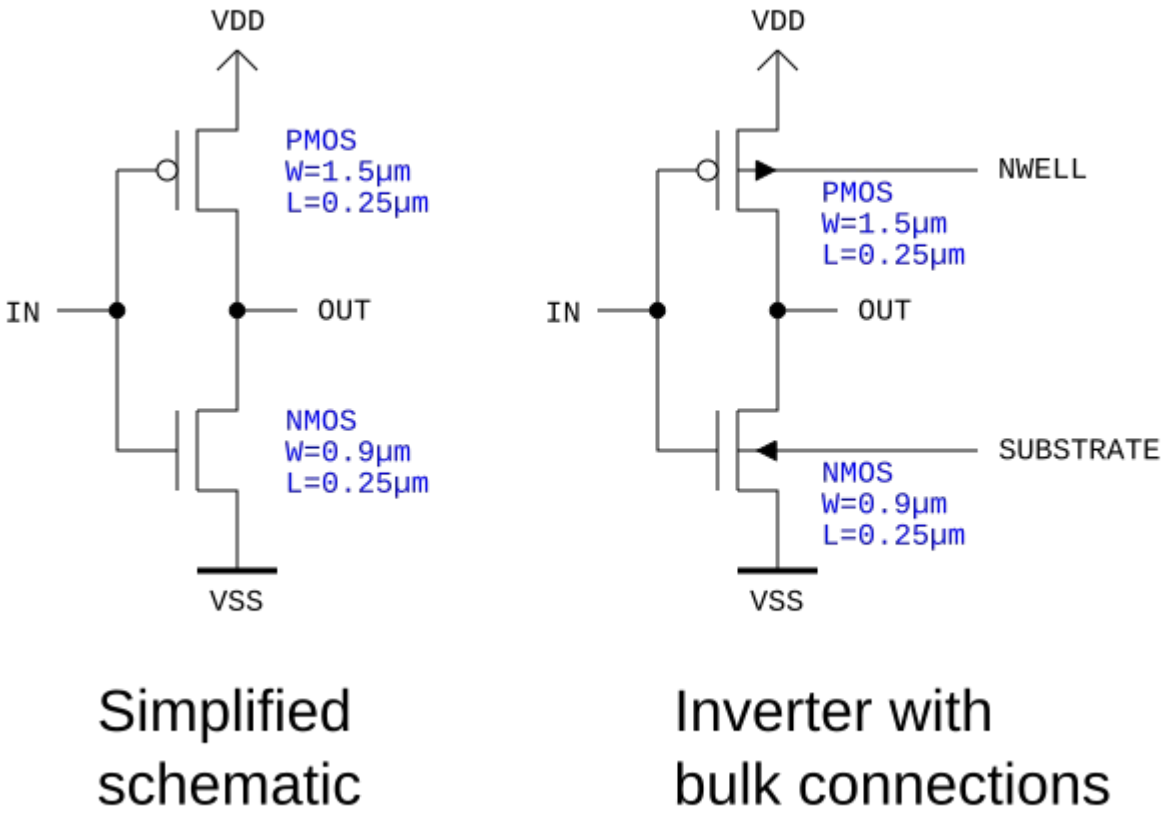
The inverter consists of two MOS transistors. A single transistor is made from an active region (a rectangle on the ACTIVE layer) and a gate (POLY layer) crossing the active region. The gate forms the channel from source to drain regions (left and right of gate). Contacts (CONTACT) provide connections from the first metal layer (METAL1) to the gate polysilicon (POLY) and to source/drain regions (where over ACTIVE). Via holes (VIA1) provide connections from the first (METAL1) to the second metal (METAL2). Finally, specific devices are formed by the source/drain implants which is n+ (NPLUS marker) for NMOS and p+ (PPLUS marker) for PMOS devices. PMOS devices sit in a n implant region (n-well) which forms the p-channel region. NMOS devices are built over substrate which is p doped to supply the n-channel region.

The actual layout is made as a standard cell. Multiple standard cells can be arrayed horizontally in a row. The power rails are formed in the second metal for VDD at the top and VSS at the bottom. The n-well extends over the top of the cell and is supposed to connect to neighbor well regions:



Schematic

For the inverter we can draw a schematic in a simplified form (left) and in a more realistic form (right) which also includes the bulk potentials of the transistors. It is important to keep the bulk of of the transistors at a defined potential to avoid latch-up. Hence we need pins for these terminals too. This makes a total of six pins: for input (IN) and output (OUT), for the power (VDD, VSS) and the two bulk potentials (NWELL, SUBSTRATE):



For LVS we first need a reference schematic. This is the SPICE netlist corresponding to the schematic with the bulk connections:

```
* Simple CMOS inverer circuit (inv.cir)
.SUBCKT INVERTER VSS IN OUT NWELL SUBSTRATE VDD
Mp VDD IN OUT NWELL     PMOS W=1.5U L=0.25U
Mn OUT IN VSS SUBSTRATE NMOS W=0.9U L=0.25U
.ENDS
```

The circuit we are going to analyze is a cell which is embedded in bigger circuits. Hence it makes sense to describe the inverter as a subcircuit. If the netlist consists of a subcircuit only, KLayout will consider this circuit. Otherwise it will consider the global definitions as the main circuit. In the latter case, pins cannot be defined while with subcircuits pins can be listed as given names too.

Sample LVS script

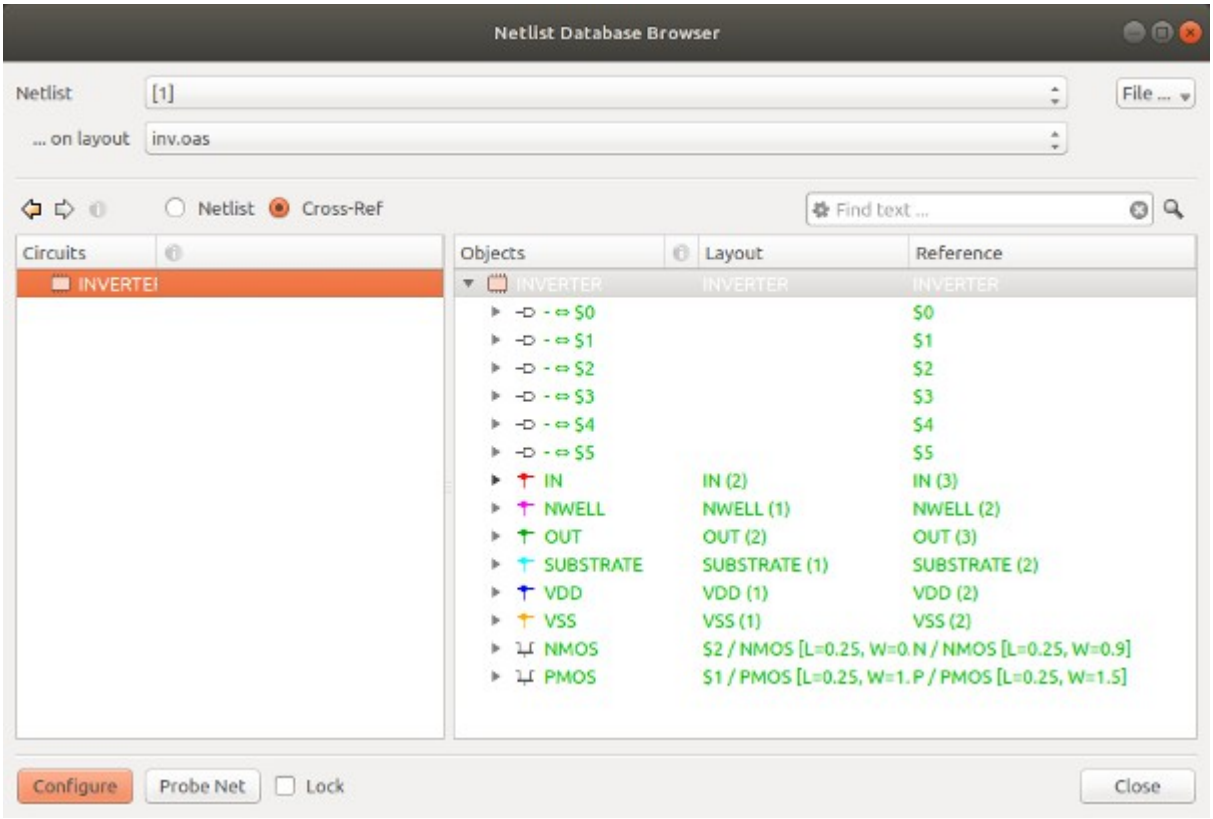
The LVS script to compare the layout above and the schematic now is this (for more details see LVS Reference):

```
# LVS script (demo technology, KLayout manual)

# Preamble:
```

```
deep

# Reports generated:

report_lvs     # LVS report window

# Drawing layers:

nwell       = input(1, 0)
active      = input(2, 0)
pplus       = input(3, 0)
nplus       = input(4, 0)
poly        = input(5, 0)
contact     = input(6, 0)
metal1      = input(7, 0)
metal1_lbl  = labels(7, 1)
via1        = input(8, 0)
metal2      = input(9, 0)
metal2_lbl  = labels(9, 1)

# Bulk layer for terminal provisioning:

bulk        = polygon_layer

# Computed layers:

active_in_nwell       = active & nwell
pactive               = active_in_nwell & pplus
pgate                 = pactive & poly
psd                   = pactive - pgate

active_outside_nwell  = active - nwell
nactive               = active_outside_nwell & nplus
ngate                 = nactive & poly
nsd                   = nactive - ngate

# Device extraction

# PMOS transistor device extraction
extract_devices(mos4("PMOS"), { "SD" => psd, "G" => pgate, "W" => nwell,
                                "tS" => psd, "tD" => psd, "tG" => poly, "tW" =>
nwell })

# NMOS transistor device extraction
extract_devices(mos4("NMOS"), { "SD" => nsd, "G" => ngate, "W" => bulk,
                                "tS" => nsd, "tD" => nsd, "tG" => poly, "tW" =>
bulk })

# Define connectivity for netlist extraction

# Inter-layer
connect(psd,        contact)
connect(nsd,        contact)
connect(poly,       contact)
connect(contact,    metal1)
connect(metal1,     metal1_lbl)   # attaches labels
connect(metal1,     via1)
connect(via1,       metal2)
```

```
connect(metal2,     metal2_lbl)    # attaches labels

# Global
connect_global(bulk,  "SUBSTRATE")
connect_global(nwell, "NWELL")

# Compare section

schematic("inv.cir")

compare
```

For trying this script, load the inverter layout from "testdata/lvs/inv.oas" (KLayout sources) and open the Macro Editor IDE (Tools/Macro Development). Create a new script in the LVS tab and paste the text from above. Then run the script. The LVS report browser will open and show everything in green. This indicates the compare was successful:



Anatomy of the LVS script

The first and important statement of a LVS script should be the "deep" switch which enables hierarchical mode. Without hierarchical mode, the netlist is produced without subcircuits. Such flat netlist are inefficient to compare and hard to debug. Hence we switch to hierarchical mode with the "deep" statement (see deep):

```
deep
```

We also instruct LVS to create a report and open it in the report browser once LVS has finished:

```
report_lvs
```

We can also write the report to a file if we want (see report_lvs):

```
report_lvs("inv.lvsdb")
```

The next step is the declaration of the input layers:

```
nwell       = input(1, 0)
active      = input(2, 0)
pplus       = input(3, 0)
nplus       = input(4, 0)
poly        = input(5, 0)
contact     = input(6, 0)
metal1      = input(7, 0)
metal1_lbl  = labels(7, 1)
via1        = input(8, 0)
metal2      = input(9, 0)
metal2_lbl  = labels(9, 1)
```

"input" and "labels" are functions which pull layout layers from the layout source (the layout source is - as in DRC - usually the current layout). While "input" pulls all kind of shapes, "labels" will only pull texts. We use "labels" to pull labels for first metal from GDS layer 7, datatype 1 and labels for second metal from GDS layer 9, datatype 1. For details see input and labels.

In addition, we create an empty layer which we will need to represent the "substrate". This layer does not constitute a closed region but rather a heap of shapes which will all connect to the same (global) net later:

```
bulk = polygon_layer
```

The names we give to the layers are actually variables which represent a layout layer. As in DRC, we can use these to compute some derived layers:

```
active_in_nwell       = active & nwell
pactive               = active_in_nwell & pplus
pgate                 = pactive & poly
psd                   = pactive - pgate

active_outside_nwell  = active - nwell
nactive               = active_outside_nwell & nplus
ngate                 = nactive & poly
nsd                   = nactive - ngate
```

These formulas are all boolean operations. "&" is the boolean AND operation and "-" is the boolean NOT. Hence "active_in_nwell" is the part of "ACTIVE" which is inside "NWELL" while "active_outside_nwell" is the part of "ACTIVE" outside it. The main purpose of these formulas is to separate source and drain regions but cutting away the gate area from the "ACTIVE" area. This renders "psd" and "nsd" (PMOS and NMOS source/drain). The boolean operations are part of the DRC feature set. For more functions and detailed descriptions see DRC Reference: Layer Object.

We also separate gate regions for PMOS (pgate) and NMOS transistors (ngate) and with these ingredients we are ready to move to device extraction:

```
extract_devices(mos4("PMOS"), { "SD" => psd, "G" => pgate, "W" => nwell,
                                "tS" => psd, "tD" => psd, "tG" => poly, "tW" =>
nwell })
```

The first argument of "extract_devices" (see extract_devices) is the device extractor. The device

extractor is an object responsible for the actual extraction of a certain device type. In our case the template is "MOS4" and we want to produce a new class of devices called "PMOS". `mos4("PMOS")` will create a new device extractor which produces devices of "MOS4" kind with class name "PMOS".

The second argument is a hash of layer symbols and layers. Each device extractor type defines a specific set of layer symbols. For all devices, two sets of the layers are required: the input layers which the extractor employs to recognize the device and the terminal connection layers which the extractor uses to place "magic" terminal shapes on. These polygons will create connections to the devices produced by the extractor.

The input layers are designated by upper-case letters, while the terminal output layers are designated with a lower-case "t" followed by the terminal name. The specification above is complete, but because "tW" defaults to "W" and "tS" and "tD" default to "SD", it can be written shorter as:

```
extract_devices(mos4("PMOS"), { "SD" => psd, "G" => pgate, "W" => nwell, "tG" =>
poly })
```

We also need an extractor for the "NMOS" class. It's built exactly the same way than the PMOS extractor:

```
extract_devices(mos4("NMOS"), { "SD" => nsd, "G" => ngate, "W" => bulk,
                                "tS" => nsd, "tD" => nsd, "tG" => poly, "tW" =>
bulk })
```

Having the devices is already half the work. We now need to supply the connectivity (see connect):

```
connect(psd,         contact)
connect(nsd,         contact)
connect(poly,        contact)
connect(contact,     metal1)
connect(metal1,      metal1_lbl)    # attaches labels
connect(metal1,      via1)
connect(via1,        metal2)
connect(metal2,      metal2_lbl)    # attaches labels
```

These statements will connect PMOS source/drain regions (psd) with CONTACT regions (contact), NMOS source/drain regions (nsd) also with CONTACT. POLY will also connect to CONTACT. Remember that we specified psd, nsd and poly as terminal outputs "tS", "TD" and "tG" in the device extraction. By including these layers into the connectivity, we establish device terminal connections to the nets formed by these layers.

