OpenPCells

PCell Design Guide and API

Patrick Kurth

October 6, 2023

This is the official documentation of the OpenPCells project. It is split in several different files for clarity. This document provides an overview of the creation of PCells in the OpenPCells environment as well as a detailed API documentation. If you are looking for a general overview of the project and how to use it, start with the user guide, which also contains a tutorial for getting started quickly. If you want to know more about the technical details and implementation notes, look into the technical documentation.

Contents

1 PCell Creation – Introductory Examples

We will start this documentation by a series of examples to show the main features and API functions. The to-be-created cells will get increasingly complex to demonstrate various features.

Every cell is defined by a function where all shapes making up the shape are described. This function gets called by the cell generation system, which passes the main object and a table with all defined parameters. The name for this function is `layout()`. Additional functions such as `parameters()` are also understood.

1.1 First Example – Simple Rectangle

The first example is a simple rectangle of variable width and height. As mentioned, all the code for the rectangle resides in a function `layout()`. The parameters of the cell are defined in a function `parameters()`, which is optional in theory, but since we're designing pcells, there is not much point of leaving it out. In `layout()`, we receive the main object and the defined parameters. Here we can modify the object based on the parameters.

The simple rectangle looks like this:

```lua
-- define parameters
function parameters()
    pcell.add_parameters(
        { "width",  100 },
        { "height", 100 }
    )
end

-- define layout
function layout(obj, _P)
    -- create the shape and add it to the main object
    geometry.rectanglebltr(
        obj,
        generics.metal(1),
        point.create(0, 0),
        point.create(_P.width, _P.height)
    )
end
```

Let's walk through this line-by-line (sort of). First, we declare the function for the parameter definition:

```
function parameters()
```

In the function, we add the parameters, here we use the width and the height of the rectangle:

```
pcell.add_parameters(
    { "width",  100 },
    { "height", 100 }
)
```

We can add as many parameters as we like (`pcell.add_parameters()` accepts any number of arguments). For every argument, the first entry in the table is the name of the parameter, the second entry is the default value. This is the simplest form, we can supply more information for finer control. We will see some examples of this later on.

The default value for both parameters is 100, which is a *size*, meaning it has a unit. Physical/geometrical parameters like width or height are specified in nanometers.[1]

This is all for the `parameters()` function, so let's move on to `layout()`. This functions takes two arguments: the main object that will be placed in the layout and the table with parameters for the cell (which already includes any parsed arguments given before the cell creation).

We can name them in any way that pleases us, the common name used in all standard cells distributed by this project is `_P` (as hommage to the global environment `_G` in lua). Of course it is possible to "unpack" the parameters, storing them in individual variables, but for cells with many parameters this rather is a bloat.

```
function layout(obj, _P)
```

Now that we have all the layout parameters, we can already create the rectangle:

```
geometry.rectanglebltr(
    obj,
    generics.metal(1),
    point.create(0, 0),
    point.create(_P.width, _P.height)
)
```

There is a lot going on here: We use the `geometry.rectanglebltr` function to create a rectangle with two corner points (bottom-left, bl and top-right, tr). Since we are creating shapes of IC geometry, we have to specify a layer. But we also want to create technology-independent pcells, so there is a generics system for layers. Right now we are just using the `generics.metal` function, which takes a single number as argument. `generics.metal(1)` specifies the first metal (counted from silicon), you can also say something like `generics.metal(-2)`, where `-1` is the index of the highest metal. Lastly we pass the main object as first argument to the function, which places the rectangle within this object.

This cell can now be created by calling the main program with an appropriate export and technology. Note that there's another manual about that, so we won't get into any details here. The simplest call would be something like

[1]Well, this is not entirely sure. Only integers are allowed and the base unit is assumed to be nanometer. This is also currently reflected for example in the GDSII export, where the scaling is done approriately. However, it is planned that this will change in the future, making the base unit in opc arbitrary.

```
opc --technology opc --export gds --cell library/simple_rectangle
```

where `library` is a folder where the cell is placed in and lies in the cell search path.

Now you already now how to create simple rectangles with generic layers. As integrated circuits are mostly made up of rectangles, one can already built a surprising amount of pcells. However, we have to discuss how we can create layers other than metals, vias and shapes with more complex outlines than rectangles. Furthermore, to reduce complexity hierarchical layout design is also important. We will talk about these topics in the remaining cell tutorials.

1.2 Metal-Oxide-Metal Capacitors

Many technologies don't have so-called metal-insulator-metal capacitors (mimcaps), so the standard way to implement capacitors is be using interdigitated metals. Let's do that. As before, we set up the pcell. Useful parameters are the number of fingers, the width and height of the fingers and the spacing in between. Furthermore, we shift one collection of fingers (one plate) up and the other down to separate them and connect them together. Lastly, we also specify the width of the connecting rails and the used metals. The shown cell is a simplified version of the actual momcap implementation in opc:

```lua
function parameters()
    pcell.add_parameters(
        { "fingers(Number of Fingers)", 4 },
        { "fwidth(Finger Width)",     100 },
        { "fspace(Finger Spacing)",   100 },
        { "fheight(Finger Height)",  1000 },
        { "foffset(Finger Offset)",   100 },
        { "rwidth(Rail Width)",       100 },
        { "rext(Rail Extension)",       0 },
        { "firstmetal(Start Metal)",    1 },
        { "lastmetal(End Metal)",       2 }
    )
end
```

The parameter definition also shows how you can use better names for displaying: Simply write them in parantheses. When listing the defined parameters of a cell, the display names are used, but within the cell the regular names are significant. The parameters are stored in a lua table and can be accessed in two ways: `_P.fingers` and `_P["fingers"]`. Usually, the first way is easier, but it requires the parameter name to be a valid lua identifier. Names like `foo-bar` (with a hyphen) are not valid identifiers. In this case, the second way would have to be used.

In `layout()` first the metals are resolved. This makes sure that only positive integers are used (for instance, with a metal stack of five metals, the index -2 is resolved to 4). This is done in order to have properly-defined values for the following for-loop iterating over all metals.

```lua
local firstmetal = technology.resolve_metal(_P.firstmetal)
local lastmetal = technology.resolve_metal(_P.lastmetal)
```

Then we can set up the loop:

```
for m = firstmetal, lastmetal do
```

At first, we create the rails (upper and lower):

```
    geometry.rectanglebltr(
        momcap, generics.metal(m),
        point.create(-_P.rext, 0),
        point.create(_P.fingers * _P.fwidth + (_P.fingers - 1) *
            _P.fspace + _P.rext, _P.rwidth)
    )
    geometry.rectanglebltr(
        momcap, generics.metal(m),
        point.create(-_P.rext, _P.fheight + 2 * _P.foffset +
            _P.rwidth),
        point.create(_P.fingers * _P.fwidth + (_P.fingers - 1) *
            _P.fspace + _P.rext, _P.fheight + 2 * _P.foffset + 2 *
            _P.rwidth)
    )
```

Then we create the fingers separately. A second loop represents the fingers. Every finger has to be moved right, every second finger has to be moved up:

```
    for f = 1, _P.fingers do
        local xshift = (f - 1) * pitch
        local yshift = (f % 2 == 0) and 0 or _P.foffset
        geometry.rectanglebltr(
            momcap, generics.metal(m),
            point.create(xshift, _P.rwidth + yshift),
            point.create(xshift + _P.fwidth, _P.rwidth + yshift +
                _P.fheight + _P.foffset)
        )
    end
```

What remains is the drawing of the vias between the metals. For this we introduce a new `geometry` function: `geometry.viabltr`. It takes a rectangular area and creates individual cuts as well as surrounding metals. There has to be some technology translation for this (proper layer generation as well as calculating the proper geometry of the cuts). The details on this are not important for this discussion. It is covered more in-depth in the technology translation manual. For this case it is enough to know that an appropriate and manufacturable amount of via cuts is placed within this rectangular area. Since the region is a rectangular, `geometry.viabltr` takes almost the same arguments as `geometry.rectanglebltr`. Only the metal layer is changed into two indices for the first and the last metal of the stack. This means that `geometry.viabltr` does *not* expect a generic layer as input. All layer creation is done by the function itself. Furthermore, we don't have to specify the individual vias between each layer in the stack, this is resolved later by the technology translation. For the capacitor, the vias are placed in the rails:

```
if firstmetal ~= lastmetal then
    geometry.viabltr(
        momcap, firstmetal, lastmetal,
        point.create(-_P.rext, 0),
        point.create(_P.fingers * _P.fwidth + (_P.fingers - 1) *
            _P.fspace + _P.rext, _P.rwidth)
    )
    geometry.viabltr(
        momcap, firstmetal, lastmetal,
        point.create(-_P.rext, _P.fheight + 2 * _P.foffset +
            _P.rwidth),
        point.create(_P.fingers * _P.fwidth + (_P.fingers - 1) *
            _P.fspace + _P.rext, _P.fheight + 2 * _P.foffset + 2 *
            _P.rwidth)
    )
end
```

With this the pcell is almost finished, the remaining code defines so-called *anchors*, which are used for relative positioning. These will be discussed further down this document and therefore are skipped in this section. A cell similar to this is bundled in this release of openPCells (`cells/passive/capacitor/mom.lua`). A few optimizations and additional parameters are added, but the here shown implementation is the basic structure of this capacitor.

1.3 Octagonal Inductor

RF designs often require on-chip inductors, which usually are built in an octagonal shape due to angle restrictions in most technologies (no true circles or better approximations available). We will show how to built a differential (symmetric) octagonal inductor with a variable number of turns (integers). We will skip some basic techniques that we already discussed a few times such as setting up the cell body, cell parameters and main object. Look into `cells/passive/inductor/octagonal.lua` for the defined parameters.

An inductor is basically a wire (a *path*) routed in a special manner, therefore we will describe the inductor as a `path`. This is a series of points that describe a line with a certain width. To create a path, we have to pass the points to `geometry.path`, which we will store in a `table`. The cell for the octagonal inductor requries some calculations for the right point positions and uses some helper functions. We won't discuss the entire cell here as some issues are not important for general advice on building cells. The main purpose is to show how to handle points, point lists and how to draw paths.

