# OpenPCells

**PCell Design Guide and API**

Patrick Kurth

October 27, 2022

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 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.rectangle(obj, generics.metal(1), _P.width, _P.height)
end
```

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

```lua
function parameters()
```

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

```lua
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.rectangle(obj, generics.metal(1), _P.width, _P.height)
```

There is a lot going on here: We use the `geometry.rectangle` function to create a rectangle with with and height (third and fourth argument). 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:

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

## 1.2 Array of Rectangles

Now that we know how to create a rectangle, we want to create an entire array, that is a rectangular area made up of several individual rectangles. This could be used for example as filling. We will setup the cell exactly as before, we only have to add two new parameters: the repetition and the pitch (we will start with quadratic arrays with equal pitch in both directions):

[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.

```
function parameters()
    pcell.add_parameters(
        { "width",  100 },
        { "height", 100 },
        { "pitch",  200 },
        { "rep",     10 }
    )
end
```

The default arguments are 200 for the pitch and 10 for the number of repetitions, which creates a ten-by-ten array of rectangles with a spacing of 100 and a width and height of 100. Again, remember that we work in nanometers here.

For the repetition we could use a loop to create the individual rectangles, but we also have to move them to their correct position. The `geometry.rectangle` function takes a shift in x and y direction as fifth and sixth argument. We have to calculate the correct offset, which results in:

```
for x = 1, _P.rep do
    for y = 1, _P.rep do
        local o = geometry.rectangle(
            obj,
            generics.metal(1), _P.width, _P.height,
            (x - 1) * _P.pitch - (_P.rep - 1) * _P.pitch / 2,
            (y - 1) * _P.pitch - (_P.rep - 1) * _P.pitch / 2
        )
    end
end
```

This loop is cumbersome as it has to be set up and furthermore the calculation of the offset for the rectangles is done by hand. The function `geometry.rectangle` already can do what we need, as it can take the repetition and pitch in x and y as additional arguments. This places repeated copies of a template rectangle (with width and height) with their center in the origin. The full interface of `geometry.rectangle` is: With it, we can replace the whole loop construct with:

```
geometry.rectangle(obj,
    generics.metal(1), _P.width, _P.height,
    0, 0, -- xshift and yshift
    _P.rep, _P.rep, _P.pitch, _P.pitch
)
```

With this, the whole cell looks like this:

```
function parameters()
    pcell.add_parameters(
        { "width",  100 },
        { "height", 100 },
        { "pitch",  200 },
        { "rep",     10 }
    )
end
```

```lua
function layout(obj, _P)
    -- first naive attempt (don't use!)
    for x = 1, _P.rep do
        for y = 1, _P.rep do
            local o = geometry.rectangle(
                obj,
                generics.metal(1), _P.width, _P.height,
                (x - 1) * _P.pitch - (_P.rep - 1) * _P.pitch / 2,
                (y - 1) * _P.pitch - (_P.rep - 1) * _P.pitch / 2
            )
        end
    end

    -- better approach
    geometry.rectangle(obj,
        generics.metal(1), _P.width, _P.height,
        0, 0, -- xshift and yshift
        _P.rep, _P.rep, _P.pitch, _P.pitch
    )
end
```

Now you already now how to create simple rectangles and rectangle arrays 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. We will talk about that in the remaining cell tutorials.

## 1.3 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:

```lua
function parameters()
    pcell.add_parameters(
        { "fingers(Number of Fingers)", 5 },
        { "fwidth(Finger Width)",     100 },
        { "fspace(Finger Spacing)",   100 },
        { "fheight(Finger Height)",  1000 },
        { "foffset(Finger Offset)",   100 },
        { "rwidth(Rail Width)",       200 },
        { "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()` we loop over all metals to draw the fingers. We don't have to create every finger separately, with `geometry.rectangle` and the multiple capabilities this becomes very simple. Since the upper and lower fingers are one-off and `geometry.rectangle` centeres all objects, we only have to move them a little bit up/down. This is done with the corresponding arguments to `geometry.rectangle` (note that // denotes floor division in lua):

```
-- fingers
for i = _P.firstmetal, _P.lastmetal do
    geometry.rectangle(
        momcap, generics.metal(i),
        _P.fwidth, _P.fheight, -- width and height
        0, -_P.foffset / 2, -- xshift and yshift
        _P.fingers // 2, 1, 2 * (_P.fwidth + _P.fspace), 0 --
            repetition
    )
    geometry.rectangle(
        momcap, generics.metal(i),
        _P.fwidth, _P.fheight, -- width and height
        0, _P.foffset / 2, -- xshift and yshift
        _P.fingers // 2 + 1, 1, 2 * (_P.fwidth + _P.fspace), 0 --
            repetition
    )
end
```

We create two arrays of fingers, one for the "upper plate", one for the "lower plate". All fingers have the same width, height and pitch. For the upper plate, we use one more finger, the placement in `geometry.rectangle` automatically arranges them centered, so that this "just works". The `ypitch` for `geometry.rectangle` is 0, which is ok since we only have a `yrep` of 1.

