3/28/2018

PEARDROP
DESIGN
SYSTEMS

GLADE REFERENCE MANUAL

Version 4.6.37

1 Table of Contents

1.1 Introduction

Glade is a versatile tool for IC design, enabling schematic capture, netlisting, layout generation and verification. Unlike many commercial tools, Glade is cross-platform, running on Windows (32 and 64 bit), Linux (32 and 64 bit) and Mac OSX (64 bit), with a database that is platform-independent.

Glade reads and writes common data formats, such as GDS2, OASIS, DXF, LEF/DEF, SPICE/CDL, and Verilog.

Glade is programmable in Python, and features such as PCells and DRC/LVS use Python scripting for ease of use.

1.2 Getting Started

1.2.1 Command line options

From the command line, Glade can be invoked with a number of command line options:

glade [-open library] [-libName libname [-tech techFile] [[-map gdsmapfile] -drf displayfile -tf cdstechFile] [-edif edifFile] [-gds gdsfile] | [-oasis oasisfile] | [-oasisout oasisfile] | [-dxf dxffile] | [-lef leffile -def deffile]] [-cell cellname] [-script pythonfile] [-h] [-v]

-**ng**: run in non-graphics mode. Note that this is only meaningful with the -script option, and any python script must NOT call any gui functions e.g getEditCellView(), else an exception will occur.

-open <name> : the directory name to load as a Glade library. It must have been created by a previous Save Lib command. The library is made current and subsequent import options e.g. –tech, -gds etc. will import data into this library.

-libName <name> : the library name to import GDS2 or LEF/DEF into. If not specified, the library name will be 'default'. The library is made current and subsequent import options e.g. –tech, -gds etc. will import data into this library.

-tech <filename> : An optional Glade technology file to read. Technology files define layer colours, line and fill patterns and are described in section XXX.

-map <filename> : An optional GDS2 layer map file, used only when the -drf option is specified. It must be specified before -drf.

-drf <filename> : An optional Cadence display resource file. If specified, -tf must also be specified subsequently. It should not be used with the –tech option.

-tf <filename> : An optional Cadence technology file (ascii Skill format). If specified, -drf must also be specified first.

-edif <filename> : the name of an EDIF file to import. The EDIF file defines the libraries, cells and views that will be imported.

-gds <filename> : the name of a GDS2 file to import into the current library. Multiple GDS2 files can be specified.

-gdsout <filename> : the name of a GDS2 file to export from the current library. The program will exit after the GDS2 file is written.

-oasis <filename> : the name of an OASIS file to import into the current library. Multiple OASIS files can be specified.

-**oasisout <filename>** : the name of an OASIS file to export from the current library. The program will exit after the oasis file is written.

-lef <filename> : the name of a LEF file to import into the current library. Multiple LEF files can be imported, however duplicate definitions of SITEs and MACROs should be avoided as duplicate definitions will be ignored.

-def <filename> : the name of a DEF file to import into the current library. A cell will be created according to the DEF DESIGN statement. Multiple DEF files can be specified.

-dxf <filename> : the name of a DXF file to import into the current library. The top level cell will be called 'top'.

-cdl <filename> : the name of a CDL file to import into the current library. Cells will be created with a view type of 'netlist' for each subcircuit in the CDL/Spice file.

-cell <name> : the name of a cell to open and display. Note that the viewType is assumed to be "layout".

-cellview <cellname> <viewname> : the name of a cell and a view to open and display.

-script <filename> : the name of a Python script file to run. The script will be run after all other commands.

-h: prints usage info

-v: prints the current version

1.2.2 Environment Variables

Glade can make use of several environment variables. These are the documented ones:

- GLADE_HOME – used by the help browser to locate the html help files.
- GLADE_NO_EXCEPTION_HANDLER – Do not use the exception handler for Qt events. Can sometimes help with debugging info if errors occur.
- GLADE_DEBUG_SUBMASTERS – Display PCell submasters in the library browser. Normally these are hidden, as they are not for the user to manipulate.
- GLADE_LOGFILE_DIR – The directory to write the logfile to. If not specified, Glade will write the logfile to the current working directory, or the home directory.
- GLADE_USE_OPENGL – If set to no, Glade will not use OpenGL for drawing layout views, even if the user's system has OpenGL capabilities. Useful if only software OpenGL implementations are present.
- GLADE_DRC_FILE – the full path to a DRC file used to seed the Run DRC dialog.
- GLADE_EXT_FILE - the full path to an extraction file used to seed the Run LPE dialog.
- GLADE_NETLIST_FILE – the full path to a CDL netlist file for the Run LVS dialog.
- GLADE_DRC_VARS – a string list of DRC variables to preset the Run DRC dialog.
- GLADE_EXT_VARS – a string list of extraction variables to preset the Run LPE dialog.
- GLADE_DRC_WORK_DIR – a working directory for writing geom… temporary files.
- GLADE_THREADED_EXTRACTION – set to number of threads allowed for running extraction. Default is the maximum number allowed by the CPU(s).
- GLADE_FASTCAP_WORK_DIR – a working directory for writing FastCap mesh files.

- GLADE_NO_DELETE_TMPFILES – if set, do not delete temporary FastCap mesh files. Useful if you want to view the mesh geometry.
- PYTHONPATH – Glade's Python interpreter uses this to locate Python modules, e.g. PCell files.

1.2.3 Style Sheet

Glade will read a stylesheet file named glade.qss if found in the same directory as the executable. This can be used to e.g. set font size for the whole application, or for specific widgets. A guide to the file format is given in the Qt documentation.

1.2.4 Settings file

Glade reads a gladerc.xml settings file whenever a design is opened. The settings file contains display and selection settings, window arrangement and bindkey settings. Glade will attempt to read the gladerc.xml from the user's home directory, as defined by the HOME environment variable. It will also open a local gladerc.xml file in the current working directory, if it exists, and merge this with the settings from the HOME gladerc.xml. This allows project-specific settings to be applied. For backwards compatibility, if a gladerc.xml file is not found, a .gladerc file (old non-xml format) will be read. On exit, Glade will write to the local gladerc.xml file if it exists, else it will write to the gladerc.xml file in the users HOME directory.

1.2.5 Startup script file

Glade can read a Python startup file if it exists, and execute the Python commands in it **before** processing command line arguments. The startup file must be called *.glade.py* (note the preceding dot). Glade will load the startup file from the user's HOME directory, if the file exists. It will also load a startup file from the current working directory if the file exists there. The order of loading is the home directory first followed by the local directory. A startup file is useful for loading e.g. a technology file or loading libraries. An example:

```
mylibs = ["CNM25TechLib", "SPICE3Lib", "XSpiceLib", "ExampleLib"]
nlib = len(mylibs)
libinit = [0 for i in range(nlib)]
for n in range(nlib):
    libinit[n] = library(mylibs[n])
    libinit[n].dbOpenLib("./"+mylibs[n])
```

1.2.6 The main window

The Glade main window (Figure 1.1) comprises the following components:

- Menu Bar
- Toolbars
- Tab or Multiple Window (MDI) area
- Dock Windows
    - Message Window
    - Library Browser
    - LSW (Layer Select Window)
    - World View
    - Net Browser
    - Hierarchy Browser
- Command Line

- Status Bar

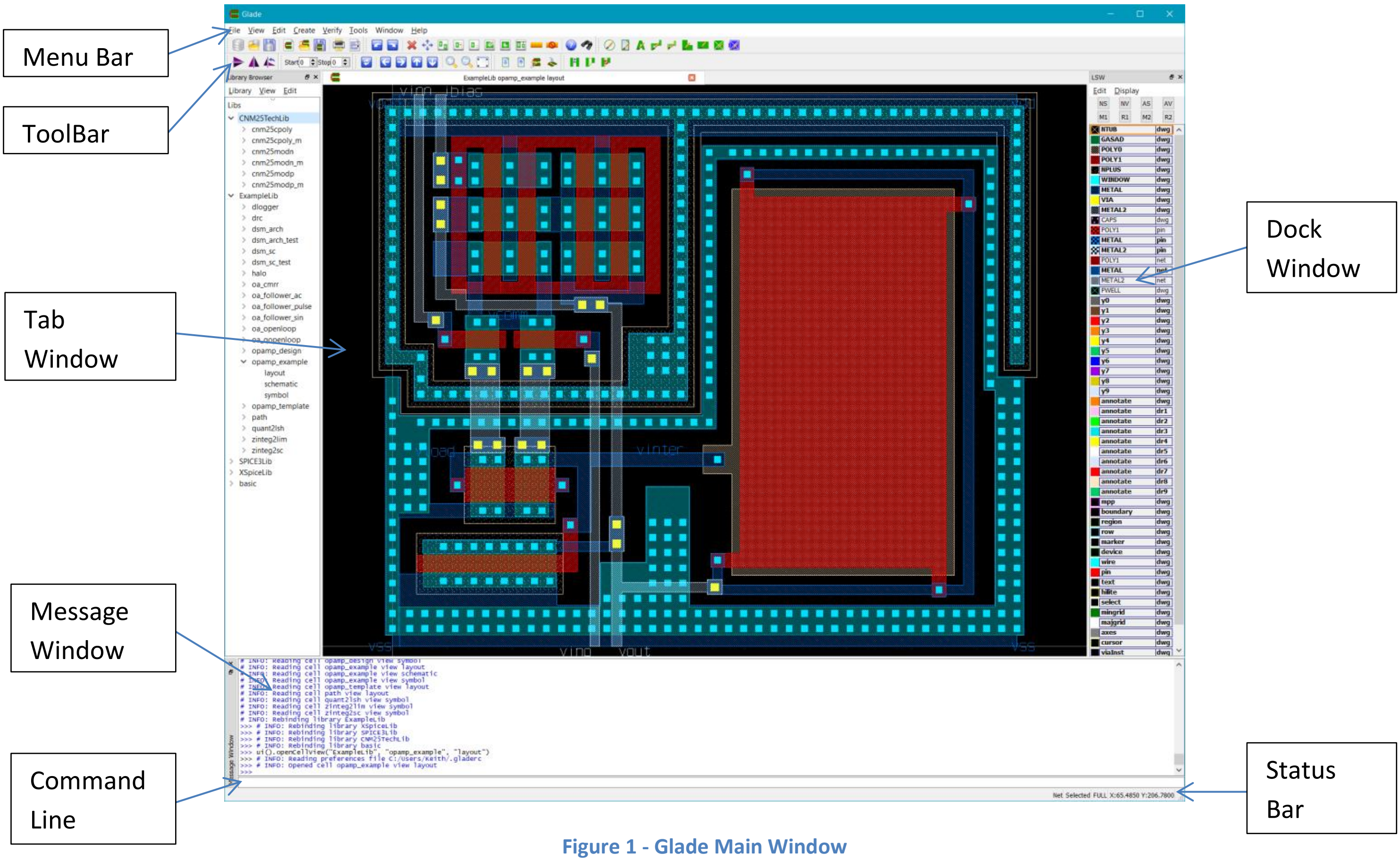


Figure 1 - Glade Main Window

The menu bar shows the current menu items. The default menu bar items are the File, Tools, Window and Help menus. When a cellView is opened, the menu items and toolbars will change according to the viewType of the cellView.

CellViews are displayed in the central area of the main window. They can be displayed either in tab windows or multiple subwindows. Tab windows allow easy navigation between designs by clicking on the tab; multiple windows allow different designs to be shown and the windows tiled or cascaded; for example a schematic view and a layout view of the same cell can be displayed side by side when in MDI window mode. The Window menu allows for switching between tab and MDI mode, and for switching between subwindows or tab windows.

Dock Windows are used for the library browser, LSW and other browsers. They can be dragged and positioned at the sides of the central window, including stacking them to save space. The message window is normally displayed at the bottom of the central window, with other dock windows on the left and/or right of the central window.

The Command Line is displayed just below the Message Window and is used to enter textual commands in Python. The built in Python interpreter in Glade displays the results in the message window. The Command Line allows normal editing e.g. ctrl+A to move to the beginning of a line, ctrl+E to move to the end, and the up/down arrow keys to recall the last/next command in the command history.