First, some information on points: Points are a structure with an x- and a y coordinate. They represent absolute locations in the layout and many layout- and object-related take either a pair of x/y coordinates or a point. Points are simply created by `point.create`, which takes two numbers for the coordinates (x comes first). The coordinates can be queried:

```
local pt = point.create(100, 100) -- create a point
local x, y = pt:unwrap() -- get both x and y
x = pt:getx() -- get only x
y = pt:gety() -- get only y
```

For some layouts, combinations of points are needed, where for instance the x-coordinate of one point should be combined with the y-coordinate of another point. There are two ways to achieve this:

```
-- newpt == point.create(pt1.x, pt2.y)
local newpt = point.combine_12(pt1, pt2)
-- newpt == point.create(pt2.x, pt1.y)
local newpt = point.combine_21(pt1, pt2)
```

Both functions do the same, they just differ on the order of their arguments. Furthermore, some mathematical operators are defined for points:

```
local pt1 = point.create(100, 100)
local pt2 = point.create(20, -100)
print(pt1 + pt2)  -- (60, 0)       -> arithmetic average
print(pt1 - pt2)  -- (80, 200)     -> difference
print(-pt1)       -- (-100, -100)  -> unary minus
print(pt1 .. pt2) -- (100, -100)   -> point.combine_12(pt1, pt2)
```

Lastly, there is a shorthand for the scalar distance in either x or y:

```
point.xdistance(pt1, pt2)
point.ydistance(pt1, pt2)
```

Let us get back to the inductor. It has the number of turns as a parameter. For every turn the points for one half of the turn are calculated, then the path is drawn twice, one time with a mirrored version of the points.

```

local mainmetal = generics.metal(_P.metalnum)
local auxmetal = generics.metal(_P.metalnum - 1)
```

The points are stored in the `table` `pathpts`, `util.make_insert_xy` is a helper function, that returns a function that appends/prepends points to an array. It's purpose is to simplify code, one might as well just use `table.insert`.

Then we add points:

```
-- draw left and right segments
local sign = (_P.turns % 2 == 0) and 1 or -1
for i = 1, _P.turns do
    local radius = _P.radius + (i - 1) * pitch
    local r = _scale_tanpi8(radius)
    sign = -sign
```

Now the cell adds points for the underpass (for the crossing of both sides) as well as the extension for the connections. This discussion will skip these parts as they don't add any value for learning about `geometry.path`. The entire code generating the path points is a bit complex and involves some trigonometric calculations.

After the points are assembled, we can create the path. The cell only draws half of the inductor, so we draw the path twice, one time with mirrored points (notice `util.xmirror(pathpts)` in the second line):

```
        prepend(-(_scale_tanpi8(_P.radius) + pitch / 2) / 2, sign *
            radius)
    end
```

The `geometry.path` function takes five arguments: the cell, the layer, the points of the path, the width and whether to use a miter- or a bevel-join. Bevel-join is default, so `true` is specified for a miter-join. The layers where created earlier as

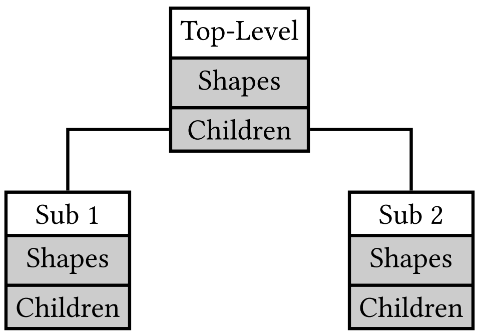
```
    { "includeextensioninboundary(Include Extension in Boundary)",
            true }
)
```

1.4 Integrating Other Cell Layouts

Layouts of integrated circuits can get very complex quite easily. The typical way to deal with high complexity is by building individual cell layouts and placing them in the appropriate location. Partitioning and hierarchy are good tools to tackle circuit complexity. In essence: Divide and conquer. Therefore, this section will discuss how to integrate existing cell layouts and how to find the correct location by relative placement. But first an introduction to how openPCells represents layouts will be given, as this will be important to understand advanced topics.

1.4.1 Representation of Hierarchical Layouts

In integrated circuits, layouts are in general represented by a mixture of instantiations of other layouts as well as shapes (such as rectangles, polygons, paths, etc.). Typically, there is a root cell for the layout, which is called top-level. The top-level layout then has some shapes as well as instantiations of other layouts, which in turn have shapes and Instantiations of other layouts and so on. This is quite similar to a filesystem, where you have a root directory with sub-directories, where every directory can (but not necessarily) contain files that are not directories. The following picture shows an example of this:



Every layout cell holds a list of shapes and a list of children (references to other layout cells). Furthermore, layout cells can be transformed, that is translated in x and y or rotated, mirrored etc. This affects all parts of a cell, so shapes and children are also transformed, which is just what you would expect. Object transformation is a very cheap operation, much cheaper than operating on every single shape in a layout. This is the reason why layout hierarchies are usually much more efficient to work on than so-called flat layouts. How cell hierarchies are created is discussed in section 1.5. In this section we will continue to look at flat layouts but still using other cells.

1.4.2 Adding Cell Contents

For this example, we assume that we have a cell containing some layout in a variable:

```
function layout(toplevel)
    local cell = -- get the layout from somewhere
```

Now we want to place the contents of `cell` in `toplevel`. The method for this is `merge_into`. This method takes all shapes from a cell, copies them and places them in the cell that called `merge_into`. In our example, in order to place the contents from `cell` in `toplevel`, we do the following:

```
toplevel:merge_into(cell)
```

This dissolves the cell and places the content (the shapes and the children) into the toplevel cell.

1.4.3 Moving Cells

Now that we saw how to add cell content to other cells, the question arises how we can place this content at the correct position. As an example, let's say we want to build a ring oscillator. Here, *N* copies of the same inverter are placed. The inverter has a certain width so we know how far we have to move it so that it does not overlap with the surrounding inverters. For the example, we will place three inverters (of which we will assume that a layout is available):

```
function layout(toplevel)
    local inverter = -- create inverter layout
    -- copy inverter and translate it
    local width = 1000 -- the width needs to be known
    local inverter1 = inverter:copy():translate(0 * width, 0)
    local inverter2 = inverter:copy():translate(1 * width, 0)
    local inverter3 = inverter:copy():translate(2 * width, 0)
    -- merge into toplevel
    toplevel:merge_into(inverter1)
    toplevel:merge_into(inverter2)
    toplevel:merge_into(inverter3)
end
```

In this example, the original cell is copied three times, since every call to `translate` changes the internal state of the cell. It is possible to do this without intermediate variables and copying:

```
function layout(toplevel)
    local inverter = -- create inverter layout
    local width = 1000 -- the width needs to be known
    -- translate and merge into toplevel
    toplevel:merge_into(inverter)
    toplevel:merge_into(inverter:translate(width, 0))
    toplevel:merge_into(inverter:translate(width, 0))
end
```

Here, `translate` is called on the original cell, so the movements in x and y accumulate. This is better in regards of processing performance, as copies of cells are expensive.

The approach of explicitly translating cells to move them to the right location works, but requires the knowledge of some parameters of the cell. Usually these values are known, since the layout of a single inverter was also created in this cell. However, for more abstraction it is desirable to describe these kind of layouts in a relative way. For this, the cells should know their own width and height, so they could be placed aligned to each other in an automatic way. There are two mechanisms in openPCells to help with these placements: anchors and alignment boxes.

1.4.4 Cell Anchors

In order to place cell at certain points without knowing their exact geometry, *anchors* are introduced. An anchor is a meaningful/important point of a cell and marks a location where other cells can be attached to or where wires can start/end etc. Think of a jigsaw puzzle pieces with their tabs and blanks: Pieces are connected by placing a tab in a blank of another piece. To come back to the example of the ring oscillator, we connect the inverters by placing the inputs on the outputs of the previous cells. The modified example looks like this:

```
function layout(toplevel)
    local inverter = -- create inverter layout
    -- copy inverter
    local inverter1 = inverter:copy()

    -- get required displacement for inverter2
    local output1 = inverter1:get_anchor("output")
    local input2 = inverter2:get_anchor("input")
    -- copy and translate inverter2
    local inverter2 = inverter:copy():translate(output1 - input2)
    -- get required displacement for inverter3
    local output2 = inverter2:get_anchor("output")
    local input3 = inverter3:get_anchor("input")
    -- copy and translate inverter3
    local inverter3 = inverter:copy():translate(output2 - input3)
    -- merge inverters into toplevel
    toplevel:merge_into(inverter1)
    toplevel:merge_into(inverter2)
    toplevel:merge_into(inverter3)
end
```

The required displacement for each inverter is calculated and then applied to `translate`. This is rather cumbersome. OpenPCells offers a specialized function for this: `move_point`. This takes a layout cell and moves it so that the specified anchor lies at the given location:

```
function layout(toplevel)
    local inverter = -- create inverter layout
    -- copy inverter and move it to the origin
    local inverter1 = inverter:copy():move_point("input")
    inverter1:move_point(
        inverter1:get_anchor("input"),
        point.create(0, 0)
    )
    -- copy and translate inverter2
    local inverter2 = inverter:copy()
    inverter2:move_point(
        inverter2:get_anchor("input"),
        inverter1:get_anchor("output")
    )
    -- copy and translate inverter3
    local inverter3 = inverter:copy()
    inverter3:move_point(
        inverter3:get_anchor("input"),
        inverter2:get_anchor("output")
    )
    -- merge inverters into toplevel
    toplevel:merge_into(inverter1)
    toplevel:merge_into(inverter2)
    toplevel:merge_into(inverter3)
end
```

The function `move_point` takes arbitrary points as arguments. Essentially, it moves the specified cell by the difference of the two points. The interface is a bit akward, because moving cells by their own anchors (a common use case) requires typing out the object name twice. This is due to the generic nature of `move_point`. A more specialized and powerful approach is provided by *area anchors*, as demonstrated in the next section.

1.4.5 Area Anchors

An area anchor is similar to a regular anchor, but it makes up a rectangular area (defined by the bottom-left and the top-right). These anchors can be queried just like regular anchors (but with `get_area_anchor` instead of `get_anchor`). Their two main properties and advantages over regular anchors are that they never change their orientation and cells can be abutted/aligned in various ways with them.

To clarify on the point of orientation: Imagine a rectangular area was made up of two regular anchors. Once the object containing these anchors is rotated by, say, 90 degrees, the bottom-left anchor is no longer the bottom left. Flipping an object for instance reverses the order of these anchors. Area anchors on the other hand always describe, as the name implies, areas. Therefore it does not matter which orientation the respective object has, the bottom-left anchor will always be bottom-left. This enables much simpler approaches, as cell code does not need to keep track of which object is rotated and which one is not.