The rails connecting the fingers are created in a similar manner:

```
-- rails
for i = _P.firstmetal, _P.lastmetal do
    geometry.rectangle(
        momcap, generics.metal(i),
        (_P.fingers + 1) * (_P.fwidth + _P.fspace), _P.rwidth,
        0, 0,
        1, 2, 0, _P.foffset + _P.fheight + _P.rwidth
    )
end
```

What remains is the drawing of the vias between the metals. For this we introduce a new `geometry` function: `geometry.via`. 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.via` takes almost the same arguments as `geometry.rectangle`. Only the metal layer is changed into two indices for the first and the last metal of the stack. This means that `geometry.via` 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:

```
for i = _P.firstmetal, _P.lastmetal do
    geometry.via(
        momcap, _P.firstmetal, _P.lastmetal,
        (_P.fingers + 1) * (_P.fwidth + _P.fspace), _P.rwidth,
        0, 0,
        1, 2, 0, _P.foffset + _P.fheight + _P.rwidth
    )
end
```

With this the pcell is finished. 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.

## 1.4 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 point lists and how to draw paths.

The inductor 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 pathpts = {}
local prepend = util.make_insert_xy(pathpts, 1)
local append = util.make_insert_xy(pathpts)
```

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:

```
append(-r + _scale_tanpi8(_P.width / 2),  sign * radius)
append(-r,  sign * radius)
append(-radius,  sign * r)
append(-radius, -sign * r)
append(-r, -sign * radius)
append(-r + _scale_tanpi8(_P.width / 2), -sign * radius)
```

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):

```
geometry.path(inductor, mainmetal, pathpts, _P.width, true)
geometry.path(inductor, mainmetal, util.xmirror(pathpts), _P.width,
    true)
```

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

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

## 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. 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. We can see a simple example of this in `analog/ringoscillator.lua`:

```lua
local inverters = {}
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[2]. 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.

## 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:

```lua
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.

[2]This instance name is not supported by all exports. For example, GDSII has no notion of an instance name

### 3.1 Transistor

The transistor 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

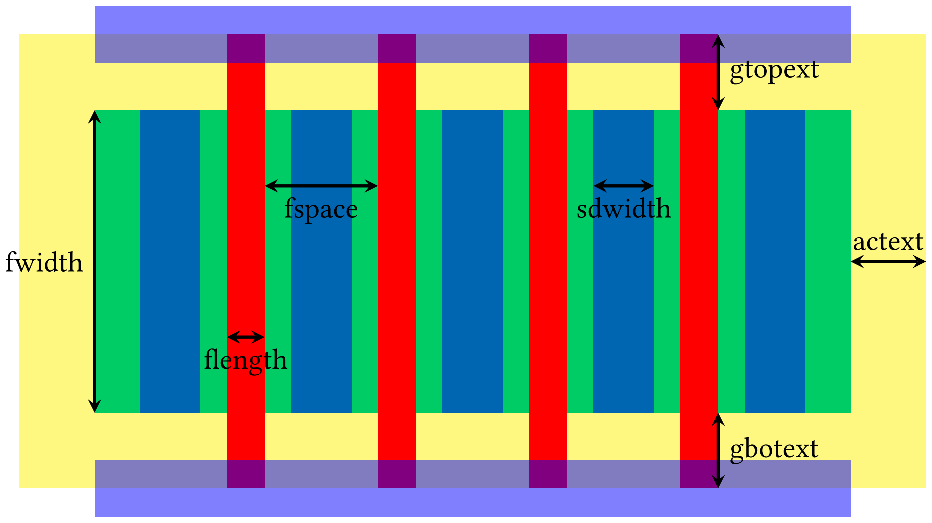

Figure 1: Overview of the transistor

and for gate contacts.

## 4 API Documentation

### 4.1 geometry Module

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

Create a rectangular shape with a width of `width` and a height of `height` in the layer-purpose-pair `layer` (usually a generic). The function returns an object.

`geometry.multiple(obj, xrep, yrep, xpitch, ypitch)`

Creates a rectangular array (mosaic) of an object with `xrep` repetitions in x and `yrep` repetitions in y. `xpitch` and `ypitch` are the center-to-center space in x and y direction. The entire array gets centered. The function returns the merged objects.

| 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

### 4.2 Object Module

### 4.3 Shape Module

### 4.4 Pointarray Module

### 4.5 Point Module