OpenPCells

Writing Custom Exports

Patrick Kurth

December 3, 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 custom export types. 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 are looking for a guide and documentation on the creation of parametrized cells, consult the celldesign manual. If you want to now more about the technical details and implementation notes, look into the technical documentation.

1 Overview

Exports in openPCells can be either written in C or in lua. C exports offer better performance, but have to be present at compile time. As the gds export is currently the most essential, it is written in C and included in openPCells. Other exports may be written in lua and can be defined and loaded by the user without recompiling the binaries. For instance, the SKILL export is written in lua. Exports work by defining functions that write specific shapes/objects such as rectangles or polygons. The functions that need to be defined follow closely the way layouts are represented in opc. Some functions (such as writing rectangles) are elementary and are mandatory, other ares optional (such as functions dealing with cell hierarchies). The calling environment of an export makes sure to reduce the layout to a representation that the export can understand (for example flattening layout hierarchies). In total, 16 different functions can be defined, but only 4 are mandatory. This is only really true for lua exports, different rules apply to C exports. As this guide focuses on lua, it will disregard special treatment of C exports.

In the following, all export functions (mandatory and optional) will be discussed in detail and some basic best practices regarding the writing of export types will be given. All viewings of export types will be focused on lua exports. C exports follow a similar fashion, but have more freedom in their processing of output data.

2 Export Functions

2.1 Mandatory Functions

2.1.1 finalize

This function is called at the end of the export: it should assemble the string/data that is then written to the respective file. Often, it simply uses `table.concat` to produce a string from a data table.

Example:

```
local __content = {} -- data table
function M.finalize()
    return table.concat(__content)
end
```

Required return value: string

2.1.2 get_extension

This function provides the file ending of the generated layout (e.g. returns ”gds” for the gds export). Most of the times, this will be just a simple return of a constant string. A call of opc with a given filename ’foo’ will then produce a file named ’foo.extension’, where ’extension’ is returned by this function.

Example:

```
function M.get_extension()
    return "gds"
end
```

Required return value: string

2.1.3 write_rectangle

This function defines how this export writes a rectangle. It receives the layer data and the bottom-left (bl) and top-right (tr) point as arguments. All arguments are tables, whereas the bl and tr table contain two fields each, x and y. The layer data table contains arbitrary data, which should be suitable for this export. Therefore, the expected data in this table must match the definition in the technology layermap file.

Modified Example from the SKILL export (the real function handles some edge cases and is more complicated):

```
table.insert(__content,
    string.format('dbWriteRect(cv, list("%s" "%s") %d:%d %d:%d)',
        layer.layer, layer.purpose,
        bl.x, bl.y,
        tr.x, tr.y
    )
))
```

The actual call to dbWriteRect is not important, but note how the layer data table and the points are used. The SKILL export expects a field layer and purpose in the layer data table. In SKILL-exported layouts, `cv` is defined outside of that file.

2.2 Optional Functions

2.2.1 initialize

This function can be used to perform required calculations at the start of the export. This function gets the maximum x- and y-coordinates passed as arguments in order to setup the canvas. It is called as first function, before `at_begin`.

Example taken from the SVG export:

```
local __width, __height
function M.initialize(minx, maxx, miny, maxy)
    local width = maxx - minx
    local height = maxy - miny
    __xoffset = -minx * __scale + __xmargin + __xoffsetmanual
    __yoffset = -miny * __scale + __ymargin + __yoffsetmanual
    __width = width * __scale + 2 * __xmargin
    __height = height * __scale + 2 * __ymargin
end
```

This sets up the `__width`, `__height`, `__xoffset` and `__yoffset` for the SVG canvas. These values are required at the beginning, so this function is called first.

2.2.2 get_techexport

This function can be used if the technology data uses a different name than the export. This way technology data can be shared between different export types. This is used for instance for the OASIS export, which is very similiar to the GDSII export. Both exports expect layer data as `{ layer = ..., purpose = ... }`. The OASIS export can then define this function to return gds, which will then select the gds data.

Example:

```
function M.get_techexport()
    return "gds"
end
```

2.2.3 set_options

Exports support options that can be given on an opc call with `-X`. This function receives a table with all collected options. The options are not parsed, only separated into tokens. It is up to the individual export to parse and process those options.

Example:

```
function M.set_options(opt)
    for i = 1, #opt do
        local arg = opt[i]
        if arg == "--foo" then
            -- set option foo
        end
        if arg == "--bar" then
            -- set option bar
        end
    end
end
```

2.2.4 at_begin

This function is called once at the beginning of the export process, after `initialize`. It is used to write some basic definitions (for instance, a GDSII stream starts with the library name, the unit definition and similar data).

Example:

```
function M.at_begin()
    print("Export start")
end
```

2.2.5 at_end

This function is called once at the end of the export process. It is used to finish the output (for instance, a GDSII stream ends with an ENDLIB entry).

Example:

```
function M.at_end()
    print("Export end")
end
```

2.2.6 write_triangle

This function defines how a triangle is written. Similar to `write_rectangle`, it receives the layer data table and three points as arguments. This function is only useful, when `write_polygon` is not defined, as triangles are just a special form of a polygon. If `write_polygon` is not defined but `write_triangle` is, all polygons are triangulated. This is for instance how polygons are represented in the export for the layout editor magic.