The second advantage of area anchors are the various abutment/alignment methods for objects. For instance, the `basic/mosfet` cell defines area anchors for every source/drain region. Two devices can be simply abutted by calling

```
mosfet1:align_area_anchor("sourcedrain1", mosfet2, "sourcedrain-1")
```

This places `mosfet1` left of `mosfet2` (as source/drain regions are counted positive beginning from the left and negative beginning from the right). Without area anchors, the call would look like this:

```
mosfet1:move_point(mosfet1:get_anchor("sourcedrain1bl"), mosfet2:
    get_anchor("sourcedrain-1bl"))
```

But this code would break if one of the devices was flipped (for instance to place a gate contact on the top and one on the bottom).

1.4.6 Alignment Boxes

Alignment boxes are tighly connected to area anchors. They provide the same concept but in a simpler and less flexible way. Every cell has at most one alignment box. With this, it can be aligned/abutted to other cells. This makes the most sense for compatible cells (like mosfets and other mosfets).

The above example of abutting two mosfets with area anchors can be re-written as:

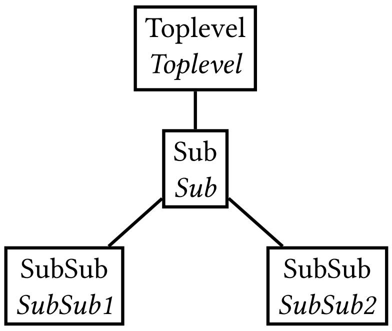
```
mosfet1:abut_right(mosfet2)
mosfet1:align_top(mosfet2)
```

This requires two calls, as every abut/align function with alignment boxes only changes either the x- or the y-coordinate. However, it is anchor-agnostic and therefore does not require any anchors names for the alignment. This also allows writing code that just assembles blocks of layout cells without knowing any anchor names. This code then works for any cells that has an alignment box. This kind of code actually exists in the placement module, which is mainly intended for digital layout.

1.5 Cell Hierarchies

Layouts of integrated circuits usually make great use of repetition/reuse of cells. For instance, a shift register uses the same flip flop over and over again. Creating *flat* layouts (that is, layouts without any hierarchies) for these cells can be quite resource-intense. More shapes have to be calculated by opc and the resulting layout is very likely to be larger in file size than a hierarchical one. Therefore, opc supports hierachical layouts. Instantiations in a cell of other cells/layouts are called *children*. They are light-weight handles to full cells and don't contain any flat shapes of their own. As example a layout that uses a sub-cell 1000 times it only has to store 1000 handles, but not 1000 versions of the same layout.[2] A layout that makes proper use of hierarchy can me multiple magnitudes faster in processing than a flat version.

Each child has a reference it points to, which is created automatically by adding a child to a cell. As an example, let's create the following hierarchy:



The upright name shows the name of the cell (which can be re-used), the italic name is the instance name (this has to be unique). In this example, the top-level cell instantiates another cell (sub), which in turn instantiates two other cells (subsub). In order to create this hierarchy in code, we have to do the following steps: First we create all cells (which are called *references*):

```
local toplevel = object.create("toplevel")
local sub = object.create("sub")
local subsub = object.create("subsub")
```

After that, we simply add the respective cells as children:

```
sub:add_child(subsub, "subsub1")
sub:add_child(subsub, "subsub2")
toplevel:add_child(sub, "sub")
```

It is important not to confuse the cell names here. All functions that create proper cells (objects) such as `pcell.create_layout` or `object.create` must be supplied with a name for that cell. Functions for adding children to objects (currently the object methods `add_child` and `add_child_array`) take an optional parameter for the instance name[3]. If this name is not given, an automatically generated name will be used.

We can see a simple example of proper instance naming in `analog/ringoscillator.lua`. Here, `string.format` is used to generate unique instance names for the individual inverters of a ring oscillator. First, the inverter reference (the actual cell) is created. This inverter is a CMOS structure with some additional wires, so the basic structure of the inverter is based on `basic/cmos`. For brevity, the additional drawings etc. are not shown here.

[2] More realistically, if the layout allows for that, an arrayed version of the sub-cell is stored.
[3] This instance name is not supported by all exports. For example, GDSII has no notion of an instance name

```
    gatecontactpos = invgatecontacts,
    pcontactpos = invactivecontacts,
    ncontactpos = invactivecontacts,
})
pcell.pop_overwrites("basic/cmos")
```

Then the inverter reference (which is only generated once) can be *instantiated* multiple times, which happens within a loop in this case:

```
for i = 1, _P.numinv do
    inverters[i] = oscillator:add_child(inverterref, string.format("
       inverter_%d", i))
    if i > 1 then
        inverters[i]:move_anchor("left", inverters[i - 1]:get_anchor(
           "right"))
    end
end
```

The above example creates a number of inverters (depending on the parameter `numinv`). To add a child, `add_child` is used, which expects a full object as reference and an instance name. Here, the single reference is instantiated multiple times, therefore the instance name is modified in every call. Additionally, the cells are left-right aligned to build a proper layout.

In the above ring oscillator example, the return value of `add_child` is stored in a table. The return value of `add_child` is a so-called *proxy object*. This proxy object behaves like a regular object that it can be moved, rotated and its anchors can be queried (as can be seen in the example). These operations only operate on small data structures and a very lightweight in general. It is usually a good idea to put repeated layout structures in sub-cells for them to be re-used, as this very likely leads to a smaller and faster-processed layout.

2 Cell Scripts

The previous section discussed the use of pcell definitions based on the functions `parameters` and `layout`. For re-used cells this is a good aproach, but some layouts are handled with more similarity to a stand-alone program. For this, *cell scripts* are also supported. For the main part, they function like proper cell files, but cellscripts just describe the content of the layout function of cells. This means that some parts are more manual, for instance the main object must be created and returned by the user. An example cell scripts could look like this:

```
local cell = object.create("toplevel")
geometry.rectangle(cell, generics.metal(1), 100, 100)
return cell
```

Cell scripts have the advantage that they don't have to be placed in some path known to opc. The layout-generation call to opc expects a (absolut or relativ to the calling path) path to the cell script, such as

```
opc --technology opc --export gds --cellscript path/to/cell.lua
```

3 Available PCells

In the following subsections, all available cells will be documented. The current status is rather a poor one, but work is ongoing.

3.1 basic/mosfet

The mosfet device might be the most important cell and currently it's also definitely the most complex one. Therefore, this documentation starts with a description of the goal. Figure 1 shows an example with all geometrical parameters, a summary of all parameters can be found in table 1. The cell draws a number of gates on top of an active area (with some implant/well/etc. markers). Furthermore, it draws some metals and vias (not shown in figure 1) in the source/drain regions

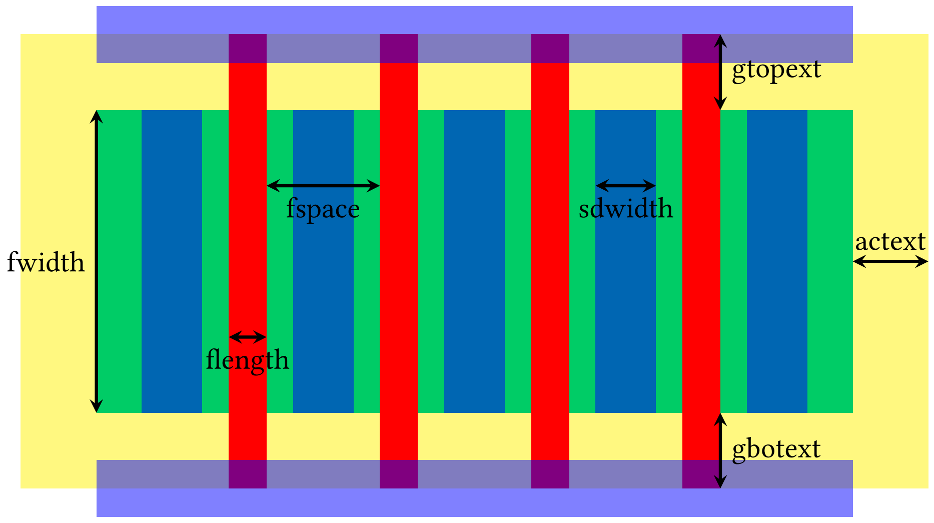


Figure 1: Overview of the transistor

and for gate contacts.

3.2 Gate Parameter

The gates are controlled by the number of fingers, the gate length and space as well as the finger width.

Parameter	Meaning	Default
channeltype	Type of Transistor	”nmos”
oxidetype	Oxide Thickness Index	1
vthtype	Threshold Voltage Index	1
fingers	Number of Fingers	4
fwidth	Finger Width	1.0
gatelength	Finger Length	0.15
fspace	Space between Fingers	0.27
actext	Left/Right Extension of Active Area	0.03
sdwidth	Width of Source/Drain Metals	0.2
sdconnwidth	Width of Source/Drain Connection Rails Metal	0.2
sdconnspace	Space of Source/Drain Connection Rails Metal	0.2
gtopext	Gate Top Extension	0.2
gbotext	Gate Bottom Extension	0.2
typext	Implant/Well Extension around Active	0.1
cliptop	Clip Top Marking Layers (Implant, Well, etc.)	false
clipbot	Clip Bottom Marking Layers (Implant, Well, etc.)	false
drawtopgate	Draw Top Gate Strap	false
drawbotgate	Draw Bottom Gate Strap	false
topgatestrwidth		0.12
topgatestrext		1
botgatestrwidth		0.12
botgatestrext		1
topgcut	Draw Top Poly Cut	false
botgcut	Draw Bottom Poly Cut	false
connectsource	Connect all Sources together	false
connectdrain	Connect all Drains together	false

Table 1: Summary of Transistor Parameters

4 API Documentation

The following section documents all available API functions for layout creation and manipulation. At the current time, it is far from complete. More up-to-date API documentation can be found by using opc directly with the command-line options `--api-search`, `--api-list` and `--api-help`.