Lastly the Status Bar displays info such as details of menu items or toolbars the cursor is hovering over, plus information about the selected object/net, number of items selected, selection mode, cursor XY coordinates and delta XY coordinates for e.g. move operations.

1.2.7 The LSW, layers and purposes

Glade draws shapes on layers. Internally, layers are represented as an index into a technology file layer table. From a user's point of view, each layer is defined by a layer name and a purpose name. This allows subdividing layer name space depending on use; for example a layer called "METAL1" may have purpose names "drawing", "net", "pin", "boundary" etc. The combination of a layer name and a purpose name is called a Layer Purpose Pair (lpp).

Layers can be either user-defined or system layers. System layers are used for specific functions, for example the cursor is drawn on the "cursor" "drawing" lpp.

The LSW (Layer Selection Window) is used to control layer display in Glade. It comprises a dockable dialog box with a scrollable panel of layers - one for each layer defined in the technology file - plus some system defined layers. Each layer in the LSW has 3 parts: a **colour box** on the left which displays the layer line and fill style; a **layer box** in the centre which displays the layer's name, and a **purpose box** on the right which displays the layer's purpose, abbreviated to 3 characters (for example 'drawing' becomes 'dwg', 'pin' becomes 'pin', 'boundary' becomes 'bdy' and 'net' is represented as 'net').

The LSW shows user-defined layers and the system layers. System layers include the following:

- Layers y0-y9, used for temporary display purposes
- Layers annotate (purpose drawing, drawing1-9), used for schematic/symbol labels
- mpp - Used internally for MPP objects. Do not draw on this layer
- boundary - used for cell boundaries for LEF cells and the DEF design boundary
- region - used to display DEF regions
- row - used to display rows from DEF
- marker - used for flagging DRC errors
- device - used for symbol shapes
- wire - used for schematic wires
- pin - used for schematic and symbol pins
- text - used for autogenerated text labels e.g. as a result of importing LEF
- hilite - used for displaying flightlines e.g. for connectivity
- select - used to highlight selected objects
- mingrid - used to draw the minor grid
- majgrid - used to draw the major grid
- axes - used to draw the axes
- cursor - used for the box or crosshair cursor
- viaInst - for via instances that are shown unexpanded
- instance - for instances that are show unexpanded
- backgnd - the background display colour (defaults to black, but can be set to any colour)

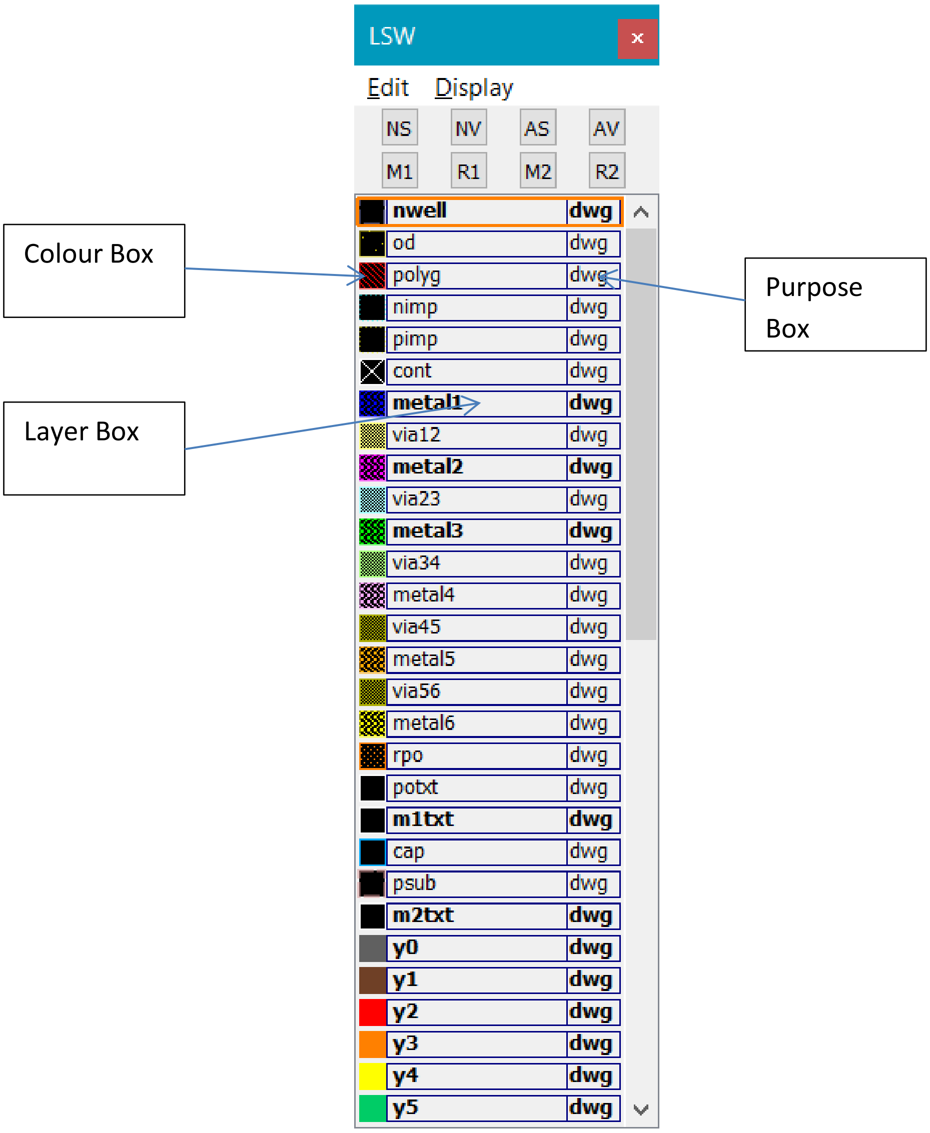


Figure 2 - The LSW

At the top of the LSW are four buttons NS (None selectable), NV (None visible), AS (All Selectable), AV (All visible) which allow all layer selectability/visibility to be set at once. Below this are 4 buttons M1 (save to memory 1), R1 (recall from memory 1), M2 (save to memory 2) and R2 (recall from memory 2). These allow the current layer selectability / visibility to be saved and recalled for frequent changes. As changes are made that affect the display (changing colour, fill pattern or layer visibility) the display is automatically updated.

1.2.8 Selection

Most Glade commands work on the 'selected set'. The left mouse button (LMB) is used for selection.

- Single click selects objects
- Shift+Click adds objects to the selected set
- Ctrl+Click removes objects from the selected set
- LMB drag selects objects within the drag area
- Shift+LMB drag adds objects within the drag area to the selected set
- Ctrl+LMB drag removes objects within the drag area from the selected set

The number of selected items is shown in the status bar. Selected objects are displayed highlited using the select layer. Unselected objects can be drawn dimmed by using the Selection Options *Dim unselected objects* option.

Selection works in two modes: Full and Partial. In Full mode, whole objects are selected. In Partial mode, edges or vertices of shapes are selected. The selection mode is set using the Selection Options dialog, or using the F4 key to toggle between modes. Shape selection can be controlled using the LSW.

Glade has two selection types: Item and Net. You can set the selection type in the Selection Options dialog. Item mode selects individual shapes, instances etc. Net mode will select all shapes of a net if any shape selected is part of a net.

1.2.9 Libraries, Cells, Views and CellViews

Glade manages design data in ***libraries***. You can create as many libraries as you need. For example, if you have a number of GDS2 files, and want to use the design data in each in an overall top level design, you could import each GDS2 file into a unique library, and then create a library to hold your top level cell which references cells from each of these libraries. A library is a collection of ***cells***, where a cell is for example an inverter, a nand gate, a block or a complete top level design. Cells correspond to GDS2 STRUCT objects, or a DEF DESIGN, for example. A cell can have different ***views***, a view being a representation of that cell. For example a view type of 'layout' is used to represent raw physical layout data e.g. the result of importing GDS2. A view type of 'abstract' is used for simplified layout data from importing LEF. A view type of 'extracted' is used when extracting layout connectivity and devices. The combination of a specific cell and a view for that cell is called a ***cellView***.

Before you can import design data, you need to create a library to hold that design data. You can use the **New Library** command for this, and then attach a technology file to the library using the **Import TechFile** command, or more simply just use the **Import TechFile** command, which allows you to enter a library name; the library will be created and the technology file attached to the library. As some people want to just read in a GDS file or LEF/DEF without bothering to create a technology file. The File->Import commands will generally allow you to create a library with a default technology file.

1.2.10 PCells

Glade can use python to create parameterised cells, or PCells. A parameterised cell has a python script that defines how the cell is created, and takes parameters. For example a MOS device might take a W and L value and have the transistor automatically created with the correct poly, diffusion and contact layers. Please note that Glade PCells are NOT compatible with Cadence Skill-based PCells, or Synopsys PyCells. PCells are described in more detail in the section “PCells”.

1.2.11 Python

The entire Glade database and much of the UI is wrapped in Python using SWIG. This means you can write Python scripts to automate tasks - PCells (parameterised cells) are a good example. Python is an object-oriented language widely used for scripting. The Python API is described in more detail in the section “Programming in Python”.

1.2.12 Error reporting

In the unfortunate event of an internal program error occurring, Glade will trap the error and report diagnostics which can be mailed to the developers so the bug can be fixed. To get diagnostics reported, you must set the environment variable GLADE_NO_EXCEPTION_HANDLER to yes. Otherwise the exception may be handled by the default GUI exception handler.

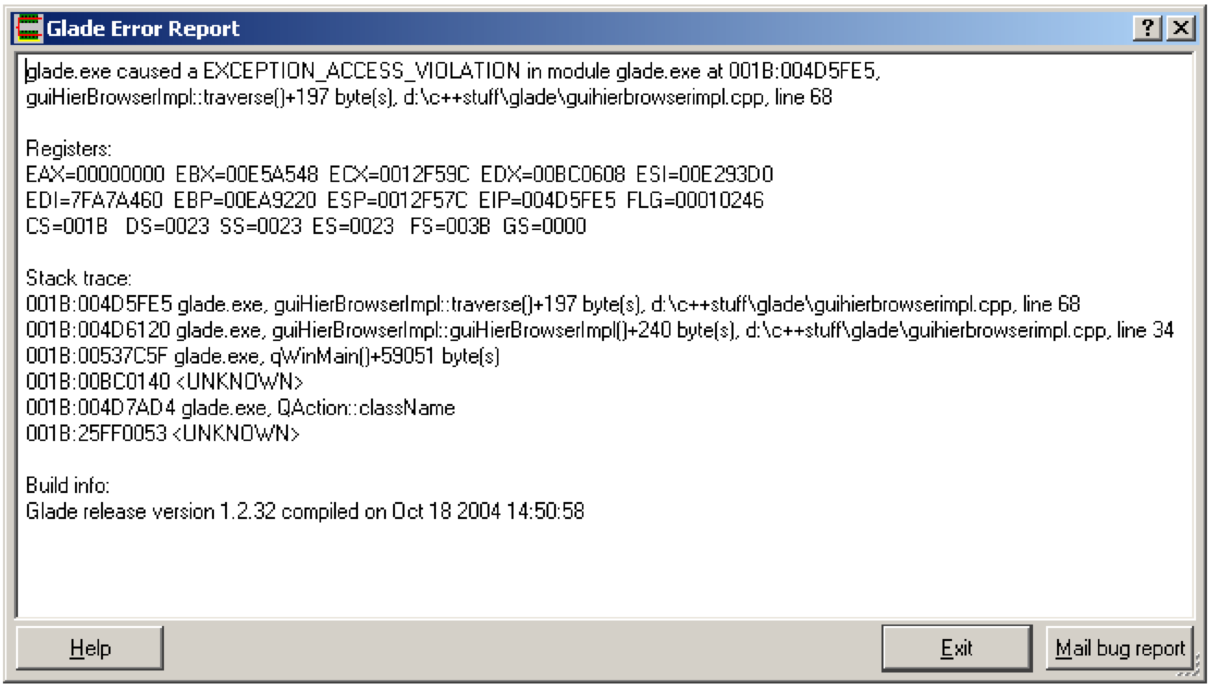


Figure 3 - Glade error report

The error report shows the type of error, CPU register contents and a stack trace with the most recent stack frame first. Clicking on the Mail bug report button will mail the stacktrace to the developers. Clicking on the Exit button will exit the application. If there is unsaved data, you will be prompted to save the library.