The metal stack is trivial (CONTACT to METAL1, METAL1 to METAL2 via VIA1). The labels are attached to nets simply by including the label layers into the connectivity. The net extractor will pull the text strings from these connected text objects and assign them to the nets as net names.

Furthermore, two special connections need to be made (see connect_global):

```
connect_global(bulk,  "SUBSTRATE")
connect_global(nwell, "NWELL")
```

Global connections basically say that all shapes on a certain layer belong to the same net - even if they do not touch - and this net is always shared between circuits and subcircuits. This is certainly true for the bulk layer, but not necessarily for the NWELL layer. Isolated NWELL patches do not connect together. We will correct this small error later when it comes to extraction with tie-down diodes.

We have now provided all the essential inputs for the netlist formation. We only have to specify the

reference netlist:

```
schematic("inv.cir")
```

Finally after having set this up, we can trigger the compare step:

```
compare
```

If we insert a netlist write statement (see target_netlist) at the beginning of the script, we can obtain a SPICE version of the extracted netlist:

```
# SPICE output statement (insert at beginning of script):
target_netlist("inv_extracted.cir", write_spice, "Extracted by KLayout")
```

Since we have a LVS match, the extracted netlist is pretty much the same than the reference netlist, but enhanced by some geometrical parameters such as source and drain area and perimeter:

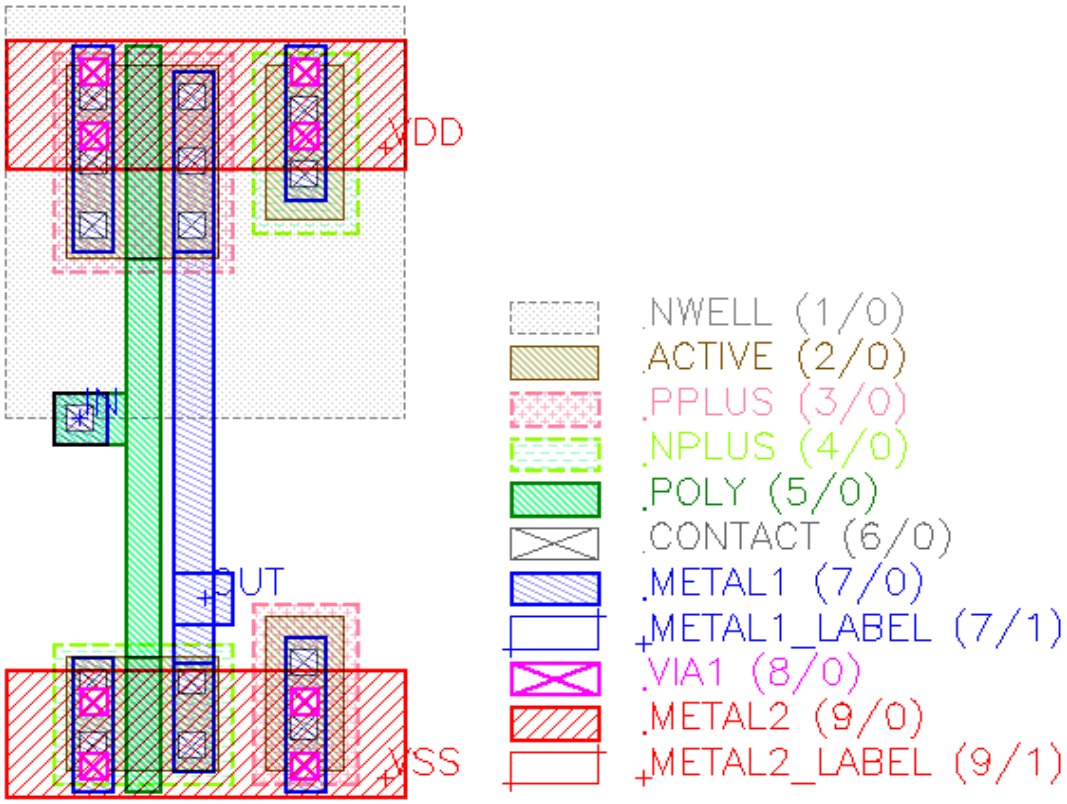
```
* Extracted by KLayout

* cell INVERTER
.SUBCKT INVERTER
* net 1 IN
* net 2 VSS
* net 3 VDD
* net 4 OUT
* net 5 NWELL
* net 6 SUBSTRATE
* device instance $1 r0 *1 1.025,4.95 PMOS
M$1 3 1 4 5 PMOS L=0.25U W=1.5U AS=0.675P AD=0.675P PS=3.9U PD=3.9U
* device instance $2 r0 *1 1.025,0.65 NMOS
M$2 2 1 4 6 NMOS L=0.25U W=0.9U AS=0.405P AD=0.405P PS=2.7U PD=2.7U
.ENDS INVERTER
```

Inverter with tie-down diodes

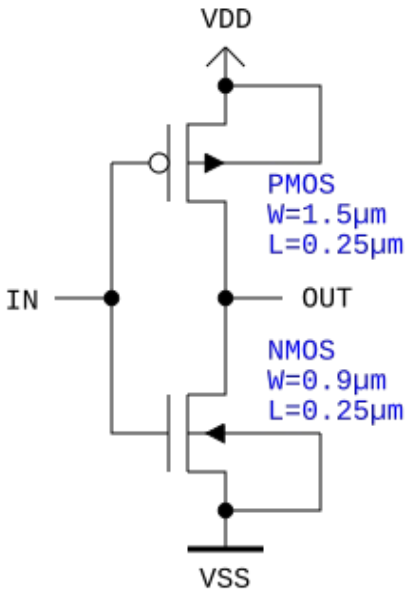
The inverter cell above is not useful by itself as it lacks features to tie the n well and the substrate to a defined potential. This is achieved with tie-down diodes.

Tie-down diodes are contacts over active regions. The active regions are implanted p+ on the substrate and n+ within the n well (the opposite implant type of transistors). With this doping profile, the metal contact won't form a Schottky barrier to the Silicon bulk and behave like an ohmic contact. So in fact, the "diode" isn't a real diode in the sense of a rectifier.

The modified layout is this one:



The corresponding schematic is this:



Inverter with
tie-down diodes

With this circuit, the n well is always at VDD potential and the substrate is tied at VSS:

```
* Simple CMOS inverer circuit
.SUBCKT INVERTER_WITH_DIODES VSS IN OUT VDD
Mp VDD IN OUT VDD PMOS W=1.5U L=0.25U
Mn OUT IN VSS VSS NMOS W=0.9U L=0.25U
.ENDS
```

The LVS script is slightly longer when extraction of tie-down diodes is included:

```
# LVS script (demo technology, KLayout manual)

# Preamble:

deep

# Reports generated:

report_lvs     # LVS report window

# Drawing layers:

nwell       = input(1, 0)
active      = input(2, 0)
pplus       = input(3, 0)
nplus       = input(4, 0)
poly        = input(5, 0)
```

```
contact     = input(6, 0)
metal1      = input(7, 0)
metal1_lbl  = labels(7, 1)
via1        = input(8, 0)
metal2      = input(9, 0)
metal2_lbl  = labels(9, 1)

# Bulk layer for terminal provisioning

bulk        = polygon_layer

# Computed layers

active_in_nwell       = active & nwell
pactive               = active_in_nwell & pplus
pgate                 = pactive & poly
psd                   = pactive - pgate
ntie                  = active_in_nwell & nplus

active_outside_nwell  = active - nwell
nactive               = active_outside_nwell & nplus
ngate                 = nactive & poly
nsd                   = nactive - ngate
ptie                  = active_outside_nwell & pplus

# Device extraction

# PMOS transistor device extraction
extract_devices(mos4("PMOS"), { "SD" => psd, "G" => pgate, "W" => nwell,
                                "tS" => psd, "tD" => psd, "tG" => poly, "tW" =>
nwell })

# NMOS transistor device extraction
extract_devices(mos4("NMOS"), { "SD" => nsd, "G" => ngate, "W" => bulk,
                                "tS" => nsd, "tD" => nsd, "tG" => poly, "tW" =>
bulk })

# Define connectivity for netlist extraction

# Inter-layer
connect(psd,        contact)
connect(nsd,        contact)
connect(poly,       contact)
connect(ntie,       contact)
connect(nwell,      ntie)
connect(ptie,       contact)
connect(contact,    metal1)
connect(metal1,     metal1_lbl)   # attaches labels
connect(metal1,     via1)
connect(via1,       metal2)
connect(metal2,     metal2_lbl)   # attaches labels

# Global
connect_global(bulk, "SUBSTRATE")
connect_global(ptie, "SUBSTRATE")

# Compare section

schematic("inv2.cir")
```

```
compare
```

The main difference is the computation of the regions for n tie-down (inside n well) and p tie-down. This is pretty straightforward:

```
ntie                  = active_in_nwell & nplus
ptie                  = active_outside_nwell & pplus
```

Device extraction does not change, but we need to include the tie-down regions into the connectivity:

```
connect(ntie,       contact)
connect(nwell,      ntie)
connect(ptie,       contact)
```

By connecting ntie to contact and nwell, we readily establish a connection to n well which behaves then like a conductive layer (although the resistance will be very high). Remember the the device extractors for PMOS will put the bulk terminals on nwell too, so the transistor is automatically connected to the nwell net.

ptie cannot be simply connected as there are no polygons for "substrate". But we can include ptie in the global connections:

```
connect_global(bulk, "SUBSTRATE")
connect_global(ptie, "SUBSTRATE")
```

nwell is no longer included in the global connections, hence we do no longer and incorrectly consider all nwell regions to be connected.

The extracted netlist shows the bulk terminals of NMOS and PMOS connected to source (drain and source are equivalent):

```
* Extracted by KLayout

* cell INVERTER_WITH_DIODES
.SUBCKT INVERTER_WITH_DIODES
* net 1 IN
* net 2 VDD
* net 3 OUT
* net 4 VSS
* device instance $1 r0 *1 1.025,4.95 PMOS
M$1 2 1 3 2 PMOS L=0.25U W=1.5U AS=0.675P AD=0.675P PS=3.9U PD=3.9U
* device instance $2 r0 *1 1.025,0.65 NMOS
M$2 4 1 3 4 NMOS L=0.25U W=0.9U AS=0.405P AD=0.405P PS=2.7U PD=2.7U
.ENDS INVERTER_WITH_DIODES
```

LVS Devices

Device extractors and device classes

KLayout provides two concepts for handling device variety:

Device classes are device categories. There are general categories such as resistors or MOS transistors. Specific categories can be created to represent specific incarnations - e.g. NMOS and PMOS devices. Device classes also determine how devices combine.

Device classes are documented here: LVS Device Classes.

Device extractors are the actual worker objects that analyze layout and produce devices. As for device classes, there are general device extractors. Each device extractor produces devices from a specific class.

Device extractors are documented here: LVS Devices Extractors.

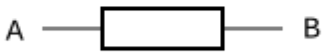
LVS Device Classes

- Resistor
- Resistor with bulk terminal
- Capacitor
- Capacitor with bulk terminal
- Diode
- MOS transistor
- MOS transistor with bulk
- Bipolar transistor
- Bipolar transistor with substrate

KLayout implements a variety of standard device classes. These device classes are the basis for forming particular incarnations of device classes. For example, the MOS4 class is the basis for the specific device classes for NMOS and PMOS transistors.

Resistor

```
DeviceClassResistor
```



The plain resistor has two terminals, A and B. It features the following parameters:

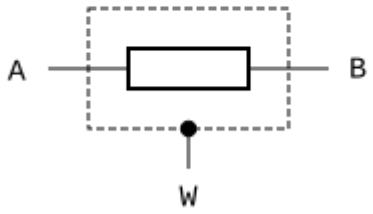
- **R**: The resistance value in Ohm
- **L**: The length in µm
- **W**: The width in µm
- **A**: The area of the resistor area in µm²
- **P**: The perimeter of the resistor area in µm

Resistors can combine in parallel or serial fashion.

In SPICE, plain resistors are represented by the "R" element. The API class is DeviceClassResistor.

Resistor with bulk terminal

```
DeviceClassResistorWithBulk
```



The resistor with bulk terminal is an extension of the plain resistor. If has the same parameters, but one additional terminal (W) which connects to the area the resistor sits in (e.g. well or substrate).

Resistors with bulk can combine in parallel or serial fashion if their bulk terminals are connected to the same net.

The API class of the resistor with bulk is DeviceClassResistorWithBulk.

Capacitor

```
DeviceClassCapacitor
```

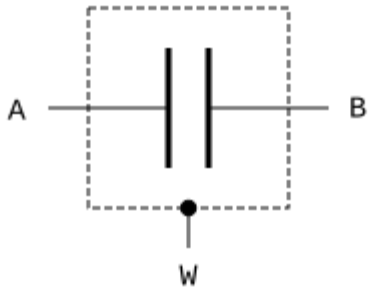
The plain capacitor has two terminals, A and B. It features the following parameters:

- **C**: The capacitance value in Farad
- **A**: The area of the capacitor area in µm²
- **P**: The perimeter of the capacitor area in µm

In SPICE, plain capacitors are represented by the "C" element. The API class is DeviceClassCapacitor.

Capacitor with bulk terminal

DeviceClassCapacitorWithBulk



The capacitor with bulk terminal is an extension of the plain capacitor. If has the same parameters, but one additional terminal (W) which connects to the area the capacitor sits in (e.g. well or substrate).

Capacitors with bulk can combine in parallel or serial fashion if their bulk terminals are connected to the same net.

The API class of the capacitor with bulk is DeviceClassCapacitorWithBulk.

Diode

DeviceClassDiode

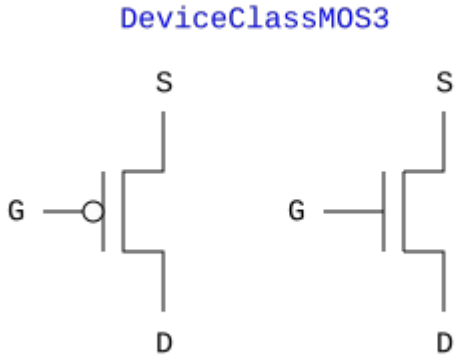
Diodes have two terminals, A and C for anode and cathode. Diodes feature the following parameters:

- **A**: The area of the diode in µm²
- **P**: The perimeter of the diode in µm

Diodes combine in parallel (A to A and C to C). In this case their areas and perimeters will add.

In SPICE, diodes are represented by the "D" element using the device class name as the model name. The API class is DeviceClassDiode.