`geometry.rectanglebltr(cell, layer, bl, tr)`

Create a rectangular shape with the given corner points in cell

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the rectangle is created
`layer`	generics	Layer of the generated rectangular shape
`bl`	point	Bottom-left point of the generated rectangular shape
`tr`	point	Top-right point of the generated rectangular shape

`geometry.rectanglepoints(cell, layer, pt1, pt2)`

Create a rectangular shape with the given corner points in cell. Similar to geometry.rectanglebltr, but any of the corner points can be given in any order

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the rectangle is created
`layer`	generics	Layer of the generated rectangular shape
`pt1`	point	First corner point of the generated rectangular shape
`pt2`	point	Second corner point of the generated rectangular shape

`geometry.rectanglepath(cell, layer, pt1, pt2, width, extension)`

Create a rectangular shape that is defined by its path-like endpoints. This function behaves like geometry.path, but takes only two points, not a list of points. This function likely will be removed in the future, use geometry.rectanglebltr or geometry.rectanglepoints

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the rectangle is created
`layer`	generics	Layer of the generated rectangular shape
`pt1`	point	First path point of the generated rectangular shape
`pt2`	point	Second path point of the generated rectangular shape
`width`	integer	Width of the path-like shape
`extension`	table	optional table argument containing the start/end extensions

`geometry.rectanglearray(cell, layer, width, height, xshift, yshift, xrep, yrep, xpitch, ypitch)`

Create an array of rectangles with the given width, height, repetition and pitch in cell

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the rectangle is created
`layer`	generics	Layer of the generated rectangular shape
`width`	integer	Width of the generated rectangular shape
`height`	integer	Height of the generated rectangular shape
`xshift`	integer	Number of repetitions in x direction. The Rectangles are shifted so that an equal number is above and below
`yshift`	integer	Number of repetitions in y direction. The Rectangles are shifted so that an equal number is above and below
`xrep`	integer	Number of repetitions in x direction. The Rectangles are shifted so that an equal number is above and below
`yrep`	integer	Number of repetitions in y direction. The Rectangles are shifted so that an equal number is above and below
`xpitch`	integer	Pitch in x direction, used for repetition in x
`ypitch`	integer	Pitch in y direction, used for repetition in y

`geometry.rectanglevlines(cell, layer, pt1, pt2, numlines, ratio)`

Fill a rectangular area with vertical lines with a given ratio between width and spacing

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the rectangle is created
`layer`	generics	Layer of the generated rectangular shape
`pt1`	point	First corner point of the target area
`pt2`	point	Second corner point of the target area
`numlines`	integer	Number of lines to be generated
`ratio`	number	Ratio between width and spacing of lines

`geometry.rectanglehlines(cell, layer, pt1, pt2, numlines, ratio)`

Fill a rectangular area with horizontal lines with a given ratio between width and spacing

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the rectangle is created
`layer`	generics	Layer of the generated rectangular shape
`pt1`	point	First corner point of the target area
`pt2`	point	Second corner point of the target area
`numlines`	integer	Number of lines to be generated
`ratio`	number	Ratio between width and spacing of lines

`geometry.rectangle_fill_in_boundary(cell, layer, width, height, xpitch, ypitch, xstartshift, ystartshift, boundary, excludes)`

Fill a given boundary (a polygon) with rectangles of a given width and height. If given, the rectangles are not placed in the regions defined by the exclude rectangles. The excludes table should contain polygons

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the rectangle is created
`layer`	generics	Layer of the generated rectangular shape
`width`	integer	Width of the rectangles
`height`	integer	Height of the rectangles
`xpitch`	integer	Pitch in x-direction
`ypitch`	integer	Pitch in y-direction
`xstartshift`	integer	Shift the start of the rectangle placment algorithm in x-direction
`ystartshift`	integer	Shift the start of the rectangle placment algorithm in y-direction
`boundary`	pointlist	List of points defining fill boundary (a polygon)
`excludes`	table	Collection of excludes (polygons)

`geometry.polygon(cell, layer, pts)`

Create a polygon shape with the given points in cell

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the polygon is created
`layer`	generics	Layer of the generated rectangular shape
`pts`	pointlist	List of points that make up the polygon

`geometry.path(cell, layer, pts, width, extension)`

Create a path shape with the given points and width in cell

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the path is created
`layer`	generics	Layer of the generated rectangular shape
`pts`	pointlist	List of points where the path passes through
`width`	integer	width of the path. Must be even
`extension`	table	optional table argument containing the start/end extensions

`geometry.path_manhatten(cell, layer, pts, width, extension)`

Create a manhatten path shape with the given points and width in cell. This only allows vertical or horizontal movements

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the path is created
`layer`	generics	Layer of the generated rectangular shape
`pts`	pointlist	List of points where the path passes through
`width`	integer	width of the path. Must be even
`extension`	table	optional table argument containing the start/end extensions

`geometry.path_2x(cell, layer, ptstart, ptend, width)`

Create a path that starts at ptstart and ends at ptend by moving first in x direction, then in y-direction (similar to an 'L')

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the path is created
`layer`	generics	Layer of the generated rectangular shape
`ptstart`	point	Start point of the path
`ptend`	point	End point of the path
`width`	integer	width of the path. Must be even

`geometry.path_2y(cell, layer, ptstart, ptend, width)`

Create a path that starts at ptstart and ends at ptend by moving first in y direction, then in x-direction (similar to an 'T')

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the path is created
`layer`	generics	Layer of the generated rectangular shape
`ptstart`	point	Start point of the path
`ptend`	point	End point of the path
`width`	integer	width of the path. Must be even

`geometry.path_cshape(cell, layer, ptstart, ptend, ptoffset, width)`

Create a path shape that starts and ends at the start and end point, respectively and passes through the offset point. Only the x-coordinate of the offset point is taken, creating a shape resembling a (possibly inverter) 'C'

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the path is created
`layer`	generics	Layer of the generated rectangular shape
`ptstart`	point	Start point of the path
`ptend`	point	End point of the path
`ptoffset`	point	Offset point
`width`	integer	width of the path. Must be even

`geometry.path_ushape(cell, layer, ptstart, ptend, ptoffset, width)`

Create a path shape that starts and ends at the start and end point, respectively and passes through the offset point. Only the y-coordinate of the offset point is taken, creating a shape resembling a (possibly inverter) 'U'

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the path is created
`layer`	generics	Layer of the generated rectangular shape
`ptstart`	point	Start point of the path
`ptend`	point	End point of the path
`ptoffset`	point	Offset point
`width`	integer	width of the path. Must be even

`geometry.path_points_xy(ptstart, pts)`

Create a point list for use in geometry.path that contains only horizontal and vertical movements based on a list of points or scalars. This function only creates the resulting list of points, no shapes by itself. A movement can be a point, in which case two resulting movements are created: first x, than y (or vice versa, depending on the current state). A scalar movement moves relatively by that amount (in x or y, again depending on the state) This function does the same as geometry.path_points_yx, but starts in x-direction

Parameters:

Parameter	Type	Explanation
`ptstart`	point	Start point of the path
`pts`	pointlist	List of points or scalars

`geometry.path_points_yx(ptstart, pts)`

Create a point list for use in geometry.path that contains only horizontal and vertical movements based on a list of points or scalars. This function only creates the resulting list of points, no shapes by itself. A movement can be a point, in which case two resulting movements are created: first x, than y (or vice versa, depending on the current state). A scalar movement moves relatively by that amount (in x or y, again depending on the state) This function does the same as geometry.path_points_xy, but starts in y-direction

Parameters:

Parameter	Type	Explanation
`ptstart`	point	Start point of the path
`pts`	pointlist	List of points or scalars

`geometry.viabltr(cell, firstmetal, lastmetal, bl, tr)`

Create vias (single or stack) in a rectangular area with the given corner points in cell

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the via is created
`firstmetal`	integer	Number of the first metal. Negative values are possible
`lastmetal`	integer	Number of the last metal. Negative values are possible
`bl`	point	Bottom-left point of the generated rectangular shape
`tr`	point	Top-right point of the generated rectangular shape

`geometry.viabarebltr(cell, firstmetal, lastmetal, bl, tr)`

Create vias (single or stack) in a rectangular area with the given corner points in cell. This function is like viabltr, but no metals are drawn

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the via is created
`firstmetal`	integer	Number of the first metal. Negative values are possible
`lastmetal`	integer	Number of the last metal. Negative values are possible
`bl`	point	Bottom-left point of the generated rectangular shape
`tr`	point	Top-right point of the generated rectangular shape

`geometry.viabltr_xcontinuous(cell, firstmetal, lastmetal, bl, tr)`

Create vias (single or stack) in a rectangular area with the given corner points in cell. This function creates vias that can be abutted in x-direction. For this, the space between cuts and the surroundings are equalized

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the via is created
`firstmetal`	integer	Number of the first metal. Negative values are possible
`lastmetal`	integer	Number of the last metal. Negative values are possible
`bl`	point	Bottom-left point of the generated rectangular shape
`tr`	point	Top-right point of the generated rectangular shape

`geometry.viabltr_ycontinuous(cell, firstmetal, lastmetal, bl, tr)`

Create vias (single or stack) in a rectangular area with the given corner points in cell. This function creates vias that can be abutted in y-direction. For this, the space between cuts and the surroundings are equalized

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the via is created
`firstmetal`	integer	Number of the first metal. Negative values are possible
`lastmetal`	integer	Number of the last metal. Negative values are possible
`bl`	point	Bottom-left point of the generated rectangular shape
`tr`	point	Top-right point of the generated rectangular shape

`geometry.viabltr_continuous(cell, firstmetal, lastmetal, bl, tr)`

Create vias (single or stack) in a rectangular area with the given corner points in cell. This function creates vias that can be abutted in both x- and y-direction. For this, the space between cuts and the surroundings are equalized

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the via is created
`firstmetal`	integer	Number of the first metal. Negative values are possible
`lastmetal`	integer	Number of the last metal. Negative values are possible
`bl`	point	Bottom-left point of the generated rectangular shape
`tr`	point	Top-right point of the generated rectangular shape

`geometry.contactbltr(cell, layer, bl, tr)`

Create contacts in a rectangular area with the given corner points in cell

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the contact is created
`layer`	string	Identifier of the contact type. Possible values: ’gate’, ’active’, ’sourcedrain’
`bl`	point	Bottom-left point of the generated rectangular shape
`tr`	point	Top-right point of the generated rectangular shape

`geometry.contactbarebltr(cell, layer, bl, tr)`

Create contacts in a rectangular area with the given corner points in cell. This function creates ’bare’ contacts, so only the cut layers, no surrouning metals or semi-conductor layers

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the contact is created
`layer`	string	Identifier of the contact type. Possible values: 'gate', 'active', 'sourcedrain'
`bl`	point	Bottom-left point of the generated rectangular shape
`tr`	point	Top-right point of the generated rectangular shape

`geometry.cross(cell, layer, width, height, crosssize)`

Create a cross shape in the given cell. The cross is made up by two overlapping rectangles in horizontal and in vertical direction.