2 Menus

2.1 The File Menu

File Menu commands are used for creating, opening and saving libraries and cellViews. They are also used for importing and exporting design data and other general functions.

Normally the sequence of importing design data into Glade is performed by importing a techFile first, then the design data e.g. GDS2. If you do not have a technology file, you can just import GDS2 or LEF/DEF, as basic technology information will be created for each layer read. In the case of GDS2, layers will be of the form L0, L1... where the number is the GDS2 layer number. All layers created by importing GDS2 will have purpose drawing, and layer colours will be assigned at random with hollow fill style. Layers created by importing LEF will have the LEF layer name and 4 purposes (drawing, net, pin and boundary). You can then subsequently export the technology file for later use.

2.1.1 File->New Lib

The **File->New Lib** command creates a new library, with name as specified by *Library Name*.

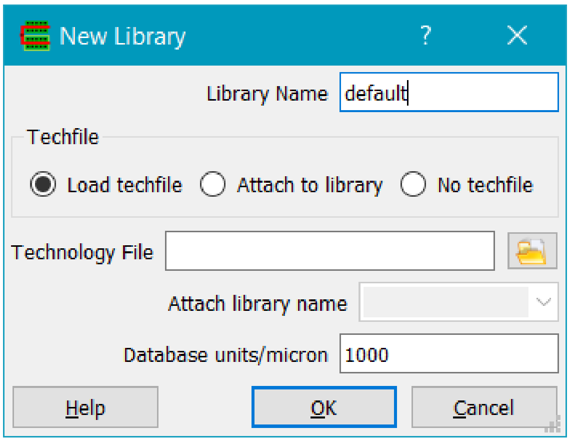


Figure 4 - New Library

Load techFile enables the *Technology File* field and will load that techFile into the new library. *Attach to library* enables the *Attach library name* field, and will attach the library's techFile to an existing (open) library. *Database units/micron* controls the precision of the represented data. Unless you have a good reason to change this and understand the implications, leave it as 1000 (i.e. 1 dbu = 1nm).

2.1.2 File->Open Lib

The **File->Open Lib** command opens an existing library. The Open File dialog is displayed (the actual look will depend on what OS you are using)

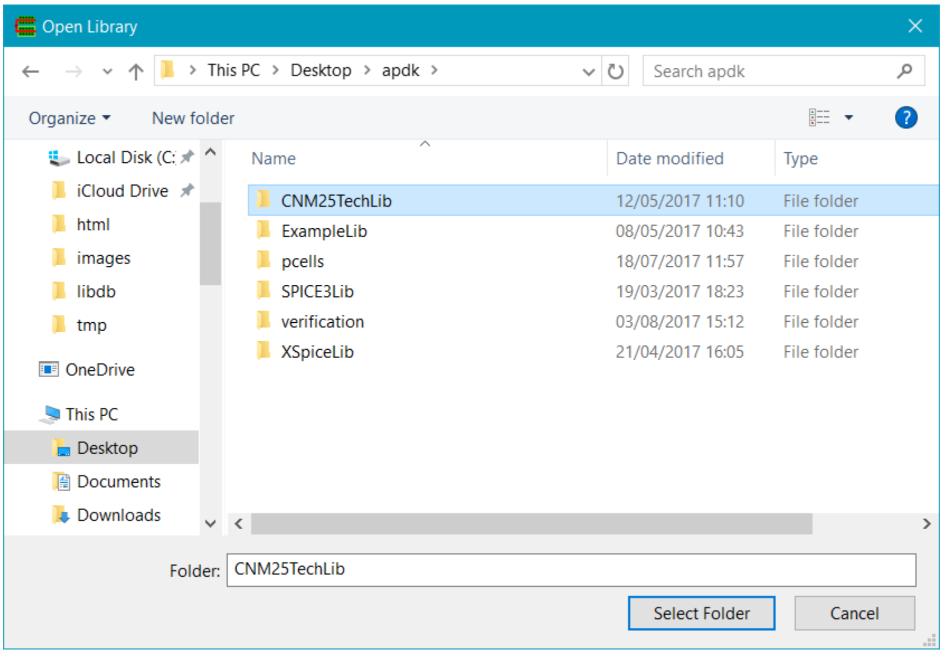


Figure 5 - Open Library

If you have an existing Glade library, you can use the Open Library command to specify a library name to open. Note that Glade libraries are just directories, so select the library by selecting the directory with the same name and click Select Folder (Windows). Internally cellViews are stored as files in a library of the form libName/cellName/viewName. The library technology file is also stored in the library in binary format and is stored as libName/glade.lib.

2.1.3 File->Save Lib

2.1.4 File->Save Lib As...

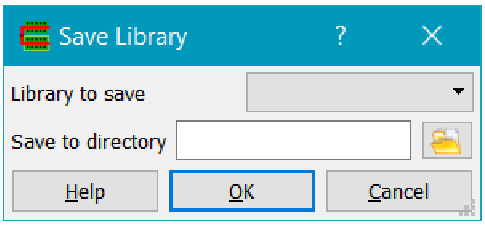


Figure 6 - Save Lib As...

Use the **File->Save Lib** or **File->Save Lib As...** to save a library to disk after importing design data. *Library to save* chooses the library you wish to save. *Save to directory* specifies a directory name in Linux or folder name in Windows. Click on the file chooser icon to browse to a directory. The library data is written to files in this directory/folder. These files are binary - do not attempt to alter them, delete or rename them, or your design data may become corrupted.

2.1.5 File->Close Lib

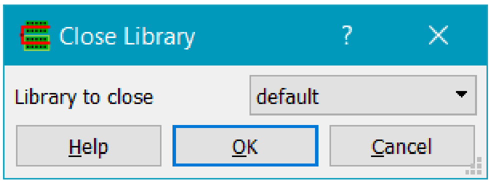


Figure 7 - Close Library

The **File->Close Lib** command closes the chosen library. All cellViews from the library will be purged from virtual memory. The system will prompt you to save any modified cellViews. If a window displaying a cellView from the library is open, it will be closed.

2.1.6 File->New Cell

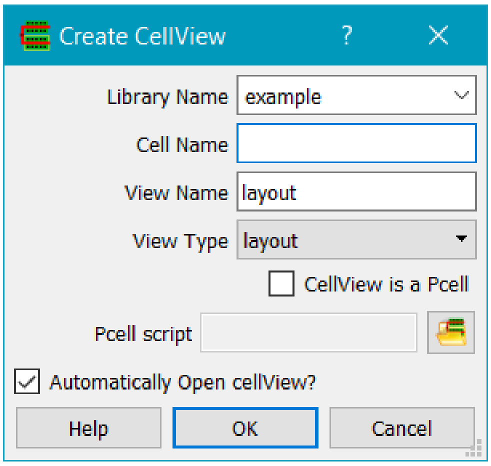


Figure 8 - New CellView

The **File->New Cell** command to creates a new cellView. The library given by *Library Name* must already exist. Specify *the Cell Name* and the *View Name*. Set the *View Type* to the type of the cellView; valid options are layout, schematic, symbol, abstract, autoLayout. Setting the *View Type* will set a default *View Name*. If *CellView is a PCell* is checked, a PCell (parameterised cell) will be read from the *PCell script* file. In this case the Cell Name is automatically assigned from the python script name, and the *Cell Name* field is greyed out.

The new cellView is added to the library and displayed in the library browser, and automatically opened if *Automatically Open cellView?* Is checked.

2.1.7 File->Open Cell...

The **File->Open Cell...** command displays the library browser, if not already shown, to allow opening of a cellView. CellViews are opened in the library browser.

2.1.8 File->Save Cell

The **File-> Save Cell** command saves the current cellView to the library on disk.

2.1.9 File->Save Cell As…

The **File-> Save Cell As…** command prompts for a new cell name, then saves the cellView to the library on disk.

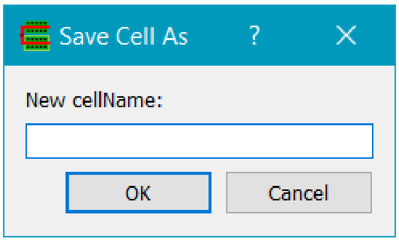


Figure 9 - Save Cell As

2.1.10 File->Restore Cell

The **File->Restore Cell** command restores a cellView from disk. Any current edits will be lost.

2.1.11 File->Import->Cadence TechFile

The **File->Import->Cadence TechFile...** command displays the Import Cadence TechFile dialog.

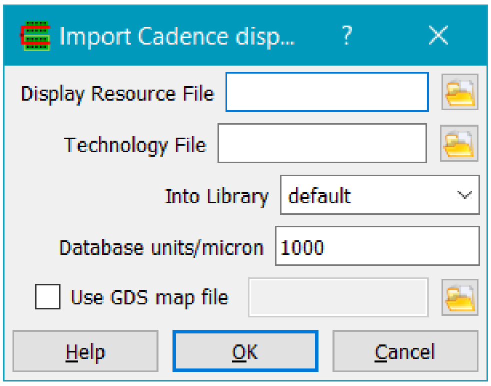


Figure 10 - Import Cadence TechFile

Display Resource File specifies the Cadence display resource file (typically display.drf). *Technology File* specifies the Cadence technology file. *Into Library* should specify a library name to import the technology into, and will be created if it does not already exist. If the library does exist, the imported techFile will be merged with the existing one. This can produce unpredictable results and is not advised. *Database units/micron* sets the internal database resolution; in most cases 1000 is suitable. If *Use GDS map file* is set, stream layer/datatype numbers to Cadence layer/purpose names are set using the specified mapfile. The map file format is simply lines containing layer name, purpose name, stream layer number and stream datatype number. Comment lines (lines beginning with the # character) are ignored.

```
#Layer Name     Purpose Name     Stream#     datatype
od              drawing          6           0
```

Note there are some limitations on importing Cadence techFiles. Stipple patterns of size 4x4, 8x8, 16x16 and 32x32 are supported, other stipple pattern sizes will be rounded up to the next supported size. The Cadence techFile should be written from Virtuoso and should not be hand edited else it may not parse successfully – Skill expressions are not parsed.

2.1.12 File->Import->Laker TechFile

The **File->Import->Laker TechFile...** command displays the Import Laker TechFile dialog.

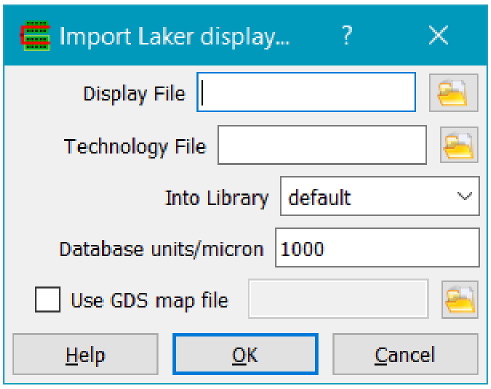


Figure 11 - Import Laker TechFile

Display File specifies the Laker display file (typically default.dsp). *Technology File* specifies the Laker technology file. *Into Library* should specify a library name to import the technology into, and will be created if it does not already exist. If the library does exist, the imported techFile will be merged with the existing one. This can produce unpredictable results and is not advised. *Database units/micron* sets the database units. *Use GDS map file*, if checked, allows a GDS layermap file to be used. The map file format is simply lines containing layer name, purpose name, stream layer number and stream datatype number. Comment lines (lines begininning with the # character) are ignored. If the technology file also contains a tfStreamIoTable section, the map file entries will be merged and will overwrite tfStreamIoTable entries.

Laker stipple patterns of size 4x4, 8x8, 16x16 and 32x32 are supported, other stipple pattern sizes will be rounded up to the next supported size. Currently only layer colour / stipple / linestyle data and stream number / datatype info is read from the Laker techFile.

2.1.13 File->Import->TechFile

The **File->Import->TechFile** command displays the Import TechFile dialog.