MOS transistor



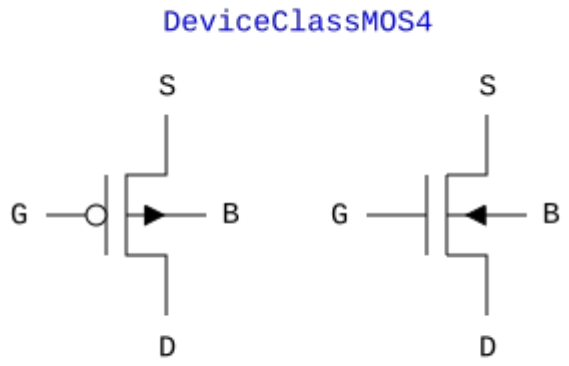
Three-terminal MOS transistors have terminals S, G and D for source, gate and drain. S and D are commutable. They feature the following parameters:

- **W**: The gate width in µm
- **L**: The gate (channel) length in µm
- **AS**: The source area in µm²
- **PS**: The source perimeter in µm
- **AD**: The drain area in µm²
- **PD**: The drain perimeter in µm

MOS3 transistors combine in parallel when the source/drains and gates are connected and their gate lengths are identical. In this case their widths, areas and perimeters will add.

MOS transistor with bulk

The API class of the three-terminal MOS transistor is DeviceClassMOS3.

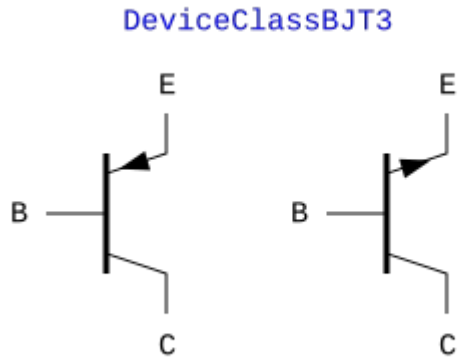


The four-terminal transistor is an extension of the three-terminal one and offers an additional bulk terminal (B). It is probably the most prominent transistor device as the four-terminal version is compatible with the SPICE "M" element.

MOS transistors with bulk can combine in parallel the same way the three-terminal versions do if their bulk terminals are connected to the same net.

In SPICE, MOS4 devices are represented by the "M" element with the device class name as the model name. The API class is DeviceClassDiode.

Bipolar transistor



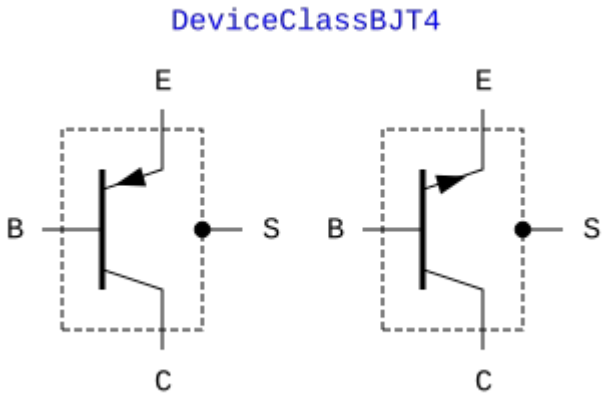
The three-terminal bipolar transistor can be either NPN or PNP type. In KLayout, this device type can represent both lateral and vertical types. The parameters are:

- **AE**: The emitter area in µm²
- **PE**: The emitter perimeter in µm
- **NE**: The emitter count (initially 1)
- **AB**: The base area in µm²
- **PB**: The base perimeter in µm
- **AC**: The collector area in µm²
- **PC**: The collector perimeter in µm

Upon extraction, multi-emitter versions are extracted as multiple devices - one for each emitter area - and NE = 1. Bipolar transistors combine when in parallel. In this case, their emitter parameters AE, PE and NE are added.

In SPICE, BJT3 devices are represented by the "Q" element with the device class name as the model name. The API class is DeviceClassBJT3.

Bipolar transistor with substrate



The four-terminal transistor is an extension of the three-terminal one and offers an additional bulk terminal (S).

Bipolar transistors with bulk can combine in parallel the same way the three-terminal versions do if their bulk terminals are connected to the same net.

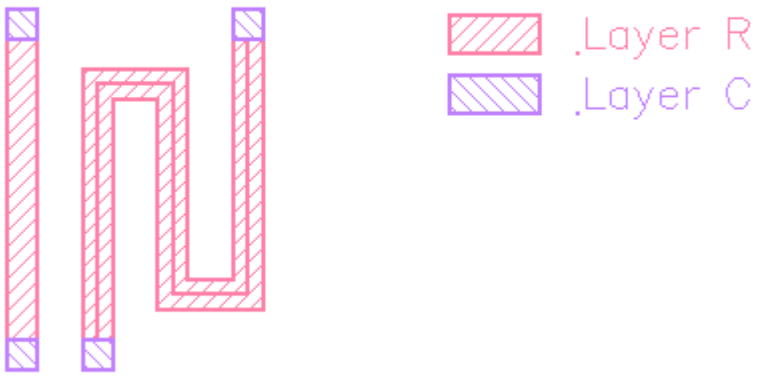
In SPICE, BJT4 devices are represented by the "Q" element with four nodes and the device class name as the model name. The API class is DeviceClassBJT4.

LVS Devices Extractors

Device extractors and the actual "workers" of the device extraction process. KLayout comes with a variety of pre-built device extractors. It's possible to implement custom device extractors in the framework of LVS scripts (speaking Ruby).

Resistor extractors (resistor and resistor_with_bulk)

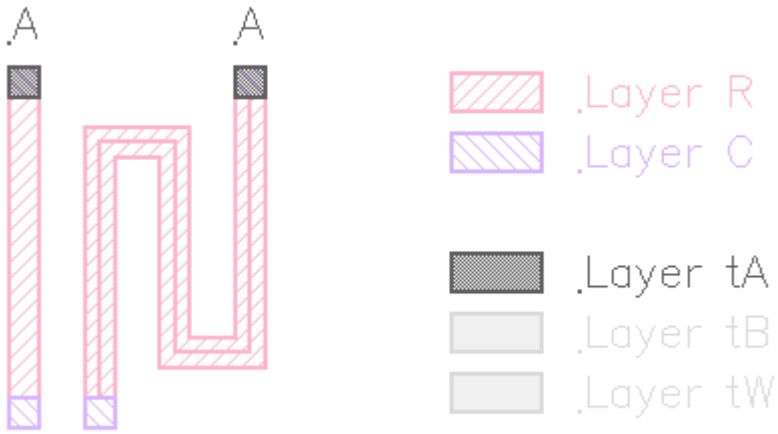
The resistor extractor assumes a layout which consists of a resistor "wire" and two caps (contacts). The wire is specified with the layer symbol "R", the caps are specified with the layer symbol "C":

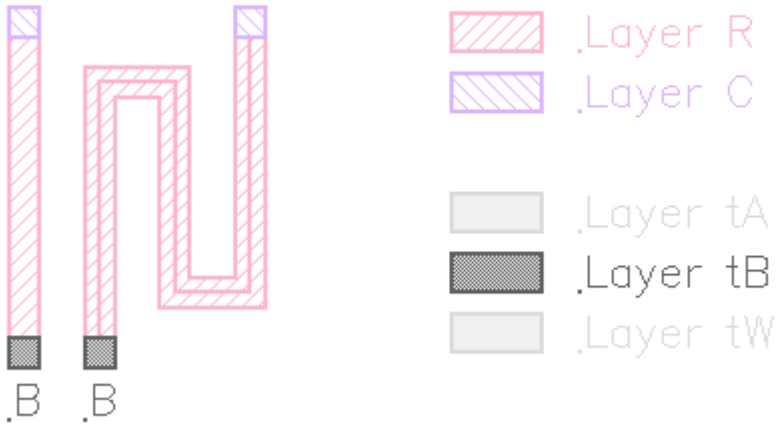


The extractor will compute the resistance from the number of squares and the sheet resistance. The sheet resistance needs to be given when creating the extractor:

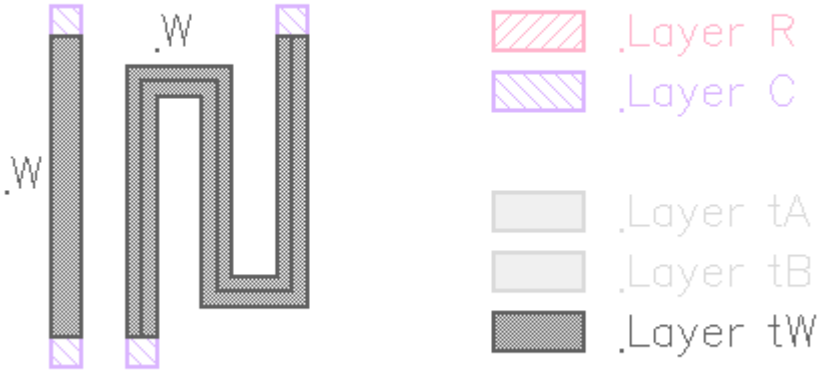
```
sheet_rho = 0.5
model_name = "RES"
extract_devices(resistor(model_name, sheet_rho), { "R" => res_layer, "C" =>
cap_layer })
```

The plain resistor offers two terminals which it outputs on "tA" and "tB" terminal layers. If "tA" or "tB" is not specified, "A" or "B" terminals will be written on the "C" layer. respectively.



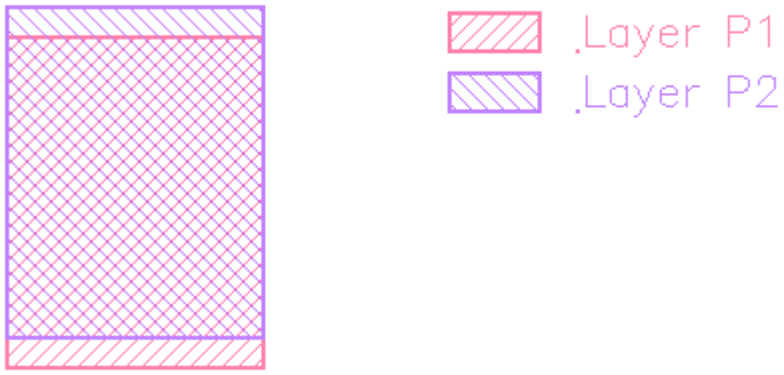


For the resistor with bulk, the wire area is output on the "tW" terminal layer as the "W" terminal:



Capacitor extractors (capacitor and capacitor_with_bulk)

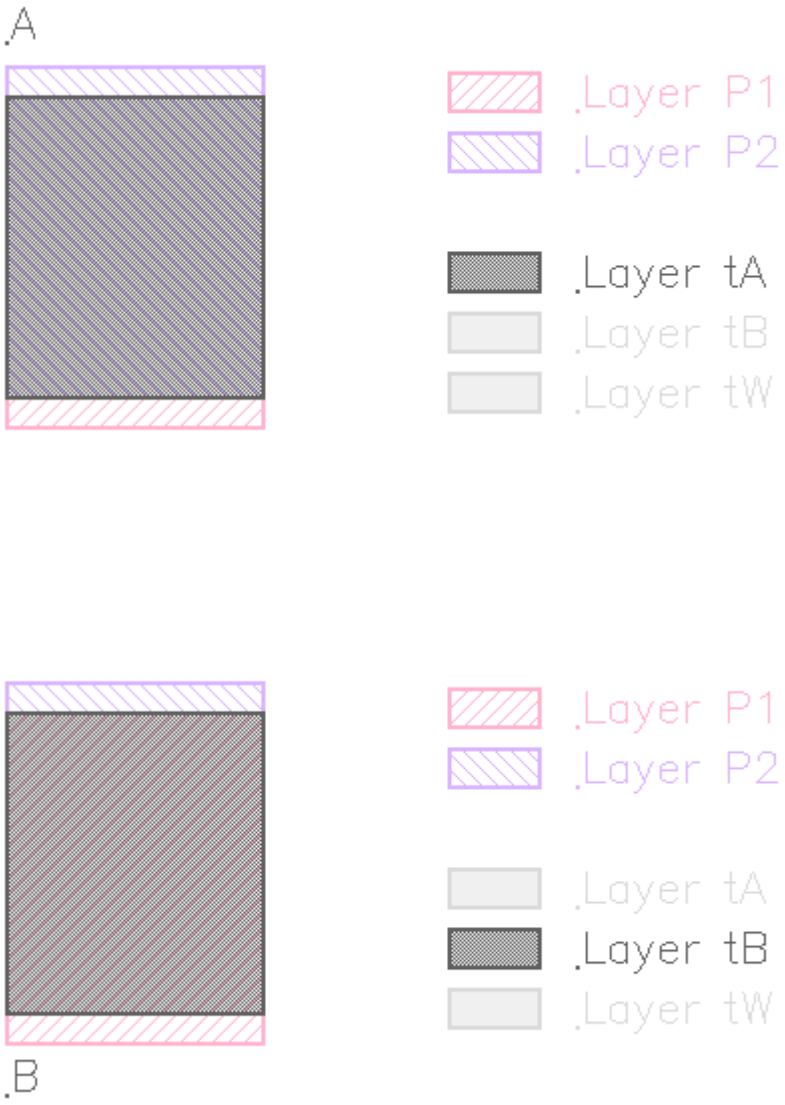
Capacitors are assumed to consist of two "plates" (vertical capacitors). The plates are on layers P1 and P2. The capacitor is extracted from the area where these two layers overlap.



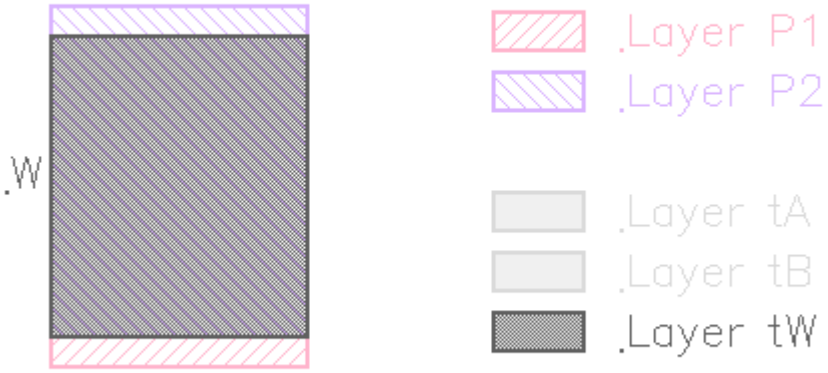
The extractor will compute the capacitance from the area of the overlap and the capacitance per area (F/µm²) value.

```
area_cap = 1.5e-15
model_name = "CAP"
extract_devices(capacitor(model_name, area_cap), { "P1" => metal1, "P2" =>
metal2 })
```

The plain capacitor offers two terminals which it outputs on "tA" and "tB" terminal layers. If "tA" or "tB" is not specified, "A" or "B" terminals will be written on the "P1" and "P2" layers respectively.