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the cross is created
`layer`	generics	Layer of the generated cross shape
`width`	integer	Width of the generated cross shape
`height`	integer	Height of the generated cross shape
`crosssize`	integer	Cross size of the generated cross shape (the 'width' of the rectangles making up the cross)

`geometry.unequal_ring_pts(cell, layer, outerbl, outertr, innerbl, innertr)`

Create a ring shape with unequal ring widths in the given cell, defined by the corner points

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the ring is created
`layer`	generics	Layer of the generated ring shape
`outerbl`	point	Outer lower-left corner of the generated ring shape
`outertr`	point	Outer upper-right corner of the generated ring shape
`innerbl`	point	Inner lower-left corner of the generated ring shape
`innertr`	point	Inner upper-right corner of the generated ring shape

`geometry.unequal_ring(cell, layer, center, width, height, leftringwidth, rightringwidth, topringwidth, bottomringwidth)`

Create a ring shape with unequal ring widths in the given cell

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the ring is created
`layer`	generics	Layer of the generated ring shape
`center`	point	Center of the generated ring shape
`width`	integer	Width of the generated ring shape
`height`	integer	Height of the generated ring shape
`leftringwidth`	integer	Left ring width of the generated ring shape (the 'width' of the path making up the left part of the ring)
`rightringwidth`	integer	Right ring width of the generated ring shape (the 'width' of the path making up the right part of the ring)
`topringwidth`	integer	Top ring width of the generated ring shape (the 'width' of the path making up the top part of the ring)
`bottomringwidth`	integer	Bottom ring width of the generated ring shape (the 'width' of the path making up the bottom part of the ring)

`geometry.ring(cell, layer, center, width, height, ringwidth)`

Create a ring shape width equal ring widths in the given cell. Like geometry.unequal_ring, but all widths are the same

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the ring is created
`layer`	generics	Layer of the generated ring shape
`center`	point	Center of the generated ring shape
`width`	integer	Width of the generated ring shape
`height`	integer	Height of the generated ring shape
`ringwidth`	integer	Ring width of the generated ring shape (the 'width' of the path making up the ring)

`geometry.curve(cell, layer, origin, segments, grid, allow45)`

Create a curve shape width in the given cell. Segments must be added for a curve to be meaningful. See the functions for adding curve segments: curve.lineto, curve.arcto and curve.cubicto

Parameters:

Parameter	Type	Explanation
`cell`	object	Object in which the ring is created
`layer`	generics	Layer of the generated ring shape
`origin`	point	Start point of the curve
`segments`	table	Table of curve segments
`grid`	integer	Grid for rasterization of the curve
`allow45`	boolean	Start point of the curve

`set(...)`

define a set of possible values that a parameter can take. Only useful within a parameter definition of a pcell

Parameters:

Parameter	Type	Explanation
`...`	...	variable number of arguments, usually strings or integers

`interval(lower, upper)`

define an interval of possible values that a parameter can take. Only useful within a parameter definition of a pcell

Parameters:

Parameter	Type	Explanation
`lower`	integer	lower (inklusive) bound of the interval
`upper`	integer	upper (inklusive) bound of the interval

`even()`

define that a parameter must be even. Only useful within a parameter definition of a pcell

Parameters:

Parameter	Type	Explanation

`odd()`

define that a parameter must be odd. Only useful within a parameter definition of a pcell

Parameters:

Parameter	Type	Explanation

`positive()`

define that a parameter must be positive. Only useful within a parameter definition of a pcell

Parameters:

Parameter	Type	Explanation

`negative()`

define that a parameter must be negative. Only useful within a parameter definition of a pcell

Parameters:

Parameter	Type	Explanation

`pcell.set_property(property, value)`

set a property of a pcell. Not many properties are supported currently, so this function is very rarely used. The base cell of the standard cell library uses it to be hidden, but that's the only current use

Parameters:

Parameter	Type	Explanation
`property`	string	property to set
`value`	any	value of the property

`pcell.add_parameter(name, defaultvalue, opt)`

add a parameter to a pcell definition. Must be called in parameters(). The parameter options table can contain the following fields: 'argtype': (type of the parameter, usually deduced from the default value), 'posvals': possible parameter values, see functions 'even', 'odd', 'interval', 'positive', 'negative' and 'set'; 'follow': copy the values from the followed parameter to this one if not explicitly specified; 'readonly': make parameter readonly

Parameters:

Parameter	Type	Explanation
`name`	string	parameter name
`defaultvalue`	any	default parameter value (can be any lua type)
`opt`	table	options table

`pcell.add_parameters(args)`

add multiple parameters to a cell. Internally, this calls pcell.add_parameter, so this function is merely a shorthand for multiple calls to pcell.parameter. Hint for the usage: in lua tables, a trailing comma after the last entry is explicitely allowed. However, this is a variable number of arguments for a function call, where the list has to be well-defined. A common error is a trailing comma after the last entry

Parameters:

Parameter	Type	Explanation
`args`	...	argument list of single parameter entries

`pcell.get_parameters(cellname)`

access the (updated) parameter values of another cell

Parameters:

Parameter	Type	Explanation
`cellname`	string	cellname of the cell whose parameters should be queried

`pcell.push_overwrites(cellname, parameters)`

overwrite parameters of other cells. This works across pcell limits and can be called before pcell layouts are created. This also affects cells that are created in sub-cells. This works like a stack (one stack per cell), so it can be applied multiple times

Parameters:

Parameter	Type	Explanation
`cellname`	string	cellname of the to-be-overwritten cell
`parameters`	table	table with key-value pairs

`pcell.pop_overwrites(cellname)`

pop one entry of overwrites from the overwrite stack

Parameters:

Parameter	Type	Explanation
`cellname`	string	cellname of the overwrite stack

`pcell.check_expression(expression, message)`

check valid parameter values with expressions. If parameter values depend on some other parameter or the posval function of parameter definitions do not offer enough flexibility, parameters can be checked with arbitrary lua expressions. This function must be called in parameters()

Parameters:

Parameter	Type	Explanation
`expression`	string	expression to check
`message`	string	custom message which is displayed if the expression could not be satisfied

`pcell.create_layout(cellname, objectname, parameters)`

Create a layout based on a parametric cell

Parameters:

Parameter	Type	Explanation
`cellname`	string	cellname of the to-be-generated layout cell in the form libname/-cellname
`objectname`	string	name of the to-be-generated object. This name will be used as identifier in exports that support hierarchies (e.g. GDSII, SKILL)
`parameters`	table	a table with key-value pairs to be used for the layout pcell. The parameter must exist in the pcell, otherwise this triggers an error

`pcell.create_layout_env(cellname, objectname, parameters, environment)`

Create a layout based on a parametric cell with a given cell environment

Parameters:

Parameter	Type	Explanation
`cellname`	string	cellname of the to-be-generated layout cell in the form libname/-cellname
`objectname`	string	name of the to-be-generated object. This name will be used as identifier in exports that support hierarchies (e.g. GDSII, SKILL)
`parameters`	table	a table with key-value pairs to be used for the layout pcell. The parameter must exist in the pcell, otherwise this triggers an error
`environment`	table	a table containing the environment for all cells called from this cell. The content of the environment can contain anything and is defined by the cells. It is useful in order to pass a set of common options to multiple cells

`tech.get_dimension(property)`

Get critical technology dimensions such as minimum metal width. Predominantly used in pcell parameter definitions, but not necessarily restricted to that. There is a small set of technology properties that are used in the standard opc cells, but there is currently no proper definitions of the supported fields. See basic/mosfet and basic/cmos for examples

Parameters:

Parameter	Type	Explanation
`property`	string	technology property name

`tech.has_layer(layer)`

Check if the chosen technology supports a certain layer

Parameters:

Parameter	Type	Explanation
`layer`	generics	generic layer which should be checked

`tech.resolve_metal(index)`

resolve negative metal indices to their 'real' value (e.g. in a metal stack with five metals -1 becomes 5, -3 becomes 3). This function does not do anything if the index is positive

Parameters:

Parameter	Type	Explanation
`index`	integer	metal index to be resolved

`placement.create_floorplan_aspectratio(instances, utilization, aspectration)` create a floorplan configuration based on utilization and an aspectratio. The 'instances' table is the result of parsing and processing verilog netlists. This function is intended to be called in a place-and-route-script for –import-verilog

Parameters:

Parameter	Type	Explanation
`instances`	table	instances table
`utilization`	number	utilization factor, must be between 0 and 1
`aspectration`	number	aspectratio (width / height) of the floorplan

`placement.create_floorplan_fixed_rows(instances, utilization, rows)`

create a floorplan configuration based on utilization and a fixed number of rows. The 'instances' table is the result of parsing and processing verilog netlists. This function is intended to be called in a place-and-route-script for –import-verilog

Parameters:

Parameter	Type	Explanation
`instances`	table	instances table
`utilization`	number	utilization factor, must be between 0 and 1
`rows`	integer	number of rows

`placement.optimize(instances, nets, floorplan)`

minimize wire length by optimizing the placement of the instances by a simulated annealing algorithm. This function returns a table with the rows and columns of the placement of the instances. It is intended to be called in a place-and-route-script for –import-verilog

Parameters:

Parameter	Type	Explanation
`instances`	table	instances table
`nets`	table	nets table
`floorplan`	table	floorplan configuration

`placement.manual(instances, plan)`

create a placement of instances manually. This function expects a row-column table with all instance names. Thus the instance names must match the ones found in the instances table (from the verilog netlist). This function then updates all required references in the row-column table, that are needed for further processing (e.g. routing). This function is useful for small designs, especially in a hierarchical flow

Parameters:

Parameter	Type	Explanation
`instances`	table	instances table
`plan`	table	row-column table

`placement.insert_filler_names(rows, width)`

equalize placement rows by inserting fillers in every row.The method tries to equalize spacing between cells.This function is intended to be called in a place-and-route-script for –import-verilog

Parameters:

Parameter	Type	Explanation
`rows`	table	placement rows table
`width`	integer	width as multiple of transistor gates. Must be equal to or larger than every row

`placement.create_reference_rows(cellnames, xpitch)`

prepare a row placement table for further placement functions by parsing a definition given in ’cellnames’.This table contains the individual rows of the placment, which every row consiting of individual cells.Cell entries can either be given by just the name of the standard cell (the ’reference’) or the instance name (’instance’) and the reference name (’reference’)This function is meant to be used in pcell definitions

Parameters:

Parameter	Type	Explanation
`cellnames`	table	row placement table with cellnames
`xpitch`	integer	minimum cell pitch in x direction

`placement.digital()`

Parameters:

Parameter	Type	Explanation

`placement.rowwise()`

Parameters:

Parameter	Type	Explanation

`routing.legalize()`

Parameters:

Parameter	Type	Explanation

`routing.route()`

Parameters:

Parameter	Type	Explanation

`curve.lineto(point)`

create a line segment for a curve

Parameters:

Parameter	Type	Explanation
`point`	point	destination point of the line segment

`curve.arcto(startangle, endangle, radius, clockwise)`

create an arc segment for a curve

Parameters:

Parameter	Type	Explanation
`startangle`	number	start angle of the line segment
`endangle`	number	end angle of the line segment
`radius`	integer	radius of the line segment
`clockwise`	boolean	flag if arc is drawn clock-wise or counter-clock-wise

`curve.cubicto(ctp1, ctp2, endpt)`

create a cubic bezier segment for a curve

Parameters:

Parameter	Type	Explanation
`ctp1`	point	first control point
`ctp2`	point	second control point
`endpt`	point	destination point of the cubic bezier segment

`object.create(cellname)`

create a new object. A name must be given. Hierarchical exports use this name to identify layout cells and no checks for duplication are done. Therefore the user must make sure that every name is unique. Note that this will probably change in the future

Parameters:

Parameter	Type	Explanation
`cellname`	string	the name of the layout cell

`object.copy(cell)`

copy an object

Parameters:

Parameter	Type	Explanation
`cell`	object	Object to copy

`object.exchange(cell, othercell)`

Take over internal state of the other object, effectively making this the main cell. The object handle to 'othercell' must not be used afterwards as this object is destroyed. This function is only really useful in cells that act as a parameter wrapper for other cells (e.g. dffpq -> dff)

Parameters:

Parameter	Type	Explanation
`cell`	object	Object which should take over the other object
`othercell`	object	Object which should be taken over. The object handle must not be used after this operation

`object.add_anchor(cell, name, where)`

add an anchor to an object

Parameters:

Parameter	Type	Explanation
`cell`	object	object to which an anchor should be added
`name`	string	name of the anchor
`where`	point	location of the anchor

`object.add_area_anchor_bltr(cell, name, bl, tr)`

Similar to add_area_anchor, but takes to lower-left and upper-right corner points of the rectangular area

Parameters:

Parameter	Type	Explanation
`cell`	object	object to which an anchor should be added
`name`	string	name of the anchor
`bl`	point	bottom-left point of the rectangular area
`tr`	point	bottom-left point of the rectangular area

`object.get_anchor(cell, anchorname)`

Retrieve an anchor from a cell. This function returns a point that contains the position of the specified anchor, corrected by the cell transformation. A non-existing anchor is an error

Parameters:

Parameter	Type	Explanation
`cell`	object	object to get an anchor from
`anchorname`	string	name of the anchor

`object.get_alignment_anchor(cell, anchorname)`

Retrieve an alignemtn anchor from a cell. These anchors are the defining points of the alignment box. Valid anchor names are ’outerbl’, ’outerbr’, ’outertl’, ’outertr’, ’innerbl’, ’innerbr’, ’innertl’ and ’innertr’. This function returns a point that contains the position of the specified anchor, corrected by the cell transformation. A non-existing anchor is an error

Parameters:

Parameter	Type	Explanation
`cell`	object	object to get an anchor from
`anchorname`	string	name of the alignment anchor

`object.get_area_anchor(cell, anchorname)`

Retrieve an area anchor from a cell. This function returns a table containing two points (bl (bottom-left) and tr (top-right)) that contain the position of the specified area anchor, corrected by the cell transformation. A non-existing anchor is an error

Parameters:

Parameter	Type	Explanation
`cell`	object	object to get an anchor from
`anchorname`	string	name of the anchor

`object.get_array_anchor(cell, xindex, yindex, anchorname)`

Like object.get_anchor, but works on child arrays. The first two argument are the x- and the y-index (starting at 1, 1). Accessing an array anchor of a non-array object is an error

Parameters:

Parameter	Type	Explanation
`cell`	object	object to get an anchor from
`xindex`	integer	x-index
`yindex`	integer	y-index
`anchorname`	string	name of the anchor

`object.get_array_area_anchor(cell, xindex, yindex, anchorname)`

Like object.get_area_anchor, but works on child arrays. The first two argument are the x- and the y-index (starting at 1, 1). Accessing an array anchor of a non-array object is an error

Parameters:

Parameter	Type	Explanation
`cell`	object	object to get an anchor from
`xindex`	integer	x-index
`yindex`	integer	y-index
`anchorname`	string	name of the anchor

`object.add_port(cell, name, layer, where)`

add a port to a cell. Works like add_anchor, but additionally a layer is expected

Parameters:

Parameter	Type	Explanation
`cell`	object	object to which a port should be added
`name`	string	name of the port
`layer`	generics	layer of the port
`where`	point	location of the port

`object.add_bus_port(cell, name, layer, where, startindex, endindex, xpitch, ypitch )`

add a bus port (multiple ports like vout[0:4]) to a cell. The port expression is portname[startindex:endindex] and portname[i] is placed at 'where' with an offset of ((i - 1) * xpitch, (i - 1) * ypitch)

Parameters:

Parameter	Type	Explanation
`cell`	object	object to which a port should be added
`name`	string	base name of the port
`layer`	generics	layer of the port
`where`	point	location of the port
`startindex`	integer	start index of the bus port
`endindex`	integer	end index of the bus port
`xpitch`	integer	pitch in x direction
`ypitch`	integer	pitch in y direction

`object.get_ports(cell)`

return a table which contains key-value pairs with all ports of a cell. The key is the portname, the value the corresponding point.

Parameters:

Parameter	Type	Explanation
`cell`	object	object to get the ports from

`object.set_alignment_box(cell, bl, tr)`

set the alignment box of an object. Overwrites any previous existing alignment boxes

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to add the alignment box to
`bl`	point	bottom-left corner of alignment box
`tr`	point	top-right corner of alignment box

`object.inherit_alignment_box(cell, othercell)`

inherit the alignment box from another cell. This EXPANDS the current alignment box, if any is present. This means that this function can be called multiple times with different objects to establish an overall alignment box

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to add the alignment box to
`othercell`	object	cell to inherit the alignment box from

`object.inherit_area_anchor(cell, othercell, anchorname)`

inherit an area anchor from another cell.

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to add the anchor to
`othercell`	object	cell to inherit the anchor from
`anchorname`	string	anchor name of the to-be-inherited anchor

`object.inherit_area_anchor_as(cell, othercell, anchorname, newname)`

inherit an area anchor from another cell under a different name.

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to add the anchor to
`othercell`	object	cell to inherit the anchor from
`anchorname`	string	anchor name of the to-be-inherited anchor
`newname`	string	new name of the inherited anchor

`object.inherit_boundary(cell, othercell)`

inherit the boundary from another cell.

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to add the boundar to
`othercell`	object	cell to inherit the boundary from

`object.extend_alignment_box(cell, extouterblx, extouterbly, extoutertrx, extoutertry, extinnerblx, extinnerbly, extinnertrx, extinnertry)`

extend an existing object alignment box. Takes eight values for the extension of the four corner points making up the alignment box

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to add the alignment box to
`extouterblx`	integer	extension of outer-left coordinate
`extouterbly`	integer	extension of outer-bottom coordinate
`extoutertrx`	integer	extension of outer-right coordinate
`extoutertry`	integer	extension of outer-top coordinate
`extinnerblx`	integer	extension of inner-left coordinate
`extinnerbly`	integer	extension of inner-bottom coordinate
`extinnertrx`	integer	extension of inner-right coordinate
`extinnertry`	integer	extension of inner-top coordinate

`object.width_height_alignmentbox(cell)`

get the width and the height of the alignment box. A non-existing alignment box triggers an error

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to compute width and height

`object.move_to(cell, x, y)`

move the cell to the specified coordinates (absolute movement). If x is a point, x and y are taken from this point

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be moved
`x`	integer	x coordinate (can be a point, in this case x and y are taken from this point)
`y`	integer	y coordinate

`object.reset_translation(cell)`

reset all previous translations (transformations are kept)

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be resetted

`object.translate(cell, x, y)`

translate the cell by the specified offsets (relative movement). If x is a point, x and y are taken from this point

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be translated
`x`	integer	x offset (can be a point, in this case x and y are taken from this point)
`y`	integer	y offset

`object.translate_x(cell, x)`

translate the cell by the specified x offset (relative movement).

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be translated
`x`	integer	x offset

`object.translate_y(cell, y)`

translate the cell by the specified y offset (relative movement).

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be translated
`y`	integer	y offset

`object.abut_left(cell, targercell)`

translate the cell so that its alignment box is abutted to the left of the alignment box of the specified target cell. This only changes the y coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be abutted
`targercell`	object	abutment target cell

`object.abut_right(cell, targercell)`

translate the cell so that its alignment box is abutted to the right of the alignment box of the specified target cell. This only changes the y coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be abutted
`targercell`	object	abutment target cell

`object.abut_top(cell, targercell)`

translate the cell so that its alignment box is abutted to the top of the alignment box of the specified target cell. This only changes the y coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be abutted
`targercell`	object	abutment target cell

`object.abut_bottom(cell, targercell)`

translate the cell so that its alignment box is abutted to the bottom of the alignment box of the specified target cell. This only changes the y coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be abutted
`targercell`	object	abutment target cell

`object.align_left(cell, targercell)`

translate the cell so that its alignment box is aligned to the left of the alignment box of the specified target cell. This only changes the x coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be aligned
`targercell`	object	alignment target cell

`object.align_right(cell, targercell)`

translate the cell so that its alignment box is aligned to the right of the alignment box of the specified target cell. This only changes the x coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be aligned
`targercell`	object	alignment target cell

`object.align_top(cell, targercell)`

translate the cell so that its alignment box is aligned to the top of the alignment box of the specified target cell. This only changes the y coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be aligned
`targercell`	object	alignment target cell

`object.align_bottom(cell, targercell)`

translate the cell so that its alignment box is aligned to the bottom of the alignment box of the specified target cell. This only changes the y coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be aligned
`targercell`	object	alignment target cell

`object.abut_area_anchor_left(cell, anchorname, targercell, targetanchorname)`

translate the cell so that the specified area anchor is abutted to the left of the target area anchor of the specified target cell. This only changes the x coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be moved
`anchorname`	string	abutment anchor
`targercell`	object	alignment target cell
`targetanchorname`	string	target abutment anchor

`object.abut_area_anchor_right(cell, anchorname, targercell, targetanchorname)`

translate the cell so that the specified area anchor is abutted to the right of the target area anchor of the specified target cell. This only changes the x coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be moved
`anchorname`	string	abutment anchor
`targercell`	object	alignment target cell
`targetanchorname`	string	target abutment anchor