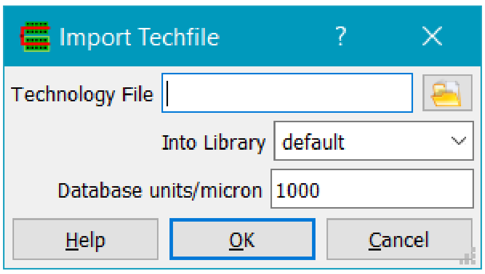


Figure 12 - Import Glade TechFile

A Glade technology file can be used when no Cadence / Laker techFile is available. The *Technology File* can be chosen using the file chooser button. *Into Library* specifies the library name, and the library will be created if it does not already exist. If the library does exist, the imported techFile will be merged with the existing one. *Database units/micron* sets the internal database resolution; in most cases 1000 is suitable.

2.1.14 File->Import->GDS2

The **File->Import->GDS2** command displays the Import GDS2 dialog:

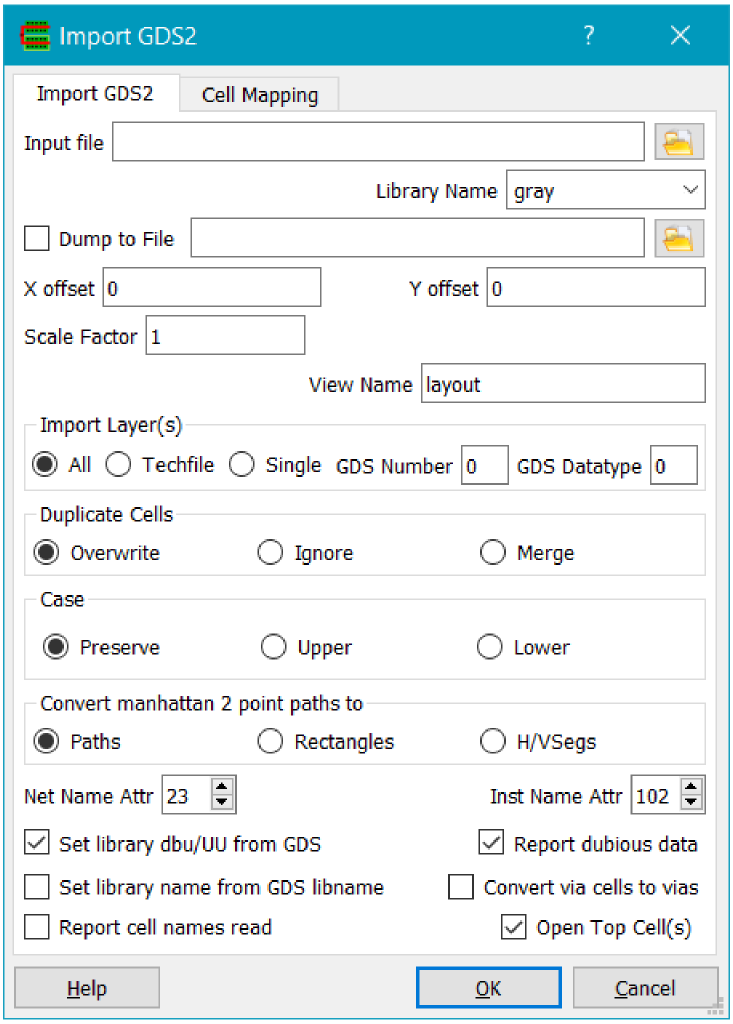


Figure 13 - Import GDS2

The *Input File* can be chosen using the file chooser button. Multiple GDS2 files can be read if they are entered separated by a comma. If the file name extension is '.gz' then the compressed file will be unzipped on the fly.

The *Library Name* field specifies the library name that the GDS2 will be imported to. If you have previously read in a techFile, the library field will be preset to this library name. If the library does not exist, it will be created with a default techFile.

For debugging purposes, the GDS2 can be dumped to a readable ASCII format if the *Dump to File* button is checked and a file name given.

An *X offset* and *Y offset* can be specified. The specified offsets are added to all coordinates in the design, in effect moving the origin of the design. Note that offsets are applied BEFORE any user-specified *Scale Factor*.

GDS2 can be scaled while read in if the *Scale Factor* field is set to a number other than 1. For example, if a scale factor of 0.5 is chosen, all coordinates will be multiplied by 0.5 and the design is shrunk by a factor of 2. This can be useful for scaling entire design databases.

The *View Name* specifies the view name created when a cell is imported. If cell mapping is used, this value will be overridden by the map library/cell/view names.

You have a choice of importing all layers, layers defined in the techFile or just a single layer in the *Import Layer(s)* field. When *Single* is selected, a *GDS2 Number* and *GDS Datatype* need to be specified for the layer, and only shapes on this layer/datatype will be imported.

When cells are imported, if a cell of the same name exists you have 3 options available in the *Duplicate Cells* field. *Overwrite* means the new cell will replace the existing cell. *Ignore* will mean the new cell definition is ignored, along with all data in it. *Merge* means the original cell data is preserved, and any data in the new cell is added to it. This may cause duplicate shapes, but can be used to merge GDS data.

GDS2 structure and array names can have their *Case* preserved, forced to upper case or lower case depending on the 'Case' radio buttons. Note that if you have a structure named 'AND2' and one called 'and2' and do not preserve case, then the second structure encountered will give rise to a duplicate cell and will be handled by the settings in the Duplicate Cells field..

Convert Manhattan 2 point paths converts paths to rectangles or H/V segs. This can result is smaller memory usage for designs that use lots of 2 point paths for e.g. metal fill.

GDS2 properties can be used to import net names and instance names into the Glade database. Many layout editors and place & route tools can output this data, and if GDS2 properties are present with the chosen attribute numbers then net and/or instance names will be annotated into the database. The *Net Name Attr* is the number of the attribute used to read net names from. *The Inst Name Attr* is the number of the attribute used to read instance names from.

Set Library dbu/UU from GDS will set the library database units from that specified in the GDS2 file. This should normally be checked if importing into an empty library. If you want to import GDS2 data into an existing library, uncheck this so the existing library units can be used; the GDS2 data will be scaled to match if the GDS2 units differ from the library units. Note this scaling occurs before any user-defined offset or user-defined scale factor is applied.

Convert via cells to vias will identify potential via cells in the GDS. A via cell is a cell with 3 layers, of which two are of function ROUTING and one of function CUT, as defined in the techFile. A via will be created for each distinct cell and added to the library. On stream out via Export Gds2, vias can be converted back to cells.

Report Dubious Data will give warnings/errors to the message window if dubious data is encountered, such as polygons with less than 3 vertices.

Report cell names read will write each cell (GDS STRUCT) encountered in the input GDS data. For large designs this can slow things down so by default it is turned off.

Open Top Cell(s) will attempt to identify and open cells that appear to be the 'top cell' of a GDS file. A top cell is not referenced by other cells, and contains one of more cell placements.

The GDS2 reader is single pass. As forward references are allowed in GDS2 (a cell, or GDS structure, can be referenced in a SREF before the cell has been defined), after reading the GDS a recursive check is made to ensure all cells have valid bounding boxes.

GDS2 magnification is supported in Glade. GDS SREFs or AREFs (instances or arrays in Glade) can only have Manhattan rotations. This is to maintain compatibility with Cadence Virtuoso, which has the same limitation.

GDS2 arrays are not allowed to have non-orthogonal row/column spacing. A warning is issued if encountered, and they will be represented as orthogonal arrays. This is consistent with Cadence Virtuoso and the GDS2 'specification'.

If a GDS file is imported without a Glade techFile having been previously read which defines the mapping between layer names/purposes and GDS layer numbers / datatypes, then the GDS layers are mapped to layer names e.g. L0 P0 for the first GDS layer/datatype shape encountered etc. The layer name assigned (L0) does NOT equate to the GDS layer number, it represents the first (internal) layer in the techFile. For this reason it is strongly recommended that you import GDS2 after importing a techFile containing layer names and GDS layer/datatype mappings.

If a GDS file is imported into an existing library containing cellViews, any existing cellView of the same name as a GDS2 struct (cell) will be handled by the Duplicate Cells setting.

GDS2 cells (STRUCTs) can be mapped to cellViews using cell mapping tab:

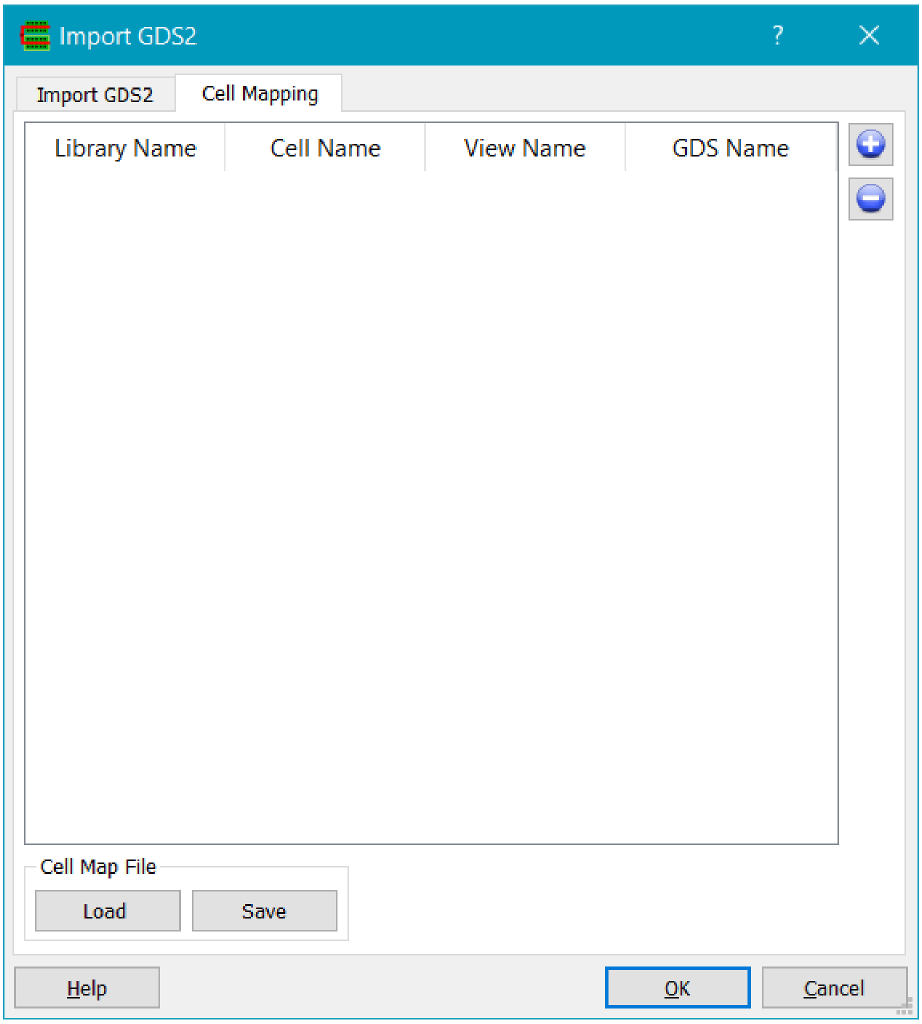


Figure 14 - Import GDS2

The + button adds an entry to the map table, the - button removed a selected entry. The GDS Name field specifies the GDS2 STRUCT name, and the Library Name, Cell Name and View Name specify the cellView to map this STRUCT to. The cell mapping can be loaded or saved to a file; the format is ascii and consists of 4 values per line (library name, cell name, view name and GDS name) separated by whitespace. The same format is used by the cell map table in the **File->Export->GDS2** command.

2.1.15 File->Import->OASIS

The **File->Import->OASIS** command displays the Import Oasis dialog.