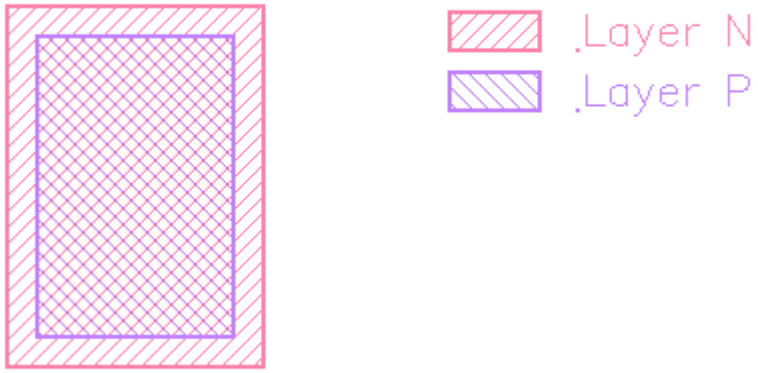


For the capacitor with bulk, the capacitor area is output on the "tW" terminal layer as the "W" terminal:



Diode extractor (diode)

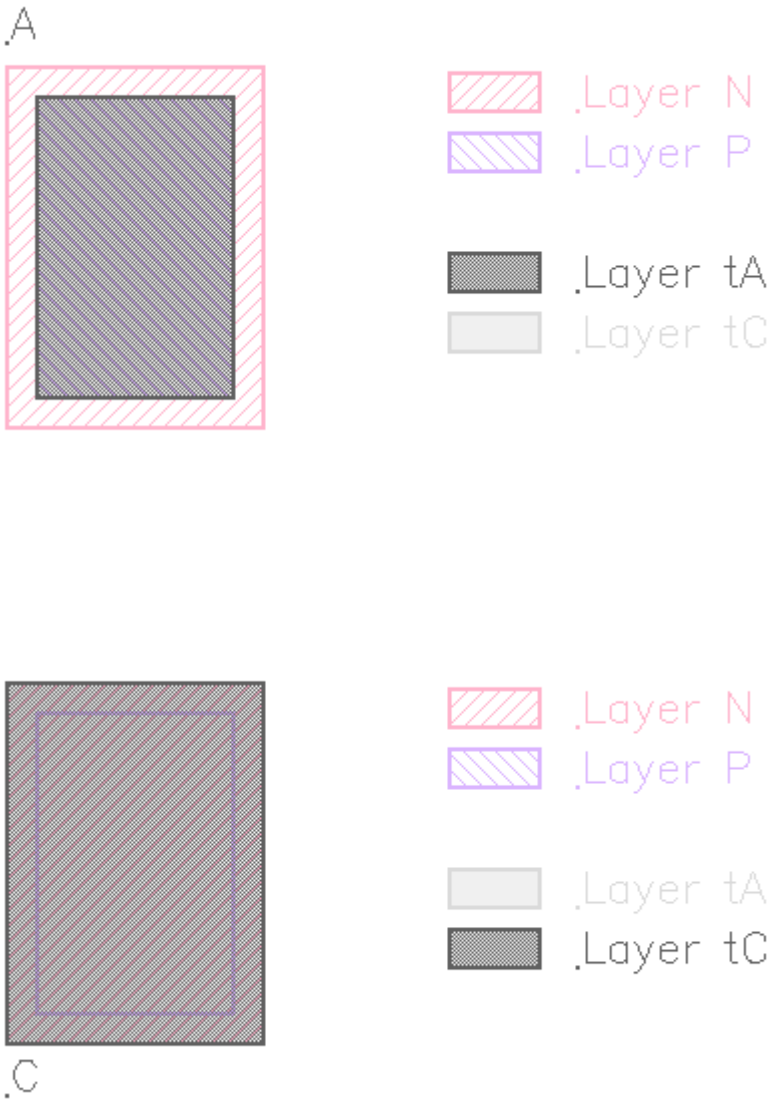
Diodes are assumed to consist of two vertical implant regions (wells, diffusion). One of the regions is p type ("P" layer) and the other "n" type ("N" layer). These layers also form the anode (p) and cathode (n) of the diode.



The extractor will compute the capacitance from the area of the overlap and the capacitance per area (F/µm²) value.

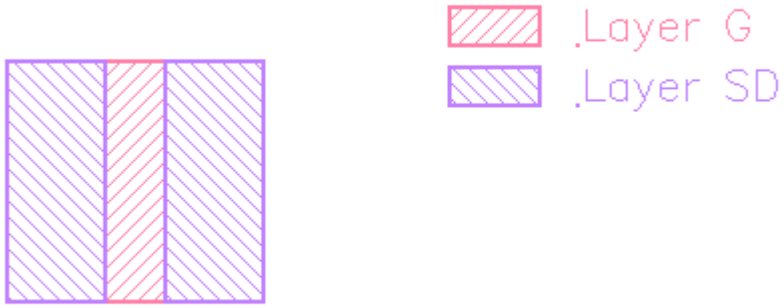
```
model_name = "DIODE"
extract_devices(diode(model_name), { "P" => pplus, "N" => nwell })
```

The diode offers two terminals which it outputs on "tA" and "tC" terminal layers. If "tA" is not specified, "A" terminals will be written on the "P" layer. If "tC" is not specified, "C" terminals will be written on the "N" layer.



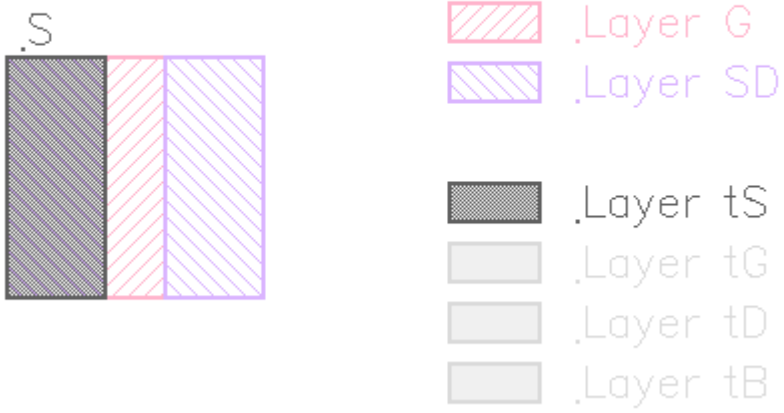
MOS transistor extractor (mos3 and mos4)

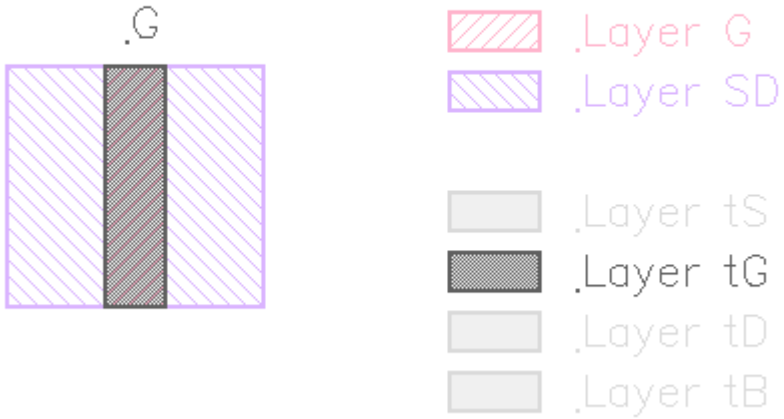
MOS transistors are recognized by their gate ("G" input) and source/drain ("SD" input) regions. Source and drain needs to be separated from the gate shape. The touching edges of gate and source/drain regions define the width of the device, the perpendicular dimension the gate length. Because the separation of source/drain, the computation of gates and the separation of these for NMOS and PMOS devices, the "G" and "SD" layers are usually derived layers. As these usually won't participate in the connectivity, it's important to specify the "tS", "tD", "tG" and "tB" (for MOS4) layers explicitly and redirect the terminal shapes to layers that really participate in connections.

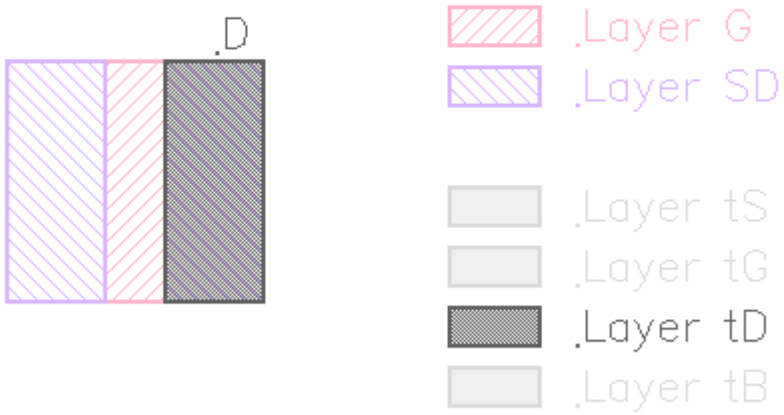



```
model_name = "PMOS"
extract_devices(mos4(model_name), { "SD" => (active - poly) & pplus, "G" => (active
& poly), "W" => nwell,
                                    "tS" => active, "tD" => active, "tG" => poly,
"tB" => nwell })
```

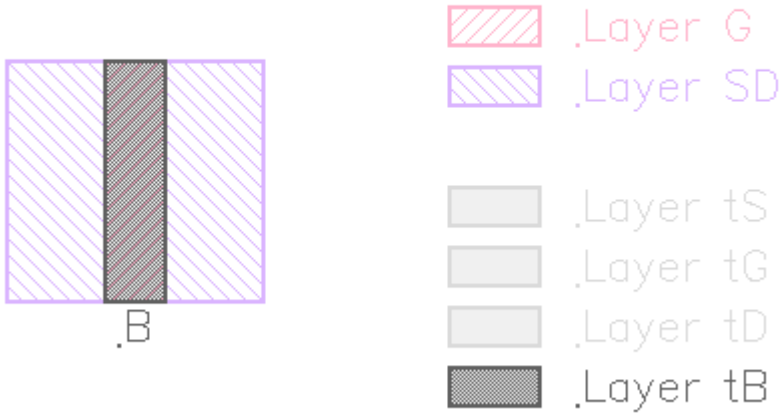
The MOS3 device produces three terminals which it outputs on "tS", "tG" and "tD" terminal layers (source, gate and drain respectively):







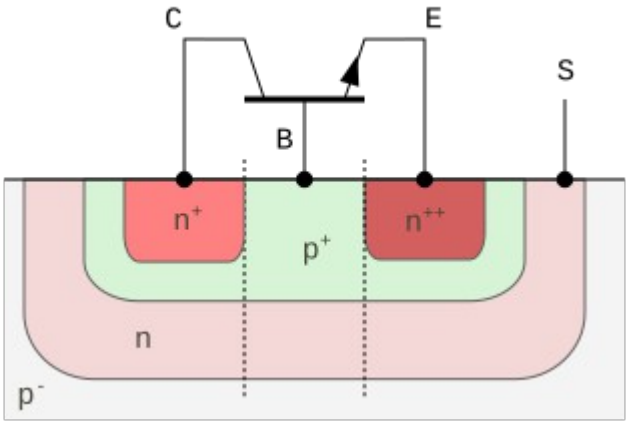
The MOS4 device offers one more terminal (bulk) which it writes on "tB".



Bipolar transistor extractor (bjt3 and bjt4)

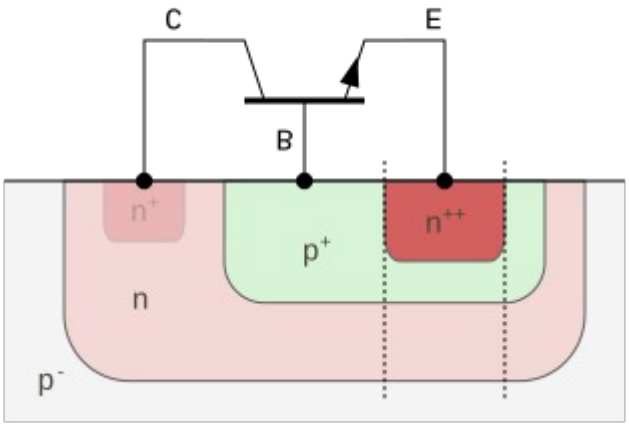
There are basically two kind of bipolar transistors: vertical and lateral ones.

Lateral transistors are formed by implant or diffusion wells creating a intermittent n/p structure on the wafer. The basic recognition region is the base region. The collector and emitter regions are inside or overlapping the base region and use the opposite doping than base: if the base region is n doped, then collector and emitter regions have to be p doped. The structure then forms a PNP transistor. KLayout recognizes lateral transistors when the base is **partially** covered by the collector region. For lateral transistors, the emitter is defined by the emitter region inside base. The collector region is defined by collector inside base and outside emitter.



(lateral NPN transistor)

Vertical transistors are formed by a stack of n/p wells. Sometimes vertical transistors are formed as parasitic devices in standard CMOS processes. A PNP transistor can be formed by taking the collector as the substrate, nwell for the base and pplus implant for the emitter. KLayout recognizes a vertical bipolar transistor when the base is covered **entirely** by the collector or has **no collector at all** - this means the collector region can be empty (e.g. bulk).

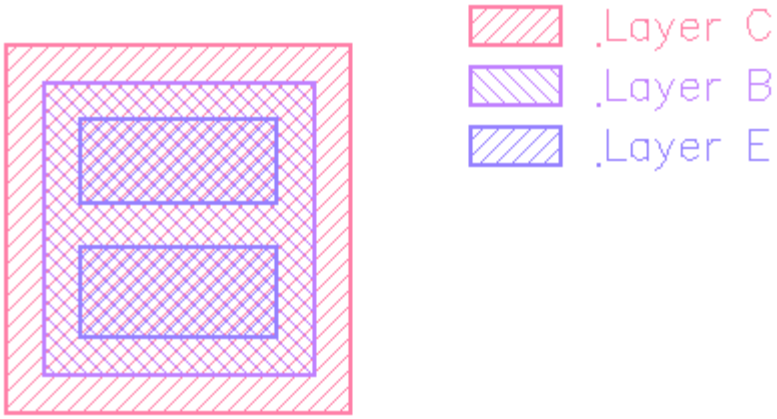


(vertical NPN transistor)

In both cases, there can be multiple emitter regions inside a base island. In this case, one transistor is extracted for each emitter region.