`object.abut_area_anchor_top(cell, anchorname, targercell, targetanchorname)`

translate the cell so that the specified area anchor is abutted to the top of the target area anchor of the specified target cell. This only changes the y coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be moved
`anchorname`	string	abutment anchor
`targercell`	object	alignment target cell
`targetanchorname`	string	target abutment anchor

`object.abut_area_anchor_bottom(cell, anchorname, targercell, targetanchorname)`

translate the cell so that the specified area anchor is abutted to the bottom of the target area anchor of the specified target cell. This only changes the y coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be moved
`anchorname`	string	abutment anchor
`targercell`	object	alignment target cell
`targetanchorname`	string	target abutment anchor

`object.align_area_anchor(cell, anchorname, targercell, targetanchorname)`

translate the cell so that the specified area anchor is aligned to the target area anchor of the specified target cell. This changes both the x and the y coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be moved
`anchorname`	string	alignment anchor
`targercell`	object	alignment target cell
`targetanchorname`	string	target alignment anchor

`object.align_area_anchor_left(cell, anchorname, targercell, targetanchorname)` translate the cell so that the specified area anchor is aligned to the left of the target area anchor of the specified target cell. This only changes the x coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be moved
`anchorname`	string	alignment anchor
`targercell`	object	alignment target cell
`targetanchorname`	string	target alignment anchor

`object.align_area_anchor_right(cell, anchorname, targercell, targetanchorname)`

translate the cell so that the specified area anchor is aligned to the right of the target area anchor of the specified target cell. This only changes the x coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be moved
`anchorname`	string	alignment anchor
`targercell`	object	alignment target cell
`targetanchorname`	string	target alignment anchor

`object.align_area_anchor_top(cell, anchorname, targercell, targetanchorname)`

translate the cell so that the specified area anchor is aligned to the top of the target area anchor of the specified target cell. This only changes the y coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be moved
`anchorname`	string	alignment anchor
`targercell`	object	alignment target cell
`targetanchorname`	string	target alignment anchor

`object.align_area_anchor_bottom(cell, anchorname, targercell, targetanchorname)`

translate the cell so that the specified area anchor is aligned to the bottom of the target area anchor of the specified target cell. This only changes the y coordinate

Parameters:

Parameter	Type	Explanation
`cell`	object	cell to be moved
`anchorname`	string	alignment anchor
`targercell`	object	alignment target cell
`targetanchorname`	string	target alignment anchor

`object.mirror_at_xaxis()`

mirror the entire object at the x axis

Parameters:

Parameter	Type	Explanation

`object.mirror_at_yaxis()`

mirror the entire object at the y axis

Parameters:

Parameter	Type	Explanation

`object.mirror_at_origin()`

mirror the entire object at the origin

Parameters:

Parameter	Type	Explanation

`object.rotate_90_left()`

rotate the entire object 90 degrees counter-clockwise with respect to the origin

Parameters:

Parameter	Type	Explanation

`object.rotate_90_right()`

rotate the entire object 90 degrees clockwise with respect to the origin

Parameters:

Parameter	Type	Explanation

`object.flipx()`

flip the entire object in x direction. This is similar to mirror_at_yaxis (note the x vs. y), but is done in-place. The object is translated so that it is still in its original location. Works best on objects with an alignment box, since this is used to calculate the required translation. On other objects, this operation can be time-consuming as an accurate bounding box has to be computed. It is recommended not to use this function on objects without an alignment box because the result is not always ideal

Parameters:

Parameter	Type	Explanation

`object.flipy()`

flip the entire object in y direction. This is similar to mirror_at_xaxis (note the y vs. x), but is done in-place. The object is translated so that it is still in its original location. Works best on objects with an alignment box, since this is used to calculate the required translation. On other objects, this operation can be time-consuming as an accurate bounding box has to be computed. It is recommended not to use this function on objects without an alignment box because the result is not always ideal

Parameters:

Parameter	Type	Explanation

`object.move_point(cell, source, target)`

translate (move) the object so that the source point lies on the target. Usually the source point is an anchor of the object, but that is not a necessity. The points are just references for the delta vector and can be any points.

Parameters:

Parameter	Type	Explanation
`cell`	object	cell which should be moved
`source`	point	source point
`target`	point	target point

`object.move_point_x(cell, source, target)`

translate (move) the object so that the x-coorindate of the source point lies on the x-coordinate target. Usually the source point is an anchor of the object, but that is not a necessity. The points are just references for the delta vector and can be any points.

Parameters:

Parameter	Type	Explanation
`cell`	object	cell which should be moved
`source`	point	source point
`target`	point	target point

`object.move_point_y(cell, source, target)`

translate (move) the object so that the y-coorindate of the source point lies on the y-coordinate target. Usually the source point is an anchor of the object, but that is not a necessity. The points are just references for the delta vector and can be any points.

Parameters:

Parameter	Type	Explanation
`cell`	object	cell which should be moved
`source`	point	source point
`target`	point	target point

`object.add_child(cell, child, instname)`

Add a child object (instance) to the given cell. This make 'cell' the parent of the child (it manages its memory). This means that you should not use the original child object any more after this call (unless it is object.add_child or object.add_child_array)

Parameters:

Parameter	Type	Explanation
`cell`	object	Object to which the child is added
`child`	object	Child to add
`instname`	string	Instance name (not used by all exports)

`object.add_child_array(cell, child, instname, xrep, yrep, xpitch, ypitch)`

Add a child as an arrayed object to the given cell. The child array has xrep * yrep elements, with a pitch of xpitch and ypitch, respectively. The array grows to the upper-left, with the first placed untranslated. The pitch does not have to be explicitly given: If the child has an alignment box, the xpitch and ypitch are deferred from this box, if they are not given in the call. In this case, it is an error if no alignment box is present in child. As with object.add_child: don't use the original child object after this call unless it is object.add_child or object.add_child_array

Parameters:

Parameter	Type	Explanation
`cell`	object	Object to which the child is added
`child`	object	Child to add
`instname`	string	Instance name (not used by all exports)
`xrep`	integer	Number of repetitions in x direction
`yrep`	integer	Number of repetitions in y direction
`xpitch`	integer	Optional itch in x direction, used for repetition in x. If not given, this parameter is derived from the alignment box
`ypitch`	integer	Optional itch in y direction, used for repetition in y. If not given, this parameter is derived from the alignment box

`object.merge_into(cell, othercell)`

add all shapes and children from othercell to the cell -> 'dissolve' othercell in cell

Parameters:

Parameter	Type	Explanation
`cell`	object	Object to which the child is added
`othercell`	object	Other layout cell to be merged into the cell

`object.flatten(cell)`

resolve the cell by placing all shapes from all children in the parent cell. This operates in-place and modifies the object. Copy the cell if this is unwanted

Parameters:

Parameter	Type	Explanation
`cell`	object	Object which should be flattened

`generics.metal(index)`

create a generic layer representing a metal. Metals are identified by numeric indices, where 1 denotes the first metal, 2 the second one etc. Metals can also be identified by negative indicies, where -1 denotes the top-most metal, -2 the metal below that etc.

Parameters:

Parameter	Type	Explanation
`index`	integer	metal index

`generics.metalport(index)`

create a generic layer representing a metal port. Metals are identified by numeric indices, where 1 denotes the first metal, 2 the second one etc. Metals can also be identified by negative indicies, where -1 denotes the top-most metal, -2 the metal below that etc.

Parameters:

Parameter	Type	Explanation
`index`	integer	metal index

`generics.metalexclude(index)`

create a generic layer representing a metal exclude where automatic filling is blocked. Metals are identified by numeric indices, where 1 denotes the first metal, 2 the second one etc. Metals can also be identified by negative indicies, where -1 denotes the top-most metal, -2 the metal below that etc.

Parameters:

Parameter	Type	Explanation
`index`	integer	metal index

`generics.viacut(m1index, m2index)`

create a generic layer representing a via cut. This does not calculate the right size for the via cuts. This function is rarely used directly. Via cuts are generated by geometry.via[bltr]. If you are using this function as a user, it is likely you are doing something wrong

Parameters:

Parameter	Type	Explanation
`m1index`	integer	first metal index
`m2index`	integer	second metal index

`generics.contact(region)`

create a generic layer representing a contact. This does not calculate the right size for the contact cuts. This function is rarely used directly. Contact cuts are generated by geometry.contact[bltr]. If you are using this function as a user, it is likely you are doing something wrong

Parameters:

Parameter	Type	Explanation
`region`	string	region which should be contacted. Possible values: ”sourcedrain”, ”gate” and ”active”

`generics.oxide(index)`

create a generic layer representing a marking layer for MOSFET gate oxide thickness (e.g. for core or I/O devices)

Parameters:

Parameter	Type	Explanation
`index`	integer	oxide thickness index. Conventionally starts with 1, but depends on the technology mapping

`generics.implant(polarity)`

Create a generic layer representing MOSFET source/drain implant polarity

Parameters:

Parameter	Type	Explanation
`polarity`	string	identifier for the type (polarity) of the implant. Can be ”n” or ”p”

`generics.vthtype(index)`

Create a generic layer representing MOSFET source/drain threshold voltage marking layers

Parameters:

Parameter	Type	Explanation
`index`	integer	threshold voltage marking layer index. Conventionally starts with 1, but depends on the technology mapping

`generics.other(identifier)`

create a generic layer representing 'something else'. This is for layers that do not need special processing, such as "gate"

Parameters:

Parameter	Type	Explanation
`identifier`	string	layer identifier

`generics.otherport(identifier)`

create a generic layer representing a port for 'something else'. This is for layers that do not need special processing, such as "gate"

Parameters:

Parameter	Type	Explanation
`identifier`	string	layer identifier

`generics.special()`

Create a 'special' layer. This is used to mark certain things in layouts (usually for debugging, like anchors or alignment boxes). This is not intended to translate to any meaningful layer for fabrication

Parameters:

Parameter	Type	Explanation

`generics.premapped(name, entries)`

Create a non-generic layer from specific layer data for a certain technology. The entries table should contain one table per supported export. The supplied key-value pairs in this table must match the key-value pairs that are expected by the export

Parameters:

Parameter	Type	Explanation
`name`	string	layer name. Can be nil
`entries`	table	key-value pairs for the entries

`point.copy(point)`

copy a point. Can be used as module function or as a point method

Parameters:

Parameter	Type	Explanation
`point`	point	point which should be copied

`point.unwrap(point)`

unwrap: get the x- and y-coordinate from a point. Can be used as module function or as a point method

Parameters:

Parameter	Type	Explanation
`point`	point	point which should be unwrapped

`point.getx(point)`

get the x-coordinate from a point. Can be used as module function or as a point method

Parameters:

Parameter	Type	Explanation
`point`	point	point whose x-coordinate should be queried

`point.gety(point)`

get the y-coordinate from a point. Can be used as module function or as a point method

Parameters:

Parameter	Type	Explanation
`point`	point	point whose y-coordinate should be queried

`point.translate(point, x, y)`

translate a point in x and y. Can be used as module function or as a point method

Parameters:

Parameter	Type	Explanation
`point`	point	point to translate
`x`	integer	x delta by which the point should be translated
`y`	integer	y delta by which the point should be translated

`point.translate_x(point, x)`

translate a point in x. Can be used as module function or as a point method

Parameters:

Parameter	Type	Explanation
`point`	point	point to translate
`x`	integer	x delta by which the point should be translated

`point.translate_y(point, y)`

translate a point in y. Can be used as module function or as a point method

Parameters:

Parameter	Type	Explanation
`point`	point	point to translate
`y`	integer	y delta by which the point should be translated

`point.create(x, y)`

create a point from an x- and y-coordinate

Parameters:

Parameter	Type	Explanation
`x`	integer	x-coordinate of new point
`y`	integer	y-coordinate of new point

`point.combine_12(pt1, pt2)`

create a new point by combining the coordinates of two other points. The new point is made up by x1 and y2

Parameters:

Parameter	Type	Explanation
`pt1`	point	point for the x-coordinate of the new point
`pt2`	point	point for the y-coordinate of the new point

`point.combine_21(pt1, pt2)`

create a new point by combining the coordinates of two other points. The new point is made up by x2 and y1. This function is equivalent to combine_12 with swapped arguments

Parameters:

Parameter	Type	Explanation
`pt1`	point	point for the y-coordinate of the new point
`pt2`	point	point for the x-coordinate of the new point

`point.combine(pt1, pt2)`

combine two points into a new one by taking the arithmetic average of their coordinates, that is x = 0.5 * (x1 + x2), y = 0.5 * (y1 + y2)

Parameters:

Parameter	Type	Explanation
`pt1`	point	first point for the new point
`pt2`	point	second point for the new point

`point.xdistance(pt1, pt2)`

calculate the distance in x between two points

Parameters:

Parameter	Type	Explanation
`pt1`	point	first point for the distance
`pt2`	point	second point for the distance

`point.ydistance(pt1, pt2)`

calculate the distance in y between two points

Parameters:

Parameter	Type	Explanation
`pt1`	point	first point for the distance
`pt2`	point	second point for the distance

`point.fix(pt, grid)`

fix the x- and y-coordinate from a point on a certain grid, that is 120 would become 100 on a grid of 100. This function behaves like floor(), no rounding is done

Parameters:

Parameter	Type	Explanation
`pt`	point	point to fix to the grid
`grid`	integer	grid on which the coordinates should be fixed

`point.operator+(pt1, pt2)`

sum two points. This is the same as point.combine

Parameters:

Parameter	Type	Explanation
`pt1`	point	first point for the sum
`pt2`	point	second point for the sum

`point.operator-(pt1, pt2)`

create a new point representing the difference of two points

Parameters:

Parameter	Type	Explanation
`pt1`	point	first point for the subtraction (the minuend)
`pt2`	point	second point for the subtraction (the subtrahend)

`point.operator..(pt1, pt2)`

combine two points into a new one. Takes the x-coordinate from the first point and the y-coordinate from the second one. Equivalent to point.combine_12(pt1, pt2)

Parameters:

Parameter	Type	Explanation
`pt1`	point	point for the x-coordinate of the new point
`pt2`	point	point for the y-coordinate of the new point

`util.xmirror(pts, xcenter)`

create a copy of the points in pts (a table) with all x-coordinates mirrored with respect to xcenter

Parameters:

Parameter	Type	Explanation
`pts`	pointlist	list of points
`xcenter`	integer	mirror center

`util.ymirror(pts, ycenter)`

create a copy of the points in pts (a table) with all y-coordinates mirrored with respect to ycenter

Parameters:

Parameter	Type	Explanation
`pts`	pointlist	list of points
`ycenter`	integer	mirror center

`util.xymirror(pts, xcenter, ycenter)`

create a copy of the points in pts (a table) with all x- and y-coordinates mirrored with respect to xcenter and ycenter, respectively

Parameters:

Parameter	Type	Explanation
`pts`	pointlist	list of points
`xcenter`	integer	mirror center x-coordinate
`ycenter`	integer	mirror center y-coordinate

`util.filter_forward(pts, fun)`

iterate forward through the list of points and create a new list with points that match the predicate. The predicate function is called with every point.

Parameters:

Parameter	Type	Explanation
`pts`	pointlist	point array to append to
`fun`	function	filter function

`util.filter_backward(pts, fun)`

iterate backward through the list of points and create a new list with points that match the predicate. The predicate function is called with every point.

Parameters:

Parameter	Type	Explanation
`pts`	pointlist	point array to append to
`fun`	function	filter function

`util.merge_forwards(pts, pts2)`

append all points from pts2 to pts1. Iterate pts2 forward. Operates in-place, thus pts is modified

Parameters:

Parameter	Type	Explanation
`pts`	pointlist	point array to append to
`pts2`	pointlist	point array to append from

`util.merge_backwards(pts, pts2)`

append all points from pts2 to pts1. Iterate pts2 backwards. Operates in-place, thus pts is modified

Parameters:

Parameter	Type	Explanation
`pts`	pointlist	point array to append to
`pts2`	pointlist	point array to append from

`util.reverse(pts)`

create a copy of the point array with the order of points reversed

Parameters:

Parameter	Type	Explanation
`pts`	pointlist	point array

`util.make_insert_xy(pts, index)`

create a function that inserts points into a point array. XY mode, thus points are given as two coordinates. If an index is given, insert at that position. Mostly useful with 1 as an index or not index at all (append)

Parameters:

Parameter	Type	Explanation
`pts`	pointlist	point array
`index`	integer	optional index

`util.make_insert_pts(pts, index)`

create a function that inserts points into a point array. Point mode, thus points are given as single points. If an index is given, insert at that position. Mostly useful with 1 as an index or not index at all (append)

Parameters:

Parameter	Type	Explanation
`pts`	pointlist	point array
`index`	integer	optional index

`util.fill_all_with(num, filler)`

create an array-like table with one entry repeated N times. This is useful, for example, for specifying gate contacts for basic/cmos

Parameters:

Parameter	Type	Explanation
`num`	integer	number of repetitions
`filler`	any	value which should be repeated. Can be anything, but probably most useful with strings or numbers

`util.fill_predicate_with(num, filler, predicate, other)`

create an array-like table with two entries (total number of entries is N). This function (compared to fill_all_with, fill_odd_with and fill_even_with) allows for more complex patterns. To do this, a predicate (a function) is called on every index. If the predicate is true, the first entry is inserted, otherwise the second one. This function is useful, for example, for specifying gate contacts for basic/cmos. Counting starts at 1, so the first entry will be 'other'

Parameters:

Parameter	Type	Explanation
`num`	integer	number of repetitions
`filler`	any	value which should be repeated at even numbers. Can be anything, but probably most useful with strings or numbers
`predicate`	function	predicate which is called with every index
`other`	any	value which should be repeated at odd numbers. Can be anything, but probably most useful with strings or numbers

`util.fill_even_with(num, filler, other)`

create an array-like table with two entries repeated N / 2 times, alternating. Counting starts at 1. This is useful, for example, for specifying gate contacts for basic/cmos. Counting starts at 1, so the first entry will be 'other'

Parameters:

Parameter	Type	Explanation
`num`	integer	number of repetitions
`filler`	any	value which should be repeated at even numbers. Can be anything, but probably most useful with strings or numbers
`other`	any	value which should be repeated at odd numbers. Can be anything, but probably most useful with strings or numbers

`util.fill_odd_with(num, filler, other)`

create an array-like table with two entries repeated N / 2 times, alternating. Counting starts at 1. This is useful, for example, for specifying gate contacts for basic/cmos. Counting starts at 1, so the first entry will be 'filler'

Parameters:

Parameter	Type	Explanation
`num`	integer	number of repetitions
`filler`	any	value which should be repeated at odd numbers. Can be anything, but probably most useful with strings or numbers
`other`	any	value which should be repeated at even numbers. Can be anything, but probably most useful with strings or numbers

`util.add_options(baseoptions, additionaloptions)`

create a copy of the baseoptions table and add all key-value pairs found in additionaloptions. This function clones baseoptions so the original is not altered. This copy is flat, so only the first-level elements are copied (e.g. tables will reference the same object). This function is useful to modify a set of base options for several devices such as mosfets, which only differ in a few options

Parameters:

Parameter	Type	Explanation
`baseoptions`	table	base options
`additionaloptions`	table	additional options

`util.ratio_split_even(value, ratio)`

create two values that sum up to the input value and have the specified ratio. The values are adjusted so that both of them are even, slightly changing the ratio. The input value must be even

Parameters:

Parameter	Type	Explanation
`value`	integer	value for division
`ratio`	number	target ratio of the two result values

`enable(bool, value)`

multiply a value with 1 or 0, depending on a boolean parameter. Essentially val * (bool and 1 or 0)

Parameters:

Parameter	Type	Explanation
`bool`	boolean	boolean for enable/disable
`value`	number	value to be enabled/disabled

`evenodddiv2(value)`

divide a value by 2. If it is odd, return floor(val / 2) and ceil(val / 2), otherwise return val / 2

Parameters:

Parameter	Type	Explanation
`value`	integer	value to divide

`divevenup(value, div)`

approximately divide a value by the divisor, so that the result is even. If this can't be achieved with the original value, increment it until it works

Parameters:

Parameter	Type	Explanation
`value`	integer	value to divide
`div`	integer	divisor

`divevendown(value, div)`

approximately divide a value by the divisor, so that the result is even. If this can't be achieved with the original value, decrement it until it works

Parameters:

Parameter	Type	Explanation
`value`	integer	value to divide
`div`	integer	divisor

`dprint(...)`

debug print. Works like regular print (which is not available in pcell definitions). Only prints something when opc is called with –enable-dprint

Parameters:

Parameter	Type	Explanation
`...`	...	variable arguments that should be printed

4.1 Object Module

4.2 Shape Module

4.3 Pointarray Module

4.4 Point Module