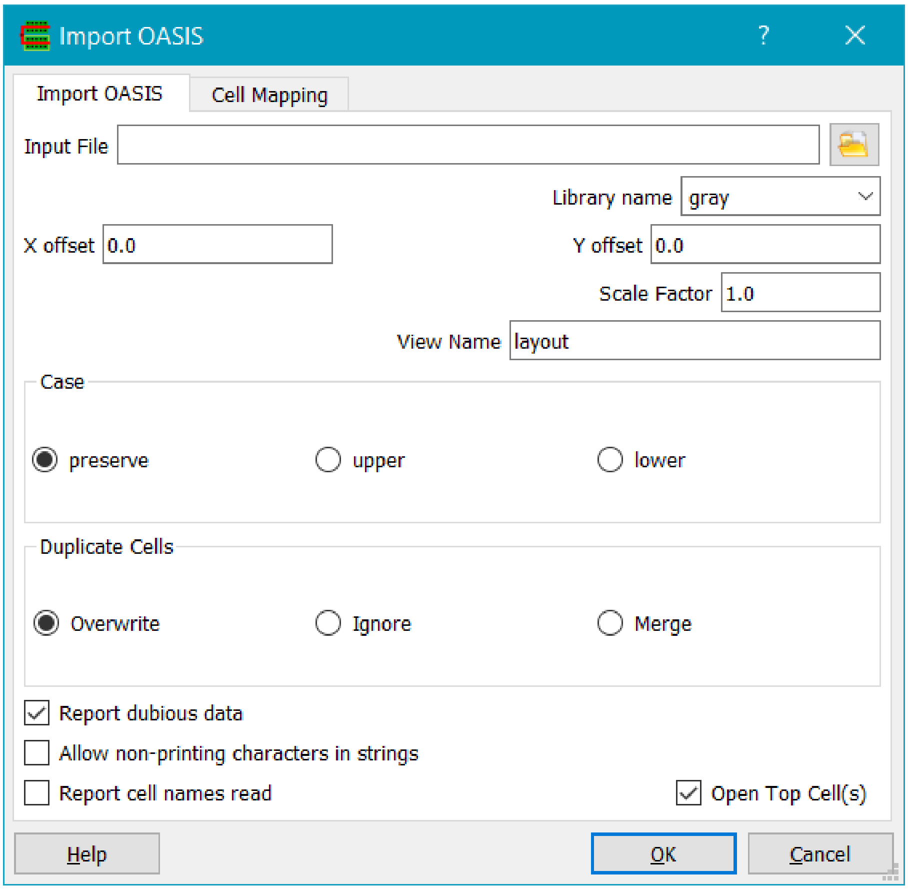


Figure 15 - Import OASIS

OASIS is a replacement for GDS2 with data compression to give much smaller file sizes. Typically 10-50x compression compared to GDS2 is achieved. The OASIS reader supports CBLOCK compressed records and both strict and non-strict mode OASIS files.

The OASIS *Input File* to be imported can be chosen using the file chooser button. A *Library name* to import the OASIS into MUST be specified, and will be created if it does not already exist. Multiple OASIS files can be read if they are entered separated by a comma.

An *X offset* and *Y offset* can be specified. The specified offsets are added to all coordinates in the design, in effect moving the origin of the design. Note that offsets are applied BEFORE any user-specified *Scale Factor*.

OASIS data can be scaled while read in if the *Scale Factor* field is set to a number other than 1. For example, if a scale factor of 0.5 is chosen, all coordinates will be multiplied by 0.5 and the design is shrunk by a factor of 2. This can be useful for scaling entire design databases.

The *View Name* specifies the view name created when a cell is imported. If cell mapping is used, this value will be overridden by the map library/cell/view names.

When cells are imported, if a cell of the same name exists you have 3 options available in the *Duplicate Cells* field. *Overwrite* means the new cell will replace the existing cell. *Ignore* will mean the new cell definition is ignored, along with all data in it. *Merge* means the original cell data is preserved, and any data in the new cell is added to it. This may cause duplicate shapes, but can be used to merge OASIS data.

Oasis cell and array names can have their *Case* preserved, forced to upper case or lower case depending on the 'Case ' radio buttons. Note that if you have a cell named 'AND2' and one called 'and2' and do not preserve case, then the second cell encountered will give rise to a duplicate cell.

If *Report dubious data* is checked, errors are reported for e.g. polygons with less than 3 vertices. If *Allow non-printing characters in strings* is checked, then any valid ascii character is allowed in e.g. text names; else only printing characters as defined in the Oasis spec are allowed.

Report cell names read will write each cell encountered in the input OASIS data. For large designs this can slow things down so by default it is turned off.

Open Top Cell(s) will attempt to identify and open cells that appear to be the 'top cell' of a OASIS file. A top cell is not referenced by other cells, and contains one of more cell placements.

At present the following OASIS constructs are silently ignored:

XNAME
XELEMENT
XGEOMETRY
PROPERTY

OASIS cells can be mapped to cellViews using cell mapping tab.

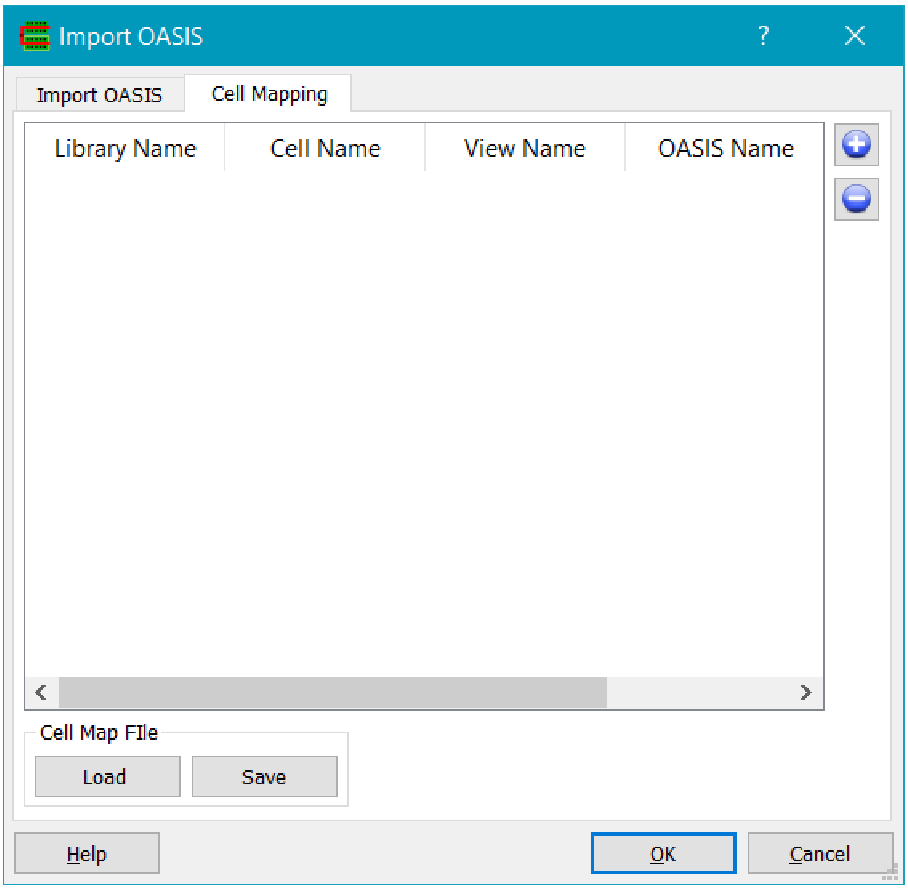


Figure 16 - Import OASIS

The + button adds an entry to the map table, the - button removed a selected entry. The OASIS Name field specifies the OASIS cell name, and the Library Name, Cell Name and View Name specify the cellView to map this name to. The cell mapping can be loaded or saved to a file; the format is ascii and consists of 4 values per line (library name, cell name, view name and OASIS name) separated by whitespace. The same format is used by the cell map table in the **File->Export->OASIS** command.

2.1.16 File->Import->LEF

The **File->Import->LEF** command displays the Import LEF dialog.

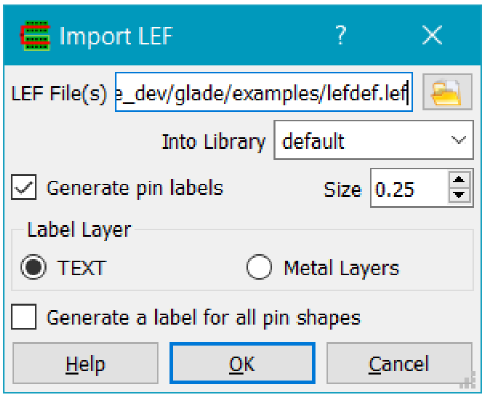


Figure 17 - Import LEF

The *LEF file(s)* to be imported can be chosen using the file chooser button. Multiple LEF files may be read by selecting each one in the file browser, or if they are entered separated by a comma. If the file name extension is '.gz' then the compressed file will be unzipped on the fly. *Into Library* specifies the library to import the LEF into, and will be created if it does not already exist. Multiple LEF files can be read, if subsequent LEF files redefine sites or macros previously defined they will be overwritten. A technology LEF should always be read first - this contains layer definitions for routing and cut layers. Note that all LEF files should have a VERSION statement to be valid LEF files.

If the LEF UNITS are larger than the database units (by default 1000 dbu/micron) e.g. 2000, then the library database units are changed to the LEF UNITS. For this reason one should ensure that the first LEF file read has the largest UNITS.

LEF Macros are imported as cells with a view type of 'abstract'. A rectangle on the system layer 'boundary' is created for each macro according to the macro's SIZE . LEF OBS statements create shapes on the 'boundary' purpose for that shape, and LEF PORT statements create shapes on the 'pin' purpose.

If the *Generate pin labels* option is set, text labels are created for the LEF pins on the system Text layer and can be displayed by making labels visible - see the Display Options command. *Size* sets the size of the generated labels. The labels are generated on a layer as specified by the *Label Layer* field; either the system layer TEXT purpose drawing or the same layer as the pin shape, but with purpose 'txt'. If *Generate a label for all pin shapes* is checked, multiple labels will be generated for each pin shape. This is not usually desirable for standard cells, but can be useful for large macros.

If a LEF file is imported into an existing library containing cellViews, any existing cellView of the same name as a LEF macro and view 'abstract' will NOT be overwritten.

2.1.17 File->Import->DEF

The File->Import->DEF command displays the Import DEF dialog.

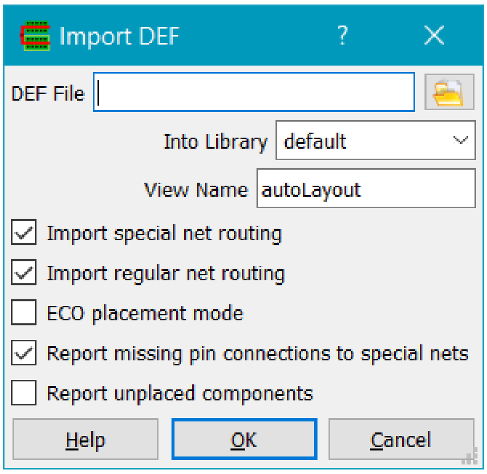


Figure 18 - Import DEF

The *DEF file* to be imported can be chosen with the file chooser button. If the file name extension is '.gz' then the compressed file will be unzipped on the fly. Into Library specifies the library to import the DEF into, and will be created if it does not already exist. LEF must have previously been imported to create abstract views for all components defined in the DEF COMPONENTS section; however you can set the *View Name* to 'abstract' when importing DEF to create an abstract for use by other DEF files, for example for a hierarchical design. A rectangle on the system 'boundary' layer is created according to the DEF DIEAREA statement.

If *Import special net routing* is checked, special net routing will be created in the design. If it is not checked only the connectivity information is imported. If *Import regular net routing* is checked, regular net routing will be created in the design. If it is not checked only connectivity information is imported. If *ECO placement mode* is checked, the DIAREA section of the DEF is updated, the COMPONENTS section of the DEF file will be parsed and instance origins and orientations of the current cellView will be updated, and the PINS section of the DEF will be parsed, replacing existing pins. All components in the ECO file must exist in the current open cellView. If *Report missing pin connections to special nets* is checked, then missing pin connections will be reported. If *Report unplaced components* is checked, the names of unplaced components will be reported as a warning, otherwise unplaced components will be silently imported.