Vertical bipolar transistors

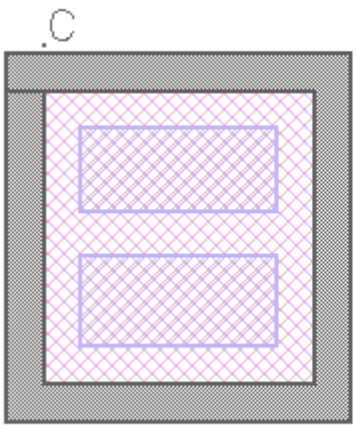
Vertical bipolar transistors take their inputs from "B" (base), "C" (collector) and "E" (emitter). "C" is optional:



Especially for bipolar devices it's important to device useful terminal output layers. Typically, the wells and diffusion areas will be connected through "contact", (not considering the Schottky diodes for now). So it's a good idea to send the terminals to the contact layer:

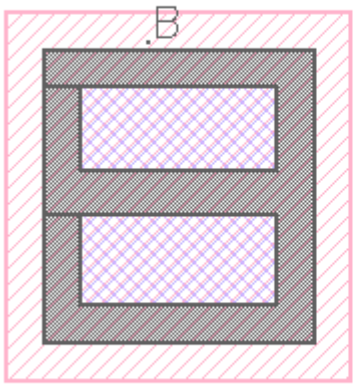
```
model_name = "PNP"
extract_devices(bjt3(model_name), { "C" => collector, "B" => base, "E" => emitter,
                                    "tC" => contact, "tB" => contact, "tE" =>
contact })
```

The BJT3 device produces three terminals which it outputs on "tC", "tB" and "tE" terminal layers (collector, base and emitter respectively):



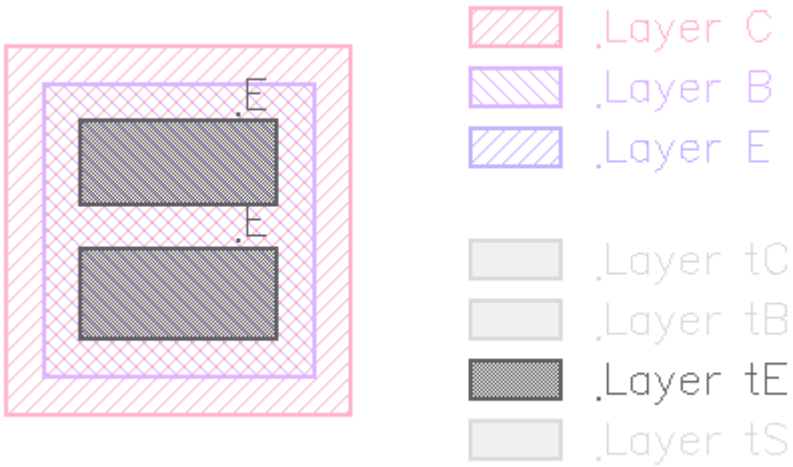
.Layer C
.Layer B
.Layer E

.Layer tC
.Layer tB
.Layer tE
.Layer tS



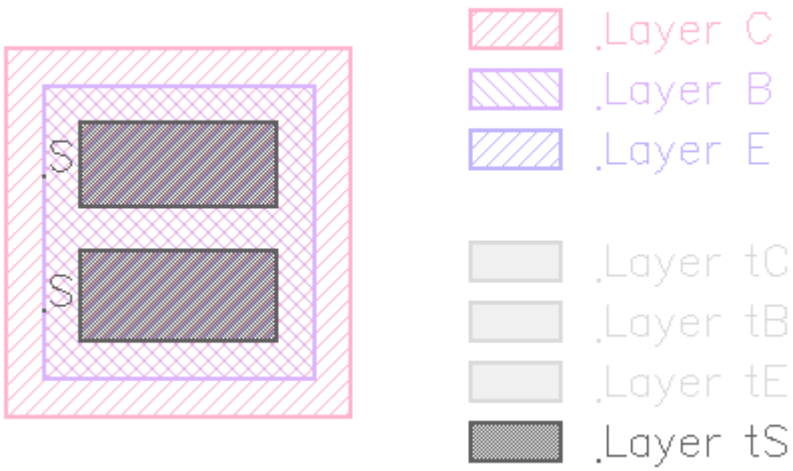
.Layer C
.Layer B
.Layer E

.Layer tC
.Layer tB
.Layer tE
.Layer tS



If the collector region is empty (e.g. p substrate), the base shape is copied to the "tC" output layer for the collector terminal.

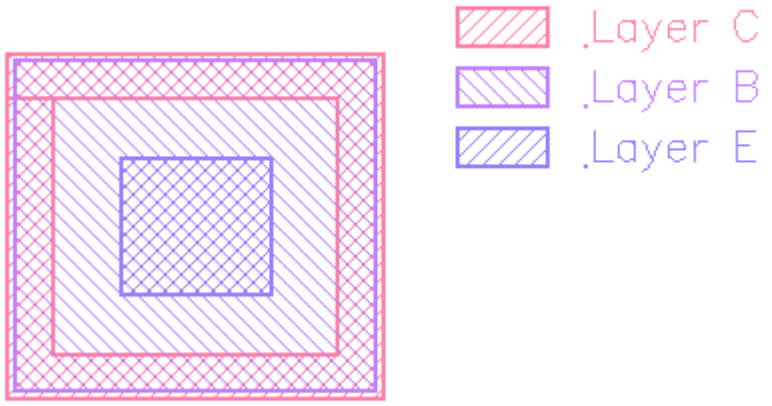
The BJT4 device offers one more terminal (substrate) which it writes on "tS". "tS" is a copy of the emitter shape but connected to the substrate terminal:



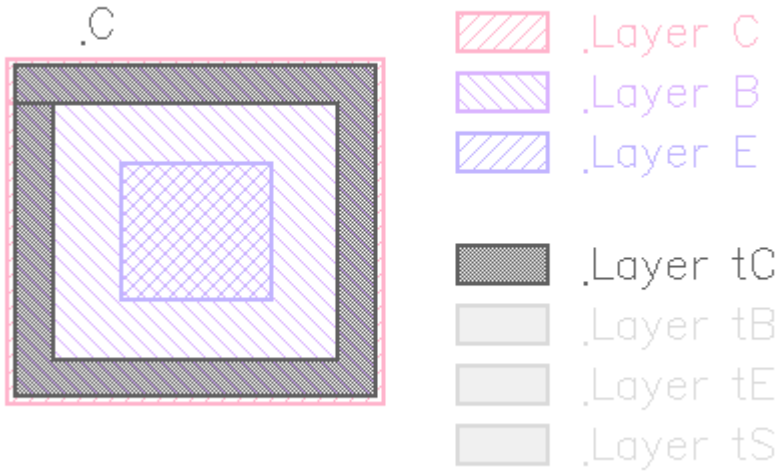
Lateral bipolar transistors

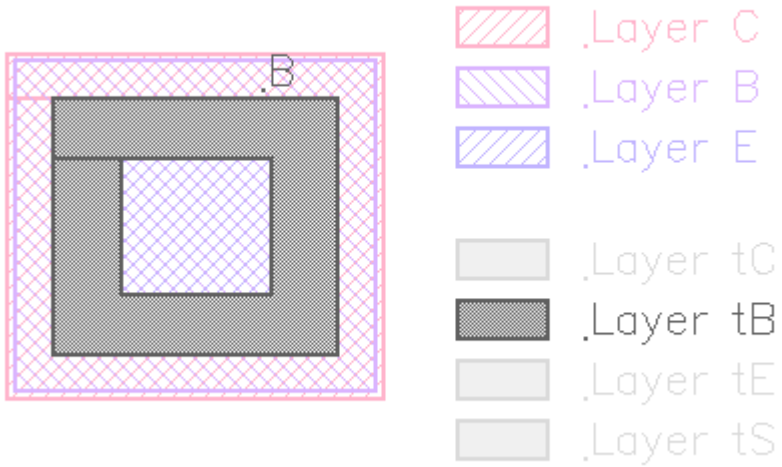
Lateral bipolar transistors also take their inputs from "B" (base), "C" (collector) and "E" (emitter). For lateral transistors, "C" is not optional and must not fully cover the base region. Apart from this, the use model for BJT3 and BJT4 extractors is identical for vertical and lateral transistors.

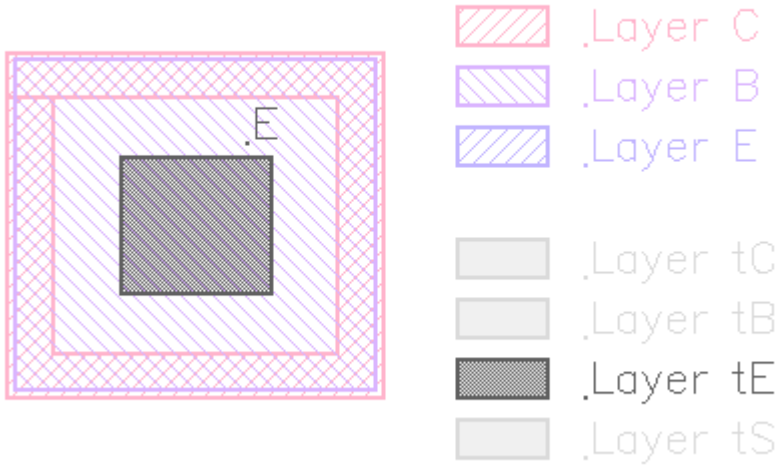
A typical lateral transistor is formed by a collector ring and emitter island inside the base region:



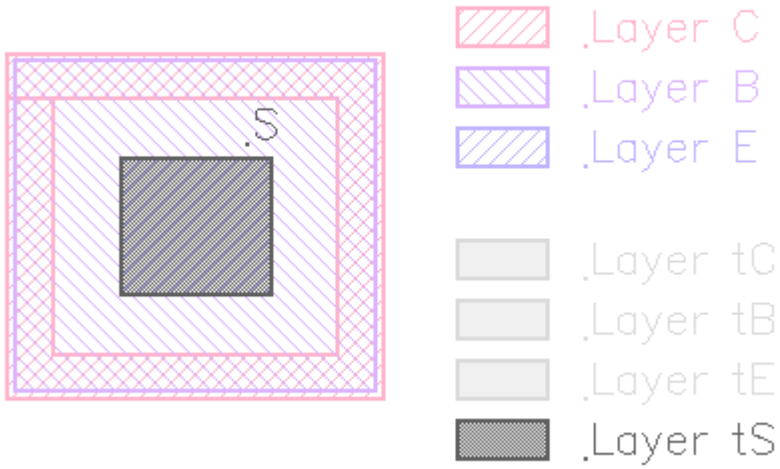
The terminals produced by the bipolar transistor extractor in the lateral case are the same than for the vertical case, but with a different geometry:







Again, for BJT4, "tS" is a copy of the emitter shape but connected to the substrate terminal:


.S
.Layer C
.Layer B
.Layer E
.Layer tC
.Layer tB
.Layer tE
.Layer tS

LVS Input/Output

LVS (and also DRC as far as netlist extraction is concerned) provides interfaces to write and read netlists/schematics, annotated layout and LVS results. There are three major categories of I/O:

- **Netlist**: this is the plain circuit information. With subcircuit this forms a hierarchical netlist. Currently, the format available to import and export netlists is a certain SPICE netlist flavor. It's possible to customize the reading and writing process to achieve some flexibility.
- **Layout-to-netlist database (L2N DB)**: also called extracted netlist or annotated layout. This is the netlist taken from the original layout together with the corresponding shapes. This database allows reconstructing a net geometrically as far as non-device shapes are involved. Devices are abstracted by their terminal geometries.
- **LVS result database (LVS DB)**: this is the L2N database plus the reference netlist and a "cross reference": a list of paired circuits, nets, devices, pins and subcircuits and status information. The cross-reference is both a lookup table and a debugging aid.

Writing netlists

You can write a netlist file to supply netlists for (functional) simulators for example. Within LVS scripts, the global "target_netlist" statement triggers writing of a netlist (see target_netlist for details).

```
target_netlist("output.cir", write_spice, "Created by KLayout")
```

This statement can basically appear anywhere in the LVS script. The netlist will written after the script has executed successfully. The first argument is the file's path (by default relative to the original layout file). The second argument is the "writer". "write_spice" creates a netlist writer writing SPICE format with a limited degree of flexbility. See below for customizing the writer. The third argument finally is an (optional) comment which will be written into the netlist as a header.

The "write_spice" configuration function has two options:

```
write_spice(use_net_names, with_comments)
```

Both options are boolean values. If true and present, the first option will make the writer use the real net's names instead of numerical IDs. If true and present, "with_comments" will embed debug comments into the netlist showing instance locations, pin names etc.

Further customization can be achieved by providing an explicit SPICE writer with a delegate (see NetlistSpiceWriterDelegate). For doing so, subclass NetlistSpiceWriterDelegate and reimplement one or several of the methods provided for reimplementation. Those are NetlistSpiceWriterDelegate#write_device, NetlistSpiceWriterDelegate#write_device_intro and NetlistSpiceWriterDelegate#write_header.

Here is an example that supplied subcircuit models rather than device elements:

```
# Write extracted netlist to extracted.cir using a special
# writer delegate

# This delegate makes the writer emit subcicuit calls instead of
# standard elements for the devices
```

```
class SubcircuitModels < RBA::NetlistSpiceWriterDelegate

  def write_header
    emit_line(".INCLUDE 'models.cir'")
  end

  def write_device(device)
    str = "X" + device.expanded_name
    device_class = device.device_class
    device_class.terminal_definitions.each do |td|
      str += " " + net_to_string(device.net_for_terminal(td.id))
    end
    str += " " + device_class.name
    str += " PARAMS:"
    device_class.parameter_definitions.each do |pd|
      str += " " + pd.name + ("=%.12g" % device.parameter(pd.id))
    end
    emit_line(str)
  end

end

# Prepare a writer using the new delegate
custom_spice_writer = RBA::NetlistSpiceWriter::new(SubcircuitModels::new)
custom_spice_writer.use_net_names= true
custom_spice_writer.with_comments = false

# The declaration of netlist production using the new custom writer
target_netlist("extracted.cir", custom_spice_writer, "Extracted by KLayout")
```

This script will produce the following netlist for the simple inverter from the LVS introduction. Instead of printing "M" elements - which is the default - subcircuit calls are produced. This allows putting more elaborate models into subcircuits. The device class name addresses these model subcircuits:

```
* Extracted by KLayout
.INCLUDE 'models.cir'

.SUBCKT INVERTER
X$1 VDD IN OUT NWELL PMOS PARAMS: L=0.25 W=1.5 AS=0.675 AD=0.675 PS=3.9 PD=3.9
X$2 VSS IN OUT SUBSTRATE NMOS PARAMS: L=0.25 W=0.9 AS=0.405 AD=0.405 PS=2.7
+ PD=2.7
.ENDS INVERTER
```

Netlists can be written directly from the netlist object. Within the script, the netlist object can be obtained with the netlist function. This function will first trigger a netlist extraction unless this was done already and return a Netlist object. Use Netlist#write to write this netlist object then. Unlike "target_netlist", this method is executed immediately and this way, a single netlist can be written to multiple files in different flavours.

Reading netlists

The main use case for reading netlists is for comparison in LVS. Reference netlists are read with the "schematic" function (see schematic):

```
schematic("inverter.cir")
```

Currently SPICE is understood with some limitations:

- Parametrized circuits are not permitted except for device subcircuits (with a delegate)
- Only a subset of elements is implemented by default. These are "M" (gives "MOS4" device classes), "Q" (gives BJT3 or BJT4 device classes), "R" (gives Resistor device classes), "C" (gives Capacitor device classes) and "D" (gives diode device classes).

As for the SPICE reader, a delegate can be provided to customize the reader. For doing so, subclass the NetlistSpiceReaderDelegate class and reimplement the methods provided. These are: NetlistSpiceReaderDelegate#wants_subcircuit, NetlistSpiceReaderDelegate#element, NetlistSpiceReaderDelegate#finish and NetlistSpiceReaderDelegate#start

This example customizes a reader to pull MOS devices from subcircuit models rather than from "M" elements. Basically this customization does the opposite part of the writer customization before (only for MOS devices).

```
# Provides a SPICE netlist reader delegate which turns
# some subcircuit models (for subcircuits NMOS and PMOS)
# into devices

class SubcircuitModelsReader < RBA::NetlistSpiceReaderDelegate

  # says we want to catch these subcircuits as devices
  def wants_subcircuit(name)
    name == "NMOS" || name == "PMOS"
  end

  # translate the element
  def element(circuit, el, name, model, value, nets, params)

    if el != "X"
      # all other elements are left to the standard implementation
      return super
    end

    if nets.size != 4
      error("Subcircuit #{model} needs four nodes")
    end

    # provide a device class
    cls = circuit.netlist.device_class_by_name(model)
    if ! cls
      cls = RBA::DeviceClassMOS4Transistor::new
      cls.name = model
      circuit.netlist.add(cls)
    end

    # create a device
    device = circuit.create_device(cls, name)

    # and configure the device
    [ "S", "G", "D", "B" ].each_with_index do |t,index|
      device.connect_terminal(t, nets[index])
    end
    params.each do |p,value|
      device.set_parameter(p, value)
    end

  end
```

```
end

# Instantiate a reader using the new delegate
reader = RBA::NetlistSpiceReader::new(SubcircuitModelsReader::new)

# Import the schematic with this reader
schematic("inv_xmodels.cir", reader)
```

Layout-to-Netlist database/report

The layout-to-netlist database (L2N DB) is written using the global report_netlist function. This function can be put anywhere in the script. Writing will happen after the script executed successfully:

```
report_netlist("extracted.l2n")
```

Without the filename, only the netlist browser will be opened but no file will be written. The layout-to-netlist database is a KLayout-specific format. It contains the netlist information plus the shape and instance information from the layout. L2N databases can be read into the netlist browser for example. Hence exchange of extracted netlists is possible.

Layout-vs-Schematic database/report

The Layout-vs-schematic database (LVS DB) is written using the global report_lvs function. This function can be put anywhere in the script. Writing will happen after the script executed successfully:

```
report_lvs("extracted.lvsdb")
```

Without the filename, only the netlist browser will be opened but no file will be written. The LVS database is a KLayout-specific format. It contains the extracted netlist information, the reference netlist and the cross-reference table. LVS databases can be read into the netlist browser for example. Hence exchange of LVS reports is possible.

LVS Connectivity

Intra- and inter-layer connections

The connectivity setup of a LVS script determines how the connections are made. Connections are usually made through conductive materials such as Aluminium or Copper. The polygons representing such a material form a connection. Connections can be made across multiple polygons - touching polygons form connected islands of conductive material. This "intra-layer" connectivity is implicit: in LVS scripts connections are always made between polygons on the same layer.

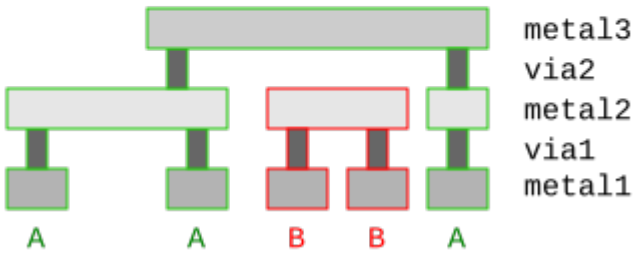
Connections often cross layers. A via for example is a hole in the insulator sheet which connects two metal layers. This connection is modelled using a "connect" statement (see connect):

```
connect(layer1, layer2)
```

A connect statement will specify an electrical connection when the polygons from layer1 and layer2 overlap. layer1 and layer2 are original or derived layers. "connect" statements should appear in the script before the netlist is required - i.e. before "compare" or any other netlist-related statement inside the LVS script. The order of the connect statements is not relevant. Neigther is the order of the arguments in "connect": connections are always bidirectional.

This is an example for a vertical cross section through a simple 3-metal layer stack with the corresponding "connect" statements:

```
connect(metal1, via1)
connect(via1, metal2)
connect(metal2, via2)
connect(via2, metal3)
```



Labels can be included in the connectivity too. Typically labels are placed on metal layers. If the labels are drawn on the same layer than the metal shapes they are automatically included when using "input" to read the layer. If only labels shall be read from a layer, use "labels" (see labels).

To attach labels to metal layers, simply connect the label and metal layers:

```
metal1_labels = labels(10, 0)
metal1        = input(11, 0)
via1          = input(12, 0)
metal2_labels = labels(13, 0)
metal2        = input(14, 0)

connect(metal1, metal1_labels)
connect(metal1, via1)
connect(via1, metal2)
connect(metal2, metal2_labels)
```

If labels are connected to metal layers, their text strings will be used to assign net names to the resulting nets. Ideally, one net is labelled with a single text or with texts with the same text string. In this case, the net name will be non-ambiguous. If multiple labels with different strings are present on a net, the net name will be made from a combination of these names.

Global connections

KLayout supports implicit connections made across all polygons on a layer, regardless whether they connect or not. A typical case for such a connection is the substrate (aka "bulk"). This connection represents the (lightly conductive) substrate material. There is no polygon representing the wafer. Instead, a layer is defined which makes a global connection with "connect_global" (see connect_global):

```
connect_global(bulk, "VSS")
```

The arguments to "connect_global" is the globally connected layer and the name of the global net to create. The function will make all shapes on "bulk" being connected to a single net "VSS". Every circuit will at least have the "VSS" net. In addition, each circuit will be given a pin called "VSS" which propagates this net to parent circuits.

Implicit connections

Implicit connections can be useful to supply preliminary connections which are supposed to be created higher up in the hierarchy: Imagine a circuit which a big power net for example. When the layout is made, the power net may not be completely connected yet because the plan is to connect all parts of this power net later when the cell is integrated. In this situation, the subcircuit cell itself won't be LVS clean because the power net is a single net schematic-wise, but exist as multiple nets layout-wise. This prevents bottom-up verification - a very useful technique to achieve LVS clean layouts.

To allow verification of such a cell, "implicit connections" can be made by giving the net parts the same name through labels and assume these parts are connected: for example to specify implicit connections between all parts of a "VDD" net, place a label "VDD" on each part and include the following statement in the script:

```
connect_implicit("VDD")
```

"connect_implicit" (see connect_implicit) can be present multiple times to make many of such connections. Implicit connections will only be made on the topmost circuit to prevent false verification results. Be careful not to use this option in a final verification of a full design as power net opens may pass unnoticed.

LVS Compare

The actual compare step is rather simple. Provided you have set up the extraction (extract_devices), the connectivity (connect, connect_global, connect_implicit) and provided a reference netlist (schematic), this function will perform the actual compare:

```
compare
```

This method (compare will extract the netlist (if not already done) and compare it against the schematic. It returns true on success and false otherwise, in case you like to take specific actions on success or failure.

The compare step can configured by providing hints.

Net equivalence hint

It can be useful to declare two nets as identical, at least for debugging. The compare algorithm will then be able to deduce the real causes for mismatches. It is helpful for example to provide equivalence for the power nets, because netlist compare fails will often cause the power nets no to be mapped. This in turn prevents matching of other, good parts of the circuit. To supply a power net equivalence for "VDD" within a circuit (e.g. "LOGIC"), use this statement:

```
same_nets("LOGIC", "VDD", "VDD:P")
```

In this example it is assumed that the power net is labelled "VDD" in the layout and called "VDD:P" in the schematic. Don't leave this statement in the script for final verification as it may mask real errors.

For more information about "same_nets" see same_nets.

Circuit equivalence hint

By default, circuits with the same name are considered equivalent. If this is not the case, equivalence can be established using the same_circuit function:

```
same_circuits("CIRCUIT_IN_LAYOUT", "CIRCUIT_IN_SCHEMATIC")
```

Declaring circuits as 'same' means they will still be compared. The function is just a hint where to look for the compare target.

Device class equivalence hint

By default, device classes with the same name are considered equivalent. If this is not the case, equivalence can be established using the same_device_classes function:

```
same_device_classes("PMOS_IN_LAYOUT", "PMOS_IN_SCHEMATIC")
same_device_classes("NMOS_IN_LAYOUT", "NMOS_IN_SCHEMATIC")
```

Pin swapping

Pin swapping can be useful in cases, where a logic element has logically equivalent, but physically different inputs. This is the case for example for a CMOS NAND gate where the logic inputs are equivalent in function, but not in the circuit and physical implementation. For such circuits, the compare function needs to be given a degree of freedom and be allowed to swap the inputs. This is

achieved with the equivalent_pins function:

```
equivalent_pins("NAND_GATE", "A", "B")
```

The first argument is the name of the circuit in the layout netlist. You can only specify equivalence in layout, not in the reference schematic. Multiple pins can be listed after the circuit name. All of them will be considered equivalent.

Capacitor and resistor elimination

This feature allows eliminating "open" resistors and capacitors. Serial resistors cannot be elimiated currently (shorted).

To eliminate all resistors with a resistance value above a certain threshold, use the max_res function. This will eliminate all resistors with a value >= 1kOhm:

```
max_res(1000)
```

To eliminate all capacitors with a capacitance value below a certain threshold, use the max_caps function. This will eliminate all capacitances with a value <= 0.1fF:

```
max_caps(1e-16)
```

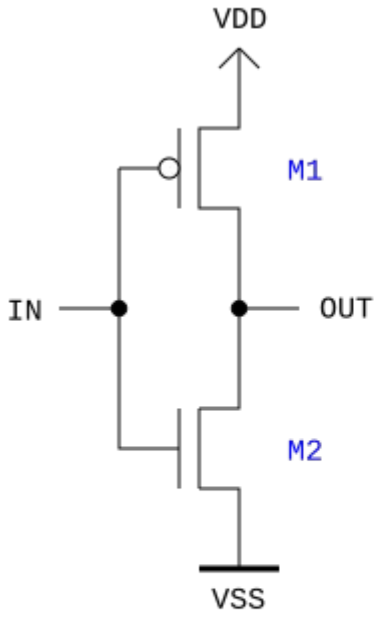
How the compare algorithm works

The coarse flow of the netlist compare algorithm is this:

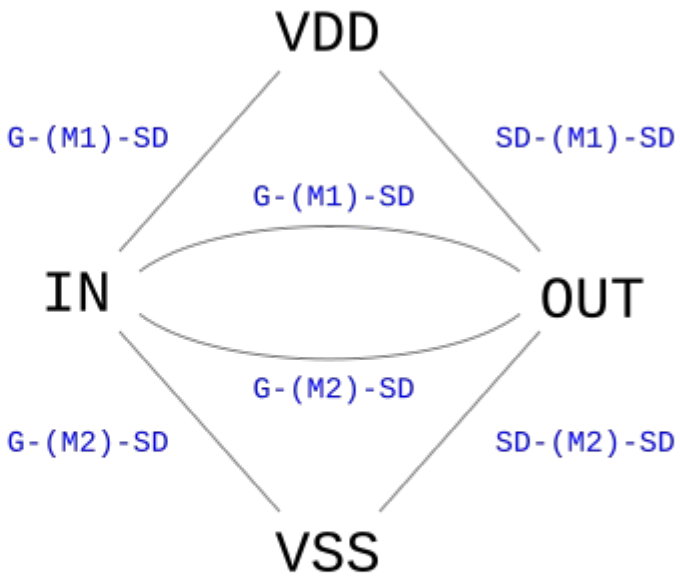
```
foreach circuit bottom up:
  if matching circuit found in reference netlist:
    if all subcircuits have been matched and pin matching has been established for
them:
      compare net graph locally from this circuit
    else:
      skip circuit with warning
  else:
    issue a circuit mismatch error
```

A consequence of this flow is that the compare will stop treating parent circuits when one circuit's pins can't be matched to pins from the corresponding reference circuit or the corresponding circuit can't be found in the reference netlist. This behaviour fosters a bottom-up debugging approach: first fix the issues in subcircuits, then proceed to the parent circuits.

The local net graph compare algorithm is a backtracking algorithm with hinting through topological net classification. Topological net classification is based on nearest-net neighborhood. The following image illustrates this:



Netlist



Net Neighborhood Graph

Here the IN net's neighborhood is VDD via a traversal of gate to source/drain over M1, to OUT via a twofold traversal of gate to source/drain over M1 and M2 and to VSS via another single traversal of gate to source/drain over M2. This uniquely identifies IN in this simple circuit. In effect, OUT, VDD and VSS can be identified uniquely because their transitions from the IN net are unambigously identifying them. The topological neighborhood is a simple metrics which allows identifying matching nets from two netlists and deducing further relations.

In big netlists, the algorithm will first try to match nets unambigously according to their neighborhood metrics and register them as paired nets. Such pairs often allow deducing further matching pairs. This deduction is continued until all non-ambiguous pairing options are exhausted. For resolving ambiguities, backtracking is employed: the algorithm proposes a match and tentatively proceeds with this assumption. If this execution path leads to a mismatch or logical contradiction, the algorith will go back to the beginning and restart with a new proposal. Backtracking is usually required mainly to match networks with a high symmetry such as clock trees.

LVS Netlist Tweaks

Netlist tweaking is important to standardize netlists. Without tweaking, the extracted netlist may contain elements that are redundant or don't match anything found in the schematic.

Netlist tweaks are applied on the extracted Netlist object. This can be obtained with the netlist function. This function will extract the netlist if not done already.

Netlist tweaks can also be applied to the schematic netlist. For example to flatten away a model subcircuit called "NMOS", use this Netlist#flatten_circuit:

```
schematic.flatten_circuit("NMOS")
```

Top level pin generation

Circuits extracted don't have pins on the top hierarchy level as the extractor cannot figure out where to connect to this circuit. The compare function does not try to match pins in this case. But to gain a useful extracted netlists, pins are required. Without pins, a circuit can't be embedded in a testbench for example.

KLayout offers a function to create top-level pins using a simple heuristics: for every named (i.e. labelled) net in the top level circuit a pin will be created (Netlist#make_top_level_pins):

```
netlist.make_top_level_pins
```

Device combination

Combining devices is important for devices which are not represented as coherent entities in the layout. Examples are:

- **Fingered MOS transistors:** MOS transistors with a large width are often split into multiple pieces to reduce the parasitic gate and diffusion resistances and capacitances. In the layout this is equivalent to multiple parallel transistors.
- **Serial resistors:** Large resistors are often separated into stripes which are then connected in a meander structure. From the device perspective such resistors consist of several resistors connected in series.
- **Array capacitors:** Large capacitors are often split into smaller ones which are arranged in an array and connected in parallel. This helps controlling the parasitic series resistances and inductances and avoids manufacturing issues.

In all these cases, the schematic usually summarizes these devices into a single one with lumped parameter values (total resistance, capacitance, transistor width). This creates a discrepancy which "device combination" avoids. "Device combination" is a step in which devices are identified which can be combined into single devices (such as serial or parallel resistors and capacitors). To run device combination, use Netlist#combine_devices:

```
netlist.combine_devices
```

The combination of serial devices might leave floating nets (the net connecting the devices originally. These nets can be removed with Netlist#purge_nets. See also Netlist#simplify, which is wrapper for several methods related to netlist normalization.

It's recommended to run "make_toplevel_pins" and "purge" before this step (see below).

Circuit flattening (elimination)

It's often required to flatten circuits that do not represent a specific level of organisation but act as a wrapper to something else. In layouts, devices are often implemented as PCells and appear as specific cells for no other reason than being implemented in a subcell. The same might happen for schematic subcircuits which wrap a device. "Flattening" means that a circuit is removed and it's contents are integrated into the calling circuits.

To flatten a circuit from the extracted netlist use Netlist#flatten_circuit:

```
netlist.flatten_circuit("CIRCUIT_NAME")
```

The argument to "flatten_circuit" is a glob pattern (shell-like). For example, "NMOS*" will flatten all circuits starting with "NMOS".

Automatic circuit flattening (netlist alignment)

Instead of flattening circuits explicitly, automatic flattening is provided through the align method.

The "align" step is optional, hence useful: it will identify cells in the layout without a corresponding schematic circuit and flatten them. "Flatten" means their content is replicated inside their parent circuits and finally the cell's corresponding circuit is removed. This is useful when the layout contains structural cells: such cells are inserted not because the schematic requires them as circuit building blocks, but because layout is easier to create with these cells. Such cells can be PCells for devices or replication cells which avoid duplicate layout work.

The "align" method will also flatten schematic circuits for which there is no layout cell:

```
align
```

Black boxing (circuit abstraction)

Circuit abstraction is a technique to reduce the verification overhead. At an early stage it might be useful to replace a cell by a simplified version or by a raw pin frame. The circuits extracted from such cells is basically empty or are intentionally simplified. But as long as there is something inside the cell which the parent circuit connects to, pins will be generated. These pins then can be thought of as the circuit's abstraction.

A useful method in this context is the "blank_circuit" method. It clears a circuit's innards and leaves only the pins. You can use this method to ensure abstracts in both the layout netlist and the schematic. After this, the compare algorithm will identify both circuits as identical, provided they feature the same number of pins.

To wipe out the innards of a circuit, use the Netlist#blank_circuit method:

```
netlist.blank_circuit("CIRCUIT_NAME")
schematic.blank_circuit("CIRCUIT_NAME")
```

The argument to "blank_circuit" is a glob pattern (shell-like). For example, "MEMORY*" will blank out all circuits starting with "MEMORY".

NOTE: Use "blank_circuit" before "purge" or "simplify" (see below). This method sets a flag (Circuit#dont_purge) which prevents purging of abstract circuits.

Purging (elimination of redundancy)

Extracted netlists often contain elements without a functional aspect: via cells for example generate subcircuits with a single pin and no device. Isolated metal islands (letters, logos, fill/planarisation patches) will create floating nets etc. Two methods are available to purge those elements.

Netlist#purge will remove all floating nets, all circuits without devices or subcircuits.
Netlist#purge_nets will only purge floating nets. Floating nets are nets which don't connect to any device or subcircuit.

```
netlist.purge
netlist.purge_nets
```

Normalization wrapper (simplification)

Netlist#simplify is a wrapper for "make_top_level_pins", "combine_devices" and "purge" in the recommended order:

```
netlist.simplify
```

LVS Reference

- LVS Reference: Netter object
- LVS Reference: Global Functions

LVS Reference: Netter object

The Netter object provides services related to network extraction from a layout plus comparison against a reference netlist. Similar to the DRC DRC::Netter (which lacks the compare ability), the relevant method of this object are available as global functions too where they act on a default incarnation. Usually it's not required to instantiate a Netter object explicitly.

The LVS Netter object inherits all methods of the DRC::Netter.

An individual netter object can be created, if the netter results need to be kept for multiple extractions. If you really need a Netter object, use the global global#netter function:

```
# create a new Netter object:
nx = netter

# build connectivity
nx.connect(poly, contact)
...

# read the reference netlist
nx.schematic("reference.cir")

# configure the netlist compare
nx.same_circuits("A", "B")
...

# runs the compare
if ! nx.compare
puts("no equivalence!")
end
```

- "align" - Aligns the extracted netlist vs. the schematic
- "compare" - Compares the extracted netlist vs. the schematic
- "equivalent_pins" - Marks pins as equivalent
- "lvs_data" - Gets the internal LayoutVsSchematic object
- "max_depth" - Configures the maximum search depth for net match deduction
- "max_res" - Ignores resistors with a resistance above a certain value
- "min_caps" - Ignores capacitors with a capacitance below a certain value
- "same_circuits" - Establishes an equivalence between the circuits
- "same_device_classes" - Establishes an equivalence between the device classes
- "same_nets" - Establishes an equivalence between the nets
- "schematic" - Gets, sets or reads the reference netlist

"align" - Aligns the extracted netlist vs. the schematic

Usage:

- `align`

The align method will modify the netlists in case of missing corresponding circuits. It will flatten these

circuits, thus improving the equivalence between the netlists. Top level circuits are not flattened.

This feature is in particular useful to remove structural cells like device PCells, reuse blocks etc.

This method will also remove schematic circuits for which there is no corresponding layout cell. In the extreme case of flat layout this will result in a flat vs. flat compare.

"netlist.flatten_circuit(...)" or "schematic.flatten_circuit(...)" are other (explicit) ways to flatten circuits.

Please note that flattening circuits has some side effects such as loss of details in the cross reference and net layout.

"compare" - Compares the extracted netlist vs. the schematic

Usage:

- `compare`

Before using this method, a schematic netlist has to be loaded with schematic. The compare can be configured in more details using same_nets, same_circuits, same_device_classes and equivalent_pins.

The compare method will also modify the netlists in case of missing corresponding circuits: the unpaired circuit will be flattened then.

This method will return true, if the netlists are equivalent and false otherwise.

"equivalent_pins" - Marks pins as equivalent

Usage:

- `equivalent_pins(circuit, pin ...)`

This method will mark the given pins as equivalent. This gives the compare algorithm more degrees of freedom when establishing net correspondence. Typically this method is used to declare inputs from gates are equivalent where are are logically, but not physically (e.g. in a CMOS NAND gate):

```
netter.equivalent_pins("NAND2", 0, 1)
```

The circuit argument is either a circuit name (a string) or a Circuit object from the schematic netlist.

The pin arguments are zero-based pin numbers, where 0 is the first number, 1 the second etc. If the netlist provides named pins, names can be used instead of numbers.

Before this method can be used, a schematic netlist needs to be loaded with schematic.

"lvs_data" - Gets the internal LayoutVsSchematic object

Usage:

- `lvs_data`

The LayoutVsSchematic object provides access to the internal details of the netter object.

"max_depth" - Configures the maximum search depth for net match deduction

Usage:

- `max_depth(n)`

The netlist compare algorithm works recursively: once a net equivalence is established, additional matches are derived from this equivalence. Such equivalences in turn are used to derive new equivalences and so on. The maximum depth parameter configures the number of recursions the algorithm performs before picking the next net. With higher values for the depth, the algorithm pursues this "deduction path" in greater depth while with smaller values, the algorithm prefers picking nets in a random fashion as the seeds for this deduction path. The default value is 8.

"max_res" - Ignores resistors with a resistance above a certain value

Usage:

- `max_res(threshold)`

After using this method, the netlist compare will ignore resistor devices with a resistance value above the given threshold (in Farad).

"min_caps" - Ignores capacitors with a capacitance below a certain value

Usage:

- `min_caps(threshold)`

After using this method, the netlist compare will ignore capacitance devices with a capacitance values below the given threshold (in Farad).

"same_circuits" - Establishes an equivalence between the circuits

Usage:

- `same_circuits(circuit_a, circuit_b)`

This method will force an equivalence between the two circuits. By default, circuits are identified by name. If names are different, this method allows establishing an explicit correspondence.

One of the circuits may be nil. In this case, the corresponding other circuit is mapped to "nothing", i.e. ignored.

Before this method can be used, a schematic netlist needs to be loaded with schematic.

"same_device_classes" - Establishes an equivalence between the device classes

Usage:

- `same_device_classes(class_a, class_b)`

This method will force an equivalence between the two device classes. Device classes are also known as "models". By default, device classes are identified by name. If names are different, this method allows establishing an explicit correspondence.

One of the device classes may be nil. In this case, the corresponding other device class is mapped to "nothing", i.e. ignored.

Before this method can be used, a schematic netlist needs to be loaded with schematic.

"same_nets" - Establishes an equivalence between the nets

Usage:

- `same_nets(circuit, net_a, net_b)`
- `same_nets(circuit_a, net_a, circuit_b, net_b)`

This method will force an equivalence between the net_a and net_b from circuit_a and circuit_b (circuit in the three-argument form is for both circuit_a and circuit_b).

In the four-argument form, the circuits can be either given by name or as Circuit objects. In the three-argument form, the circuit has to be given by name. Nets can be either given by name or as Net objects.

After using this function, the compare algorithm will consider these nets equivalent. Use this method to provide hints for the comparer in cases which are difficult to resolve otherwise.

Before this method can be used, a schematic netlist needs to be loaded with schematic.

"schematic" - Gets, sets or reads the reference netlist

Usage:

- `schematic(filename)`
- `schematic(filename, reader)`
- `schematic(netlist)`
- `schematic`

If no argument is given, the current schematic netlist is returned. nil is returned if no schematic netlist is set yet.

If a filename is given (first two forms), the netlist is read from the given file. If no reader is provided, Spice format will be assumed. The reader object is a NetlistReader object and allows detailed customization of the reader process.

Alternatively, a Netlist object can be given which is obtained from any other source.

LVS Reference: Global Functions

Some functions are available on global level and can be used without any object. Most of them are convenience functions that basically act on some default object or provide function-like alternatives for the methods.

LVS is built upon DRC. So all functions available in DRC are also available in LVS. In LVS, DRC functions are used to derive functional layers from original layers or specification of the layout source.

For more details about the DRC functions see DRC::global.

- "netter" - Creates a new netter object
- "report_lvs" - Specifies an LVS report for output
- "same_circuits" - Establishes an equivalence between the circuits
- "same_device_classes" - Establishes an equivalence between the device_classes
- "same_nets" - Establishes an equivalence between the nets
- "schematic" - Reads the reference netlist

"align" - Aligns the extracted netlist vs. the schematic by flattening circuits where required

Usage:

- `align`

See Netter#align for a description of that function.

"compare" - Compares the extracted netlist vs. the schematic

Usage:

- `compare`

See Netter#compare for a description of that function.

"equivalent_pins" - Marks pins as equivalent

Usage:

- `equivalent_pins(circuit, pins ...)`

See Netter#equivalent_pins for a description of that function.

"max_branch_complexity" - Configures the maximum branch complexity for ambiguous net matching

Usage:

- `max_branch_complexity(n)`

See Netter#max_branch_complexity for a description of that function.

"max_depth" - Configures the maximum search depth for net match deduction

Usage:

- `max_depth(n)`

See Netter#max_depth for a description of that function.

"max_res" - Ignores resistors with a resistance above a certain value

Usage:

- `max_res(threshold)`

See Netter#max_res for a description of that function.

"min_caps" - Ignores capacitors with a capacitance below a certain value

Usage:

- `min_caps(threshold)`

See Netter#min_caps for a description of that function.

"netter" - Creates a new netter object

Usage:

- `netter`

See Netter for more details

"report_lvs" - Specifies an LVS report for output

Usage:

- `report_lvs([ filename [, long ] ])`

After the comparison step, the LVS database will be shown in the netlist database browser in a cross-reference view. If a filename is given, the LVS database is also written to this file. If a file name is given and "long" is true, a verbose version of the LVS DB format will be used.

If this method is called together with report_netlist and two files each, two files can be generated - one for the extracted netlist (L2N database) and one for the LVS database. However, report_netlist will only write the extracted netlist while report_lvs will write the LVS database which also includes the extracted netlist.

report_lvs is only effective if a comparison step is included.

"same_circuits" - Establishes an equivalence between the circuits

Usage:

- `same_circuits(circuit_a, circuit_b)`

See Netter#same_circuits for a description of that function.

"same_device_classes" - Establishes an equivalence between the device_classes

Usage:

- `same_device_classes(class_a, class_b)`

See Netter#same_device_classes for a description of that function.

"same_nets" - Establishes an equivalence between the nets

Usage:

- `same_nets(circuit, net_a, net_b)`
- `same_nets(circuit_a, net_a, circuit_b, net_b)`

See Netter#same_nets for a description of that function.

"schematic" - Reads the reference netlist

Usage:

- `schematic(filename)`
- `schematic(filename, reader)`
- `schematic(netlist)`

See Netter#schematic for a description of that function.

DRC Reference: Global Functions

Some functions are available on global level and can be used without any object. Most of them are convenience functions that basically act on some default object or provide function-like alternatives for the methods.

- "antenna_check" - Performs an antenna check
- "bjt3" - Supplies the BJT3 transistor extractor class
- "bjt4" - Supplies the BJT4 transistor extractor class
- "box" - Creates a box object
- "capacitor" - Supplies the capacitor extractor class
- "capacitor_with_bulk" - Supplies the capacitor extractor class that includes a bulk terminal
- "cell" - Selects a cell for input on the default source
- "clear_connections" - Clears all connections stored so far
- "clip" - Specifies clipped input on the default source
- "connect" - Specifies a connection between two layers
- "connect_global" - Specifies a connection to a global net
- "connect_implicit" - Specifies a label pattern for implicit net connections
- "dbu" - Gets or sets the database unit to use
- "deep" - Enters deep (hierarchical) mode
- "device_scaling" - Specifies a dimension scale factor for the geometrical device properties
- "diode" - Supplies the diode extractor class
- "edge" - Creates an edge object
- "edge_layer" - Creates an empty edge layer
- "error" - Prints an error
- "extent" - Creates a new layer with the bounding box of the default source
- "extract_devices" - Extracts devices for a given device extractor and device layer selection
- "flat" - Disables tiling mode
- "info" - Outputs as message to the logger window
- "input" - Fetches the shapes from the specified input from the default source
- "is_deep?" - Returns true, if in deep mode
- "is_tiled?" - Returns true, if in tiled mode
- "l2n_data" - Gets the internal LayoutToNetlist object for the default Netter
- "labels" - Gets the labels (text) from an original layer
- "layers" - Gets the layers contained in the default source
- "layout" - Specifies an additional layout for the input source.
- "log" - Outputs as message to the logger window
- "log_file" - Specify the log file where to send to log to
- "make_layer" - Creates an empty polygon layer based on the hierarchical scheme selected
- "mos3" - Supplies the MOS3 transistor extractor class
- "mos4" - Supplies the MOS4 transistor extractor class
- "netlist" - Obtains the extracted netlist from the default Netter
- "netter" - Creates a new netter object
- "no_borders" - Reset the tile borders
- "output" - Outputs a layer to the report database or output layout
- "output_cell" - Specifies a target cell, but does not change the target layout
- "p" - Creates a point object

- "path" - Creates a path object
- "polygon" - Creates a polygon object
- "polygon_layer" - Creates an empty polygon layer
- "polygons" - Fetches the polygons (or shapes that can be converted to polygons) from the specified input from the default source
- "report" - Specifies a report database for output
- "report_netlist" - Specifies an extracted netlist report for output
- "resistor" - Supplies the resistor extractor class
- "resistor_with_bulk" - Supplies the resistor extractor class that includes a bulk terminal
- "select" - Specifies cell filters on the default source
- "silent" - Resets verbose mode
- "source" - Specifies a source layout
- "target" - Specify the target layout
- "target_netlist" - With this statement, an extracted netlist is finally written to a file
- "threads" - Specifies the number of CPU cores to use in tiling mode
- "tile_borders" - Specifies a minimum tile border
- "tiles" - Specifies tiling
- "verbose" - Sets or resets verbose mode
- "verbose?" - Returns true, if verbose mode is enabled
- "write_spice" - Defines SPICE output format (with options)

"antenna_check" - Performs an antenna check

Usage:

- `antenna_check(gate, metal, ratio, [ diode_specs ... ])`

See Netter#antenna_check for a description of that function.