Example:

```
function M.write_triangle(layer, pt1, pt2, pt3)
    -- layer data is in layer
    -- pt1, pt2 and pt3 are the corner points of the triangle
end
```

2.2.7 write_polygon

This function defines how a polygon is written. It is similar to `write_rectangle`, it receives the layer data table and a table of points as arguments. If this function is not defined, then polygons are triangulated and `write_triangle` is required, so in fact one of these two functions is mandatory. If none of these functions are defined, polygons are not supported (which can get you suprisingly far in integrated electronics).

Simplified example from the tikz export:

```
function M.write_polygon(layer, pts)
    local ptstream = {}
    for _, pt in ipairs(pts) do
        table.insert(ptstream, string.format("(%s, %s)", pt.x, pt.y))
    end
    table.insert(__content, string.format("\\path[draw, color = %s] %
       s -- cycle;", layer.color, table.concat(ptstream, " -- ")))
end
```

2.2.8 write_path

This function defines how a path is written. Semantics-wise, it is very similar to `write_polygon`, but it receives a path width as third argument and a path extension type as fourth argument. If this function is not defined, paths are converted to polygons (which of course then requires `write_polygon`).

Simplified example from the SKILL export:

```
function M.write_path(layer, pts, width, extension)
    local ptrstr = {}
    for _, pt in ipairs(pts) do
        table.insert(ptrstr, string.format("%s:%s", pt.x, pt.y))
    end
```

```
    local c = {}
    local extstr = ''
    if extension == "butt" then
        extstr = '"squareFlush"'
    elseif extension == "round" then
        extstr = '"roundRound"'
    elseif extension == "cap" then
        extstr = '"extendExtend"'
    end
    local fmt = 'dbCreatePath(cv list("%s" "%s") list(%s) %f %s)'
    table.insert(__content, string.format(fmt,
            layer.layer, layer.purpose,
            table.concat(ptrstr, " "),
            width, extstr
        )
    )
end
```

2.2.9 write_cell_port

This function exports layout ports (sometimes called labels). Ports don't offer physical functionality but are required for connectivity check for layout vs. schematic (LVS). This function receives a port name, a layer data table and a location (a point) as arguments. A fourth optional argument is a hint for the size, which can make generated layouts more human-readable.

Simplified example from the SKILL export:

```
function M.write_port(name, layer, where, sizehint)
    sizehint = sizehint or 0.1
    local fmt = 'dbCreateLabel(cv list("%s" "%s") %s:%s "%s" %f)'
    table.insert(__content, string.format(fmt,
        layer.layer, layer.purpose,
        pt.x, pt.y,
        name, sizehint)
    )
end
```

2.2.10 write_cell_reference

With this function, layout hierarchies are possible. This function writes a reference to a cell (a child in opc terminology). For this, two things are required: The definition of the cell (see `at_begin_cell`) and the reference to that cell (this function). This function receives four arguments: The identifier of the cell (a string), the x- and y-coordinate and a table (a matrix) representing the orientation of the cell. The orientation matrix contains entries for rotation and

mirroring. It is like a transformation matrix where

$$\bar{x} = M_1 x + M_2 y$$
$$\bar{y} = M_3 x + M_4 y$$

M_i corresponds to `orientation[i]`.

If this function is not defined, layouts are flattened (cell hierarchies are resolved) before they are exported.

Simplified example from the SKILL export (ignoring the orienation):

```
function M.write_cell_reference(identifier, x, y, orientation)
    local fmt = 'dbCreateInstByMasterName(cv libname "%s" "layout"
        nil %s:%s "R0")'
    table.insert(__content, string.format(fmt,
        identifier,
        x, y,
    )
end
```

In this example, `libname` is defined outside of the exported layout file (due to how the SKILL export is used).

2.2.11 write_cell_array

This function is very similar to `write_cell_reference`, but works with *arrays* of cell references. Therefore, the first four arguments are the same as in `write_cell_reference`. The last four arguments represent the repetition in x and y and the pitch in x and y.

If this function is not defined, cell arrays are manually written out with `write_cell_reference` .

Simplified example from the SKILL export (ignoring the orienation):

```
function M.write_cell_array(identifier, x, y, orientation, xrep, yrep
    , xpitch, ypitch)
    local fmt = 'dbCreateParamSimpleMosaicByMasterName(cv libname "%s
        " "layout" nil %s:%s "R0" %d %d %f %f nil)'
    table.insert(__content, string.format(fmt,
        identifier,
        x, y,
        xrep, yrep, xpitch, ypitch
    )
end
```

2.2.12 at_begin_cell and at_end_cell

This function is the counterpart for `write_cell_reference`. All cell references have to be defined, which is done with the regular functions like `write_rectangle` and `write_polygon`. In order to group these shapes into cells, the cell start and end are represented by `at_begin_cell` and `at_end_cell`. The former function receives the name of the cell as argument.

If `write_cell_reference` is not defined, layouts are flattened before they are exported. However, some exports (such as for GDSII streams) still require these functions for flat layouts.

Example:

```
function M.at_begin_cell(cellname)
    -- GDSII stream: write BGNSTR record with cellname
end
function M.at_end_cell()
    -- GDSII stream: write ENDSTR record
end
```