Import DEF will expect all referenced macros to have been previously imported by the Import LEF command as abstract views. Macros can be either imported into the same library as the DEF, or in multiple libraries, in which case Import DEF will search the libraries to resolve instance masters. However there is a restriction in that DEF must be imported into a library that has had a technology LEF imported (this is so the library has layer information such as layer type of routing, cut etc. defined). Failure to do so will give rise to via layers not being correctly recognised.

If you are importing hierarchical DEFs, you need to import the child cell DEF files first and set the *View Name* to abstract. You should also import each child DEF into a unique library, which has its technology file and technology LEF already imported. The reason is that P&R tools create DEF viaRule vias with names that may not be unique between different DEF files (e.g. a typical viaRule via called M1M2GEN may have variants M1M2GEN_1, M1M2GEN_2 etc. created). So if you try and import multiple DEFs into a single library, you will most likely get duplicate via name warnings, and only the viaRule vias of the first DEF file will be used.

So for example a section of Python code to load sub block DEFs and a top level DEF could be:

```
from ui import *
gui = cvar.uiptr
gui.importTech("lib1", "my.tch")
gui.importLef("lib1", "tech.lef")
gui.importLef("lib1", "stdcells.lef")
gui.importTech("lib2", "my.tch")
gui.importLef("lib2", "tech.lef")
gui.importDef("lib2", "abstract", "block1.def")
gui.importTech("lib3", "my.tch")
gui.importLef("lib3", "tech.lef")
gui.importDef("lib3", "abstract", "block2.def")
# top level DEF
gui.importDef("lib1", "autoLayout", "top.def")
```

Note that if you import DEF which references multiple libraries created by importing LEF, all the LEF libraries must have the same LEF UNITS!

Import DEF creates a cellView with a cell name as defined by the DEF DESIGN keyword.

2.1.18 File->Import->Verilog

The **File->Import->Verilog** command displays the Import Verilog dialog.

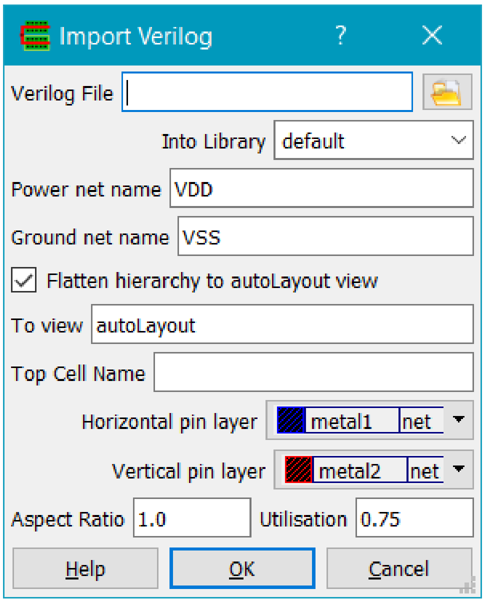


Figure 19 - Import Verilog

The *Verilog File* to be imported can be chosen with the file chooser button. Into Library specifies the library to import the Verilog into, and will be created if it does not already exist. The *Power net name* and *Ground net name* will be used to connect any logic 1 (verilog 1'b1) and logic 0 (verilog 1'b0) nets to. Verilog modules will be imported into the database as verilog views. Leaf cells must exist as abstract views (from Import LEF) for flattening to work. *If Flatten hierarchy to autoLayout view* is checked, the top cell as specified by *Top Cell Name* will be flattened into the view specified by *To view*, and Verilog leaf cells mapped to LEF cells of the same name. During the flattening process, instance pins on leaf cells are connected to the power and ground nets of the same name. Pins are created for inputs and outputs of the top level module. The pins will be on the *Horizontal pin layer* for pins on the left and right of the block and on *the Vertical pin layer* for pins on the top and bottom of the block. *Aspect ratio* sets the aspect ratio of the block; the number is the ratio of height to width. *Utilisation* sets the ratio of cell area to design boundary size. Rows are created in the design and cells are placed randomly in the rows, spaced by 2 times the site width.

Verilog modules are imported as cells with a view type of 'verilog' if not flattened.

Only basic structural level Verilog is supported. Simple ASSIGN statements are supported.

2.1.19 File->Import->ECO

The **File->Import->ECO** command displays the Import ECO dialog.

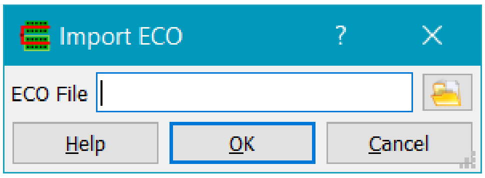


Figure 20 - Import ECO

This is used for importing an ECO file to make changes to the connectivity of the current open design.

An example of ECO file syntax is as follows. Lines beginning with a '#' are comments.

- Detach Pin AF|AFFF|U179.B from Net AF|AFFF|N351 ;
- Delete Pin AF|AFFF|U179.B ;
- Detach Pin AF|AFFF|U179.A from Net AF|AFFF|N356 ;
- Delete Pin AF|AFFF|U179.A ;
- Detach Pin AF|AFFF|U179.Y from Net AF|AFFF|N368 ;
- Delete Pin AF|AFFF|U179.Y ;
- Change Cell AF|AFFF|U179 from Model NOR2X1 to Model NOR2X2 ;
- Add Pin AF|AFFF|U179.B ;
- Attach Pin AF|AFFF|U179.B from Net AF|AFFF|N351 ;
- Add Pin AF|AFFF|U179.A ;
- Attach Pin AF|AFFF|U179.A from Net AF|AFFF|N356 ;
- Add Pin AF|AFFF|U179.Y ;
- Attach Pin AF|AFFF|U179.Y from Net AF|AFFF|N368 ;

2.1.20 File->Import->DXF

The **File->Import->DXF...** command displays the Import DXF dialog.

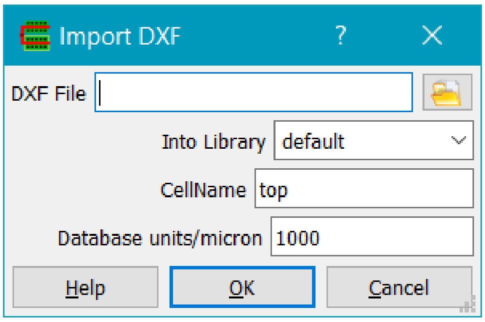


Figure 21 - Import DXF

DXF is a common drafting format. DXF file specifies the name of the DXF file to import; the file can be chosen with the file browser button. A library must be specified; it will be created if it does not already exist. A cell name to import the drawing into must also be specified; it defaults to 'top'. Hierarchical designs can be imported.

2.1.21 File->Import->EDIF

The **File->Import->EDIF...** command displays the Import EDIF dialog.

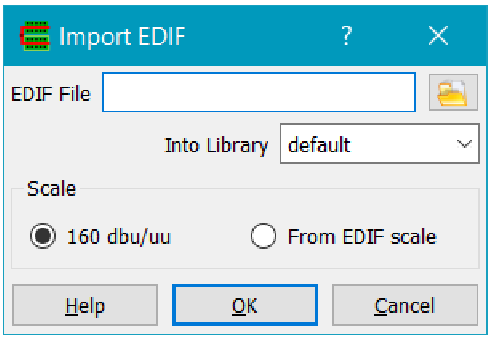


Figure 22 - Import EDIF

EDIF is a format for exchanging schematic and netlist data. *EDIF File* specifies the name of the EDIF file to import. *Into Library* specifies the library name to import to. *Scale* defines the resulting database units; *160dbu/uu* is typical for Cadence compatible schematics. *From EDIF scale* sets the database units per user unit (dbu/uu) to that defined by the EDIF numberDefinition entry.

When exporting EDIF from another CAD system, symbol libraries should be exported as externals in EDIF. Then, when importing EDIF into Glade, matching libraries should be opened before the import. The Glade symbol libraries will obviously need to have the same size symbols, with the same pin names/locations as the originals. Alternatively it is possible to export symbol libraries in EDIF and have them created in Glade.

Although EDIF is supposed to be a 'standard', interpretation is another matter and how design data is exported is very much vendor-dependent.

2.1.22 File->Export->TechFile

The **File->Export->TechFile** command displays the Export TechFile dialog.

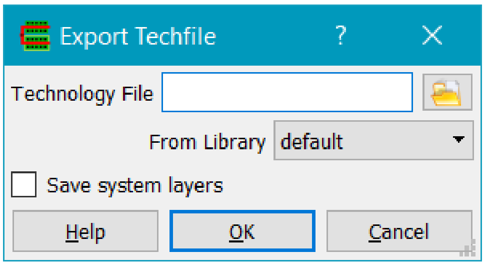


Figure 23 - Export TechFile

Technology File specified the name of the technology file to write and can be chosen using can be selected by using the file chooser button. *From Library* specifies a library to export the techFile from. If *Save system layers* is checked, they will be written to the techFile. This is only necessary if you do not want to use the default layer colours e.g. if you want a white background, you need to set the 'backgnd' system layer colour to white, and set the 'select' colour to something other than white, and e.g. the 'cursor' colour to something other than yellow etc.

2.1.23 File->Export->GDS2

The File->Export->GDS2 command displays the Export GDS2 dialog.

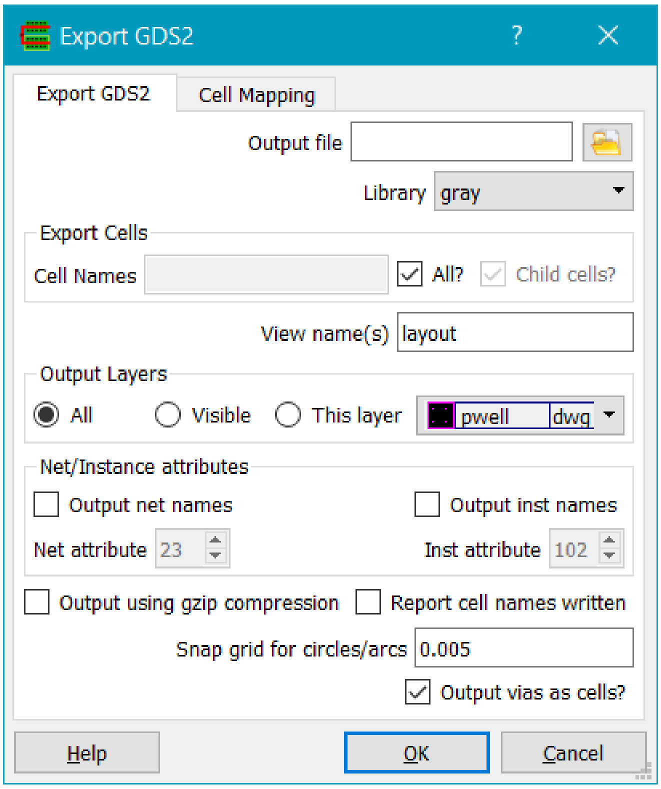


Figure 24 - Export GDS2

Output file specifies the name of the GDS2 output file and can be selected by using the file chooser button. *From Library* specifies a library to export GDS2 from.

Export Cells controls which cells are output. If you want to output only certain cells in the design, specify them in the *Export Cells Cell Names* field and uncheck the *All?* button. Else if *All?* is checked then all cells in the library will be exported. If *Child Cells?* is checked, then cells are exported to match the instances in the design hierarchy being exported.

The *View name(s)* field allows you to specify what views are exported. It is populated by default with all the view names found for the library. The view names can be delimited by whitespace. Note for example if you want to output from a LEF/DEF top level cellView, you will need to specify autoLayout (the view name of the DEF top level), abstract (the view name of the LEF cells) and layout (the view name of the vias). If your design contains cells with multiple views of viewType layout, then this field is automatically populated with the view names.

Cells specified in the *Export Cells* field will be output with all child cells i.e. the complete hierarchy will be output, thus the resulting GDS2 will be complete, as long *as Child Cells?* is checked.

Output Layers allows you to control which GDS layers are exported. *All* will output all layers, *Visible* will output layers currently set visible in the LSW, and *This Layer* will only output a specific layer chosen by the layer chooser.

Instance names can be output as properties with the default GDS2 attribute number 102 if *Output inst names* box is checked. Net names of shapes can be output with default GDS attribute number 23 if the *Output net names* box is checked. These numbers are arbitrary and can be changed as desired.

If *Output using gzip compression* is checked, the GDS2 data is compressed using the gzip algorithm. If *Report cell names written* is checked, cell names are output to the message window as they are written. *Snap grid for circles/arcs* snaps the vertices of arcs/circles to the specified grid in microns. Circles are output as GDS boundaries and lines/arcs as zero width paths.

Output vias as cells writes vias as cell instances with the cell master name equal to the instance name. This is typically useful for LEF/DEF where you don't want to flatten the vias into their individual shapes.

GDS2 cells (STRUCTs) can be mapped from cellViews using the cell mapping tab.

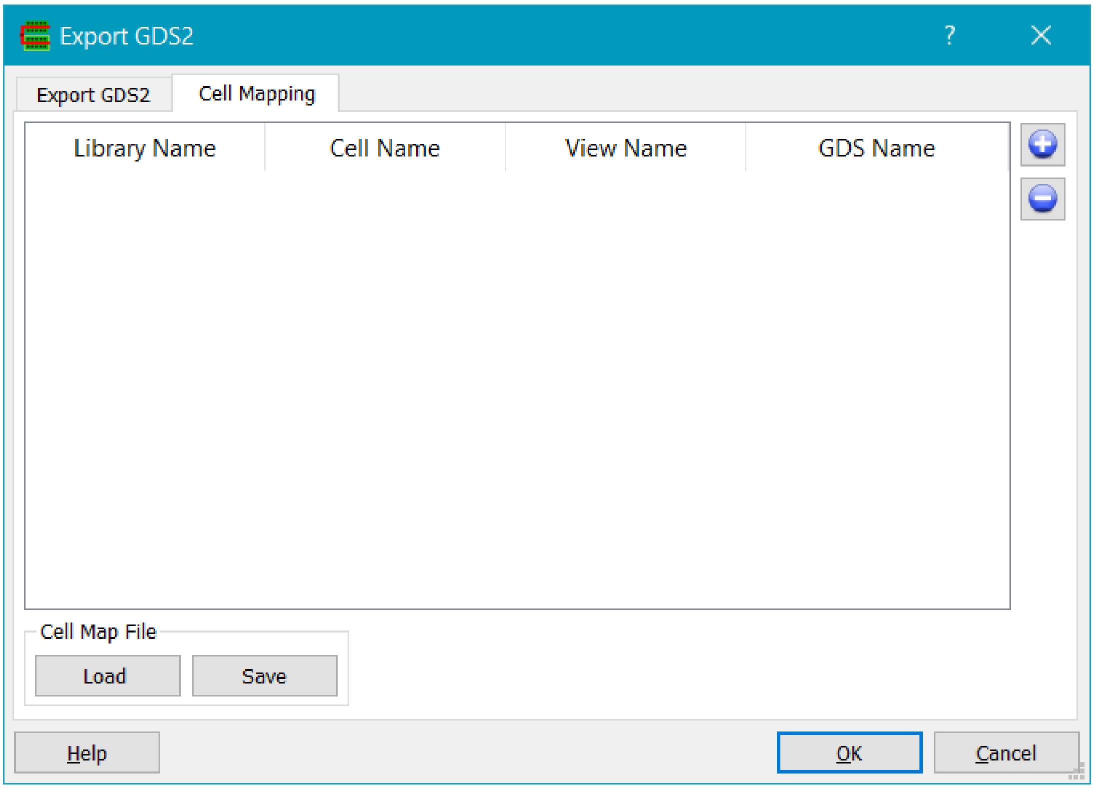


Figure 25 - Export GDS2

The + button adds an entry to the map table, the - button removed a selected entry. The GDS Name field specifies the GDS2 STRUCT name, and the Library Name, Cell Name and View Name specify the cellView to map to this STRUCT. The cell mapping can be loaded or saved to a file; the format is ascii and consists of 4 values per line (library name, cell name, view name and GDS name) separated by whitespace. The same format is used by the cell map table in the File->Import->GDS2 command. The map table is automatically populated with potentially conflicting cell/view names that would normally map to the same GDS2 STRUCT name. In this case each cell/view combination will have a map table entry, with an auto generated GDS2 name which is of the form <cellname>_01, <cellname>_02 etc.

2.1.24 File->Export->OASIS

The File->Export->OASIS command displays the Export OASIS dialog.

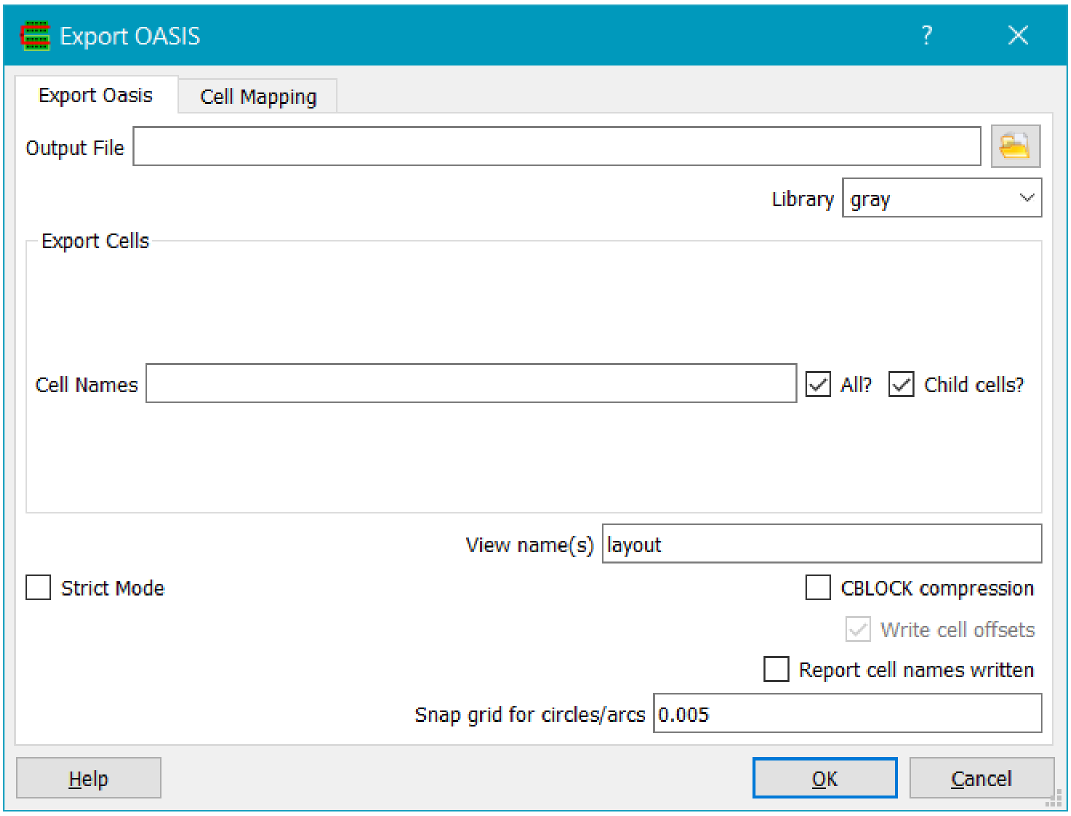


Figure 26 - Export OASIS

Output File specifies the OASIS output file name and can be selected by using the file chooser button. *From Library* specifies the library to export OASIS from.

Export Cells controls which cells are output. If you want to output only certain cells in the design, specify them in the *Export Cells Cell Names* field and uncheck the *All?* button. Else if *All?* is checked then all cells in the library will be exported. If *Child Cells?* is checked, then cells are exported to match the instances in the design hierarchy being exported.

The View name(s) field allows you to specify what views are exported. The view names can be separated by a comma or a space. They are populated by default from the views found in the library.

If *Strict Mode* is checked, names of cells, text strings, layers, property names and property strings are collected together into tables and referenced by an offset in the END record as per the OASIS standard. In *Strict mode*, if *Write cell offsets* is checked, the property S_CELL_OFFSET is written for each cell in the cell name table so that random access to cells are possible allowing e.g. multithreaded reading of the OASIS file. If *CBLOCK compression* is checked, strict mode tables and cell data is compressed using RFC1951 compression. This can result in significant reductions in file size.

If *Report cell names written* is checked, cell names are output to the message window as they are written. *Snap grid for circles/arcs* snaps the vertices of arcs/circles to the specified grid in microns. Circles are output as OASIS polygons and lines/arcs as zero width paths.

OASIS cells can be mapped from cellViews using cell mapping tab.

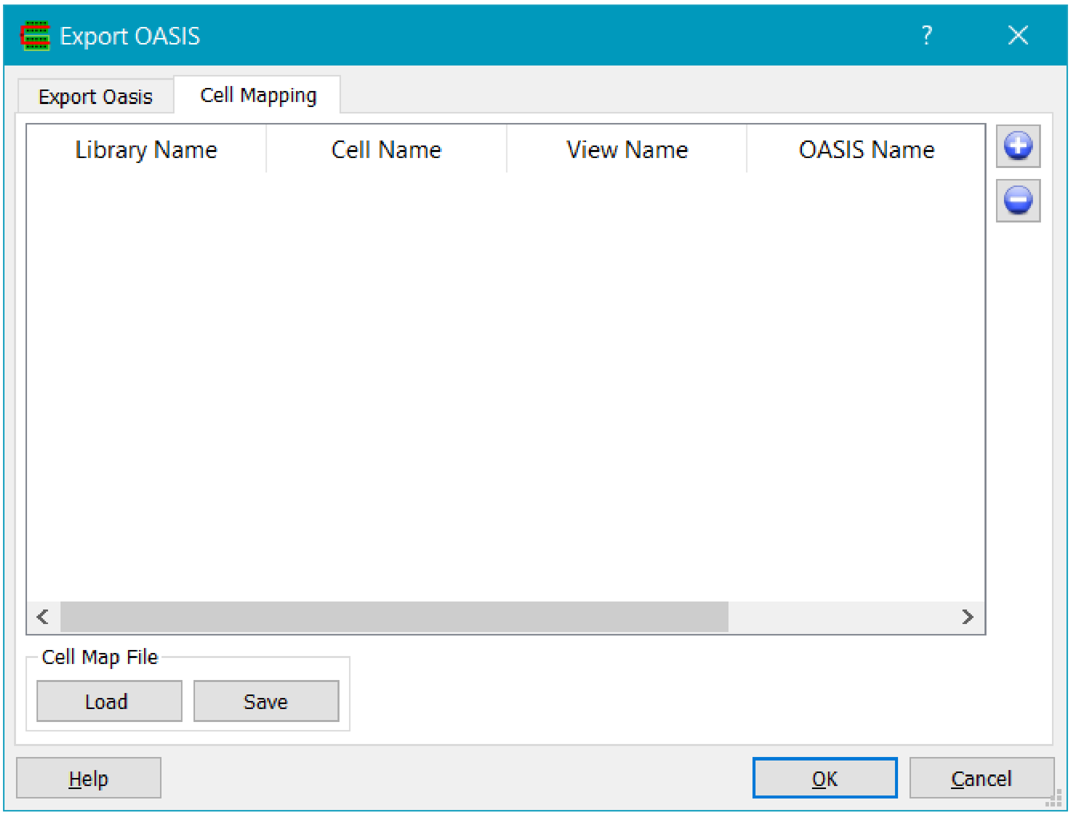


Figure 27 - Export OASIS

The + button adds an entry to the map table, the - button removed a selected entry. The OASIS Name field specifies the OASIS cell name, and the Library Name, Cell Name and View Name specify the cellView to map to this cell. The cell mapping can be loaded or saved to a file; the format is ascii and consists of 4 values per line (library name, cell name, view name and OASIS name) separated by whitespace. The same format is used by the cell map table in the File->Import->OASIS command. The map table is automatically populated with potentially conflicting cell/view names that would normally map to the same OASIS cellname. In this case each cell/view combination will have a map table entry, with an auto generated OASIS cellname which is of the form <cellname>_01, <cellname>_02 etc.

2.1.25 File->Export->LEF

The File->Export->LEF command displays the Export LEF dialog.

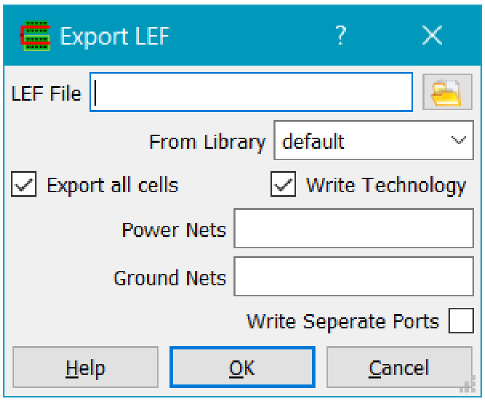


Figure 28 - Export LEF

LEF File specifies the file to export in the 'LEF file' field which can be set using the file chooser button. *From Library* specifies the library to export from. Either all cells can be written, if *Export all cells* is checked, or just the currently open cell. If the *Write Technology* is checked, then the LEF technology section is written (layer widths/spacings, vias definitions etc). *Power Nets* specifies power pins in the LEF macros that should have their USE set to POWER. *Ground Nets* specifies ground pins in the LEF macros that should have their USE set to GROUND. *Write Separate Ports* writes each port shape as a separate PORT definition in the LEF.

2.1.26 File->Export->DEF

The **File->Export->DEF** command displays the Export DEF dialog.

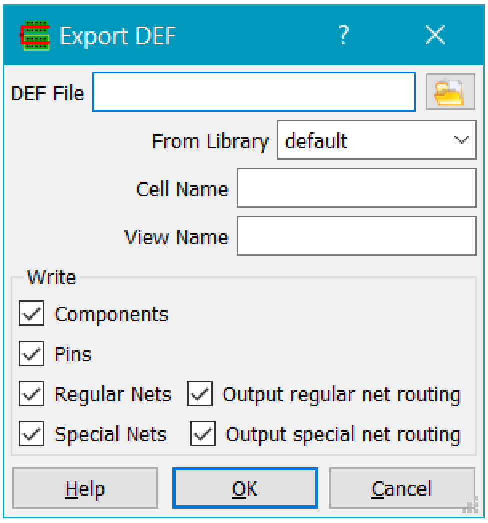


Figure 29 - Export DEF

DEF File specifies the file name to export to and can be set using the file chooser button. *From Library* specifies the library to export from. The library, *Cell Name* and *View Name* will default to the current open cellView.

You may selectively write parts of the DEF file by checking or unchecking the *Components*, *Pins*, *Regular Nets* and *Special Nets* check boxes. For example DEF with just placement information would require just the Components and Pins checked. You can also choose to write just connectivity of nets, or the physical shapes as well if *Output regular net routing* / *Output special net routing* is checked.

2.1.27 File->Export->Verilog

The **File->Export->Verilog** command displays the Export Verilog dialog.

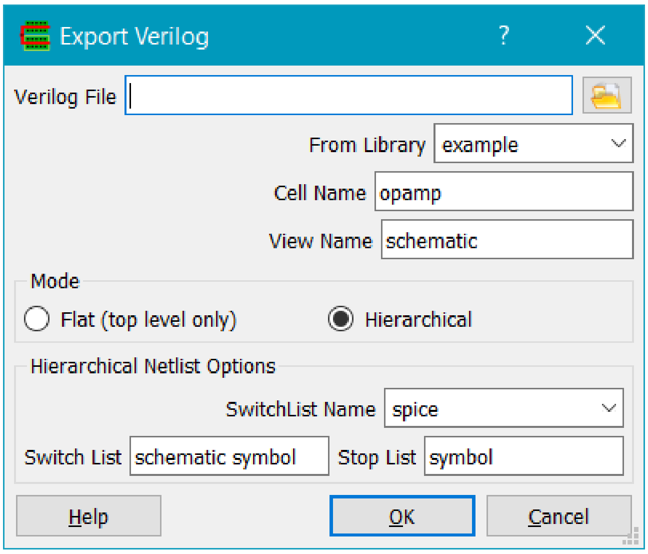


Figure 30 - Export Verilog

Verilog File specifies the file to export to and can be set using the file chooser button. *From Library* specifies the library name. The library, *Cell Name* and *View Name* fields are pre-seeded with the currently open cellView. Note that Verilog can only be exported from a cellView that has connectivity. If *Mode* is set to *Flat*, the Verilog netlist will be a flat representation of the top level design, else it will be hierarchical. *Switch List* and *Stop List* set the switch and stop lists for the netlister during hierarchical netlisting, and are space-delimited lists of view names. Switch and stop lists are named in *SwitchList Name*. To create a new name group, edit the *SwitchList Name* and set the Switch List and Stop List. The new named group will be saved in the gladerc.xml preferences file.

2.1.28 File->Export->DXF

The **File->Import->DXF** command displays the Export DXF dialog.

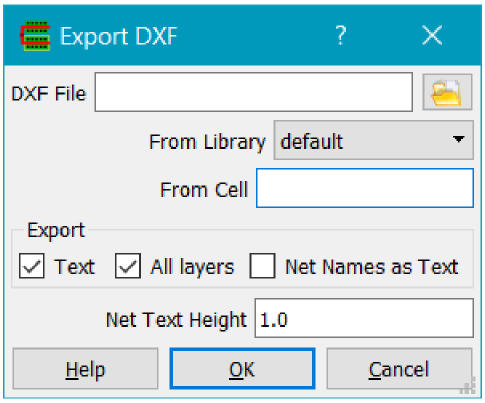


Figure 31 - Export DXF

DXF is a common drafting format. *DXF file* specifies the name of the DXF file to export; the file can be chosen with the file browser button. *From Library* and *From Cell* default to the current open cellView. If the cell contains hierarchy, subcells are also exported. If *Export Text* is checked, text labels are output to the DXF file. If *All layers* is checked, all the cell's layers are output; if not, only the currently visible layers will be output. *Net Names as Text* will output net names as text to the DXF file. *Net Text Height* sets the text label height.

2.1.29 File->Export->CDL

The **File->Export->CDL...** command displays the Export CDL dialog.

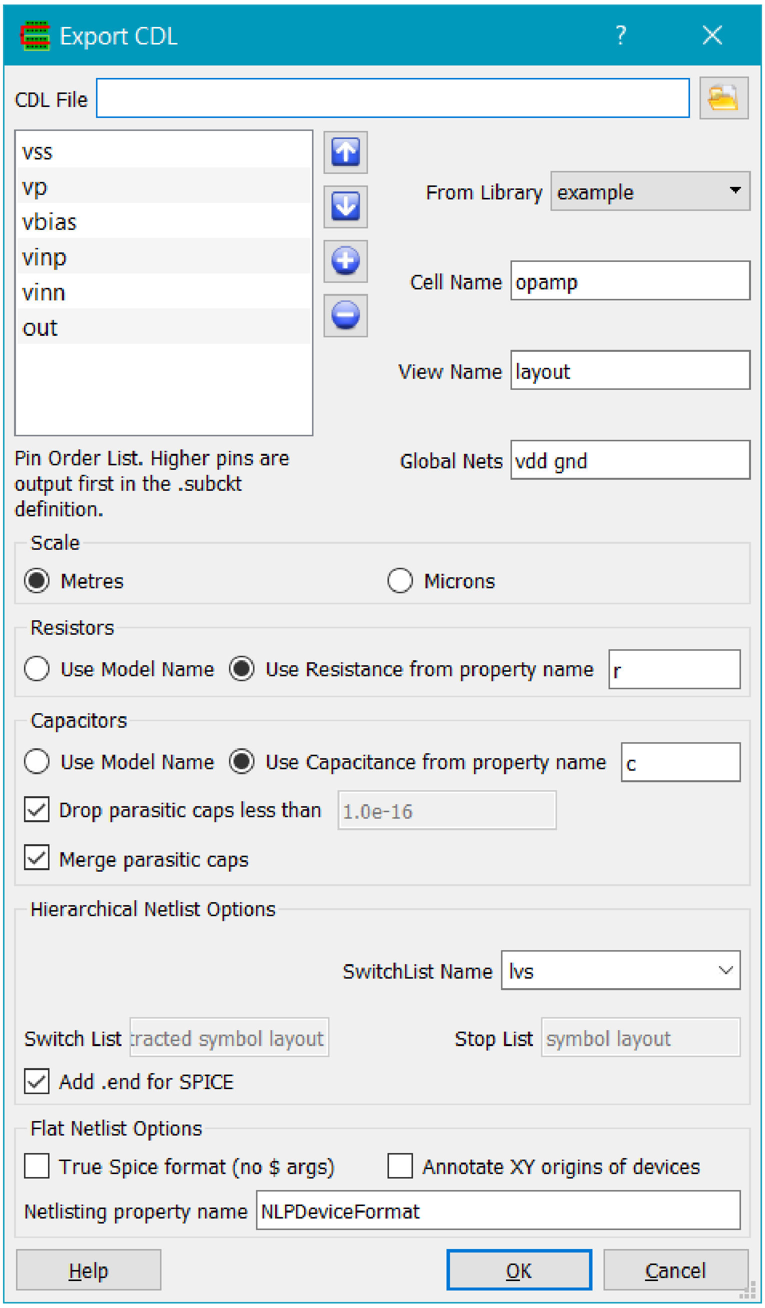


Figure 32 - Export CDL

The Export CDL dialog can be used to write a flat netlist from a layout/extracted view, or a hierarchical netlist from a schematic view. CDL is a spice like netlist format with some extensions over spice syntax.

CDL File specifies the name of the CDL file to export; the file can be chosen with the file chooser button. *From Library*, *Cell Name* and *View Name* default to the current open cellView. *The Pin Order List* shows the order pins will be written in the extracted netlist .subckt header. This is so the user can match the pin order to a simulation testbench etc. For a flat netlist the pin order can be changed by clicking on a pin name and using the up/down arrow buttons to move the pin; pins are written in the order of the list from top to bottom. For a hierarchical netlist, the pin order is obtained from the NLPDeviceFormat property on the symbol view of the top level cellView. *Global Nets* defines nets that should be global in the CDL netlist. They should be separated by a space character as delimiter. *Scale* determines the scale of the units written to the CDL file. For *Resistors*, *Use Model Name* specifies that the resistor model name should be output to the CDL file. *Use Resistance from property name* specifies that the resistance, as given by the value of the property name, is output to the CDL file rather than the model name. For *Capacitors*, *Use Model Name* specifies that the capacitor model name should be output to the CDL file. *Use Capacitance from property name* specifies that the capacitance, as given by the value of the property name, is output to the CDL file rather than the model name. If *Drop parasitic caps less than* is checked, all parasitic caps less than the specified value (in Farads) will not be output to the CDL file. If *Merge parasitic caps* is specified, multiple parasitics between two unique nets will be merged into a single lumped cap between the nets. For a hierarchical netlist from a schematic, the *Switch List* and *Stop List* control the netlist hierarchy traversal. The *Switch List* is a list of view names that the netlister can descend into. The *Stop List* is a list of views that the netlister will stop descending into, and instead write the device to the netlist according to its NLPDeviceFormat property. The *Switch List* and *Stop List* have no effect for layout view types. Switch and stop lists are named in *SwitchList Name*. To create a new name group, edit the *SwitchList Name* and set the Switch List and Stop List. The new named group will be saved in the gladerc.xml preferences file. *Add .end for SPICE* will add a .end line as the last line of the netlist, useful if you are netlisting a schematic for Spice simulation. *True Spice format* will write the netlist in SPICE compatible format, with no $ arguments. *Annotate XY origins of devices* annotates the XY coordinate of the device origin as $X=, $Y=. *Netlisting property name* is the name of a NLP expression property on the instance masters that will be used to control netlist formatting. It defaults to NLPDeviceFormat. If not present, the netlister will use a default suitable for Spice/CDL.

2.1.30 File->Export->EDIF...

The **File->Export->EDIF...** command displays the Export EDIF dialog.