"bjt3" - Supplies the BJT3 transistor extractor class

Usage:

- `bjt3(name)`

Use this class with extract_devices to specify extraction of a bipolar junction transistor

"bjt4" - Supplies the BJT4 transistor extractor class

Usage:

- `bjt4(name)`

Use this class with extract_devices to specify extraction of a bipolar junction transistor with a substrate terminal

"box" - Creates a box object

Usage:

- `box(...)`

This function creates a box object. The arguments are the same than for the DBox constructors.

"capacitor" - Supplies the capacitor extractor class

Usage:

- `capacitor(name, area_cap)`

Use this class with extract_devices to specify extraction of a capacitor. The area_cap argument is the capacitance in Farad per square micrometer.

"capacitor_with_bulk" - Supplies the capacitor extractor class that includes a bulk terminal

Usage:

- `capacitor_with_bulk(name, area_cap)`

Use this class with extract_devices to specify extraction of a capacitor with a bulk terminal. The area_cap argument is the capacitance in Farad per square micrometer.

"cell" - Selects a cell for input on the default source

Usage:

- `cell(args)`

See Source#cell for a description of that function. In addition to the functionality described there, the global function will also send the output to the specified cell.

The following code will select cell "MACRO" from the input layout:

```
cell("MACRO")
# shapes now will be taken from cell "MACRO"
l1 = input(1, 0)
```

"clear_connections" - Clears all connections stored so far

Usage:

- `clear_connections`

See Netter#clear_connections for a description of that function.

"clip" - Specifies clipped input on the default source

Usage:

- `clip(args)`

See Source#clip for a description of that function.

The following code will select shapes within a 500x600 micron rectangle (lower left corner at 0,0) from the input layout. The shapes will be clipped to that rectangle:

```
clip(0.mm, 0.mm, 0.5.mm, 0.6.mm)
# shapes now will be taken from the given rectangle and clipped to it
l1 = input(1, 0)
```

"connect" - Specifies a connection between two layers

Usage:

- `connect(a, b)`

See Netter#connect for a description of that function.

"connect_global" - Specifies a connection to a global net

Usage:

- `connect_global(l, name)`

See Netter#connect_global for a description of that function.

"connect_implicit" - Specifies a label pattern for implicit net connections

Usage:

- `connect_implicit(label_pattern)`

See Netter#connect_implicit for a description of that function.

"dbu" - Gets or sets the database unit to use

Usage:

- `dbu(dbu)`
- `dbu`

Without any argument, this method gets the database unit used inside the DRC engine.

With an argument, sets the database unit used internally in the DRC engine. Without using that method, the database unit is automatically taken as the database unit of the last input. A specific database unit can be set in order to optimize for two layouts (i.e. take the largest common denominator). When the database unit is set, it must be set at the beginning of the script and before any operation that uses it.

"deep" - Enters deep (hierarchical) mode

Usage:

- `deep`

In deep mode, the operations will be performed in a hierarchical fashion. Sometimes this reduces the time and memory required for an operation, but this will also add some overhead for the hierarchical analysis.

"deepness" is a property of layers. Layers created with "input" while in deep mode carry hierarchy. Operations involving such layers at the only or the first argument are carried out in hierarchical mode.

Hierarchical mode has some more implications, like "merged_semantics" being implied always. Sometimes cell variants will be created.

Deep mode can be cancelled with tiles or flat.

"device_scaling" - Specifies a dimension scale factor for the geometrical device properties

Usage:

- `device_scaling(factor)`

See Netter#device_scaling for a description of that function.

"diode" - Supplies the diode extractor class

Usage:

- `diode(name)`

Use this class with extract_devices to specify extraction of a planar diode

"edge" - Creates an edge object

Usage:

- `edge(...)`

This function creates an edge object. The arguments are the same than for the DEdge constructors.

"edge_layer" - Creates an empty edge layer

Usage:

- `edge_layer`

The intention of that method is to create an empty layer which can be filled with edge objects using Layer#insert.

"error" - Prints an error

Usage:

- `error(message)`

Similar to log, but the message is printed formatted as an error

"extent" - Creates a new layer with the bounding box of the default source

Usage:

- `extent`

See Source#extent for a description of that function.

"extract_devices" - Extracts devices for a given device extractor and device layer selection

Usage:

- `extract_devices(extractor, layer_hash)`
- `extract_devices(extractor_class, name, layer_hash)`

See Netter#extract_devices for a description of that function.

"flat" - Disables tiling mode

Usage:

- `flat`

Disables tiling mode. Tiling mode can be enabled again with tiles later.

"info" - Outputs as message to the logger window

Usage:

- `info(message)`

Prints the message to the log window in verbose mode. In non-verbose more, nothing is printed. log is a function that always prints a message.

"input" - Fetches the shapes from the specified input from the default source

Usage:

- `input(args)`

See Source#input for a description of that function. This method will fetch polygons and labels. See polygons and labels for more specific versions of this method.

"is_deep?" - Returns true, if in deep mode

Usage:

- `is_deep?`

"is_tiled?" - Returns true, if in tiled mode

Usage:

- `is_tiled?`

"l2n_data" - Gets the internal LayoutToNetlist object for the default Netter

Usage:

- `l2n_data`

See Netter#l2n_data for a description of that function.

"labels" - Gets the labels (text) from an original layer

Usage:

- `labels`

See Source#labels for a description of that function.

"layers" - Gets the layers contained in the default source

Usage:

- `layers`

See Source#layers for a description of that function.

"layout" - Specifies an additional layout for the input source.

Usage:

- `layout`
- `layout(what)`

This function can be used to specify a new layout for input. It returns an Source object representing that layout. The "input" method of that object can be used to get input layers for that layout.

"what" specifies what input to use. "what" be either

- A string "@n" specifying input from a cellview in the current view
- A layout filename plus an optional cell name
- A Layout object
- A Cell object

Without any arguments the default layout is returned.

If a file name is given, a cell name can be specified as the second argument. If not, the top cell is taken which must be unique in that case.

Having specified a layout for input enables to use the input method for getting input:

```
# XOR between layers 1 or the default input and "second_layout.gds":
l2 = layout("second_layout.gds")
(input(1, 0) ^ l2.input(1, 0)).output(100, 0)
```

For further methods on the source object see Source.

"log" - Outputs as message to the logger window

Usage:

- `log(message)`

Prints the message to the log window. info is a function that prints a message only if verbose mode is enabled.

"log_file" - Specify the log file where to send to log to

Usage:

- `log_file(filename)`

After using that method, the log output is sent to the given file instead of the logger window or the terminal.

"make_layer" - Creates an empty polygon layer based on the hierarchical scheme selected

Usage:

- `make_layer`

The intention of this method is to provide an empty polygon layer based on the hierarchical scheme selected. This will create a new layer with the hierarchy of the current layout in deep mode and a flat layer in flat mode. This method is similar to polygon_layer, but the latter does not create a hierarchical layer. Hence the layer created by make_layer is suitable for use in device extraction for example, while the one delivered by polygon_layer is not.

On the other hand, a layer created by the make_layer method is not intended to be filled with Layer#insert.

"mos3" - Supplies the MOS3 transistor extractor class

Usage:

- `mos3(name)`

Use this class with extract_devices to specify extraction of a three-terminal MOS transistor

"mos4" - Supplies the MOS4 transistor extractor class

Usage:

- `mos4(name)`

Use this class with extract_devices to specify extraction of a four-terminal MOS transistor

"netlist" - Obtains the extracted netlist from the default Netter

The netlist is a Netlist object. If no netlist is extracted yet, this method will trigger the extraction process. See Netter#netlist for a description of this function.

"netter" - Creates a new netter object

Usage:

- `netter`

See Netter for more details

"no_borders" - Reset the tile borders

Usage:

- `no_borders`

Resets the tile borders - see tile_borders for a description of tile borders.

"output" - Outputs a layer to the report database or output layout

Usage:

- `output(layer, args)`

This function is equivalent to "layer.output(args)". See Layer#output for details about this function.

"output_cell" - Specifies a target cell, but does not change the target layout

Usage:

- `output_cell(cellname)`

This method switches output to the specified cell, but does not change the target layout nor does it switch the output channel to layout if is report database.

"p" - Creates a point object

Usage:

- `p(x, y)`

A point is not a valid object by itself, but it is useful for creating paths for polygons:

```
x = polygon_layer
x.insert(polygon([ p(0, 0), p(16.0, 0), p(8.0, 8.0) ]))
```

"path" - Creates a path object

Usage:

- `path(...)`

This function creates a path object. The arguments are the same than for the DPath constructors.

"polygon" - Creates a polygon object

Usage:

- `polygon(...)`

This function creates a polygon object. The arguments are the same than for the DPolygon constructors.

"polygon_layer" - Creates an empty polygon layer

Usage:

- `polygon_layer`

The intention of that method is to create an empty layer which can be filled with polygon-like objects using Layer#insert. A similar method which creates a hierarchical layer in deep mode is make_layer. This other layer is better suited for use with device extraction.

"polygons" - Fetches the polygons (or shapes that can be converted to polygons) from the specified input from the default source

Usage:

- `polygons(args)`

See Source#polygons for a description of that function.

"report" - Specifies a report database for output

Usage:

- `report(description [, filename [, cellname ] ])`

After specifying a report database for output, output method calls are redirected to the report database. The format of the output calls changes and a category name plus description can be specified rather than a layer/datatype number of layer name. See the description of the output method for details.

If a filename is given, the report database will be written to the specified file name. Otherwise it will be shown but not written.

If external input is specified with source, "report" must be called after "source".

The cellname specifies the top cell used for the report file. By default this is the cell name of the default source. If there is no source layout you'll need to give the cell name in the third parameter.

"report_netlist" - Specifies an extracted netlist report for output

Usage:

- `report_netlist([ filename [, long ] ])`

This method applies to runsets creating a netlist through extraction. Extraction happens when connections and/or device extractions are made. If this statement is used, the extracted netlist plus the net and device shapes are turned into a layout-to-netlist report (L2N database) and shown in the netlist browser window. If a file name is given, the report will also be written to the given file. If a file name is given and "long" is true, a verbose version of the L2N DB format will be used.

"resistor" - Supplies the resistor extractor class

Usage:

- `resistor(name, sheet_rho)`

Use this class with extract_devices to specify extraction of a resistor. The sheet_rho value is the sheet resistance in ohms/square.

"resistor_with_bulk" - Supplies the resistor extractor class that includes a bulk terminal

Usage:

- `resistor_with_bulk(name, sheet_rho)`

Use this class with extract_devices to specify extraction of a resistor with a bulk terminal. The sheet_rho value is the sheet resistance in ohms/square.

"select" - Specifies cell filters on the default source

Usage:

- `select(args)`

See Source#select for a description of that function.

"silent" - Resets verbose mode

Usage:

- `silent`

This function is equivalent to "verbose(false)" (see verbose)

"source" - Specifies a source layout

Usage:

- `source`
- `source(what)`

This function replaces the default source layout by the specified file. If this function is not used, the currently active layout is used as input.

layout is a similar method which specifies *a additional* input layout.

"what" specifies what input to use. "what" be either

- A string "@n" specifying input from a layout in the current panel
- A layout filename plus an optional cell name
- A Layout object plus an optional cell name
- A Cell object

Without any arguments the default layout is returned. If a filename is given, a cell name can be specified as the second argument. If none is specified, the top cell is taken which must be unique in that case.

```
# XOR between layers 1 of "first_layout.gds" and "second_layout.gds" and sends the
results to "xor_layout.gds":
target("xor_layout.gds")
source("first_layout.gds")
l2 = layout("second_layout.gds")
(input(1, 0) ^ l2.input(1, 0)).output(100, 0)
```

For further methods on the source object see Source.

"target" - Specify the target layout

Usage:

- `target(what [, cellname])`

This function can be used to specify a target layout for output. Subsequent calls of "output" will send their results to that target layout. Using "target" will disable output to a report database. If any target was specified before, that target will be closed and a new target will be set up.

"what" specifies what input to use. "what" be either

- A string "@n" specifying output to a layout in the current panel
- A layout filename
- A Layout object

- A Cell object

Except if the argument is a Cell object, a cellname can be specified stating the cell name under which the results are saved. If no cellname is specified, either the current cell or "TOP" is used.

"target_netlist" - With this statement, an extracted netlist is finally written to a file

Usage:

- `target_netlist(filename [, format [, comment ] ])`

This method applies to runsets creating a netlist through extraction. Extraction happens when connections and/or device extractions are made. If this statement is used, the extracted netlist is written to the given file.

The format parameter specifies the writer to use. You can use nil to use the standard format or produce a SPICE writer with write_spice. See write_spice for more details.

"threads" - Specifies the number of CPU cores to use in tiling mode

Usage:

- `threads(n)`

If using threads, tiles are distributed on multiple CPU cores for parallelization. Still, all tiles must be processed before the operation proceeds with the next statement.

"tile_borders" - Specifies a minimum tile border

Usage:

- `tile_border(b)`
- `tile_border(bx, by)`

The tile border specifies the distance to which shapes are collected into the tile. In order words, when processing a tile, shapes within the border distance participate in the operations.

For some operations such as booleans (and, or, ...), size and the DRC functions (width, space, ...) a tile border is automatically established. For other operations such as with_area or edges, the exact distance is unknown, because such operations may have a long range. In that cases, no border is used. The tile_borders function may be used to specify a minimum border which is used in that case. That allows taking into account at least shapes within the given range, although not necessarily all.

To reset the tile borders, use no_borders or "tile_borders(nil)".

"tiles" - Specifies tiling

Usage:

- `tiles(t)`
- `tiles(w, h)`

Specifies tiling mode. In tiling mode, the DRC operations are evaluated in tiles with width w and height h. With one argument, square tiles with width and height t are used.

Special care must be taken when using tiling mode, since some operations may not behave as expected

at the borders of the tile. Tiles can be made overlapping by specifying a tile border dimension with tile_borders. Some operations like sizing, the DRC functions specify a tile border implicitly. Other operations without a defined range won't do so and the consequences of tiling mode can be difficult to predict.

In tiling mode, the memory requirements are usually smaller (depending on the choice of the tile size) and multi-CPU support is enabled (see threads). To disable tiling mode use flat or deep.

Tiling mode will disable deep mode (see deep).

"verbose" - Sets or resets verbose mode

Usage:

- `verbose`
- `verbose(m)`

In verbose mode, more output is generated in the log file

"verbose?" - Returns true, if verbose mode is enabled

Usage:

- `verbose?`

In verbose mode, more output is generated in the log file

"write_spice" - Defines SPICE output format (with options)

Usage:

- `write_spice([ use_net_names [, with_comments ] ])`

Use this option in target_netlist for the format parameter to specify SPICE format. "use_net_names" and "with_comments" are boolean parameters indicating whether to use named nets (numbers if false) and whether to add information comments such as instance coordinates or pin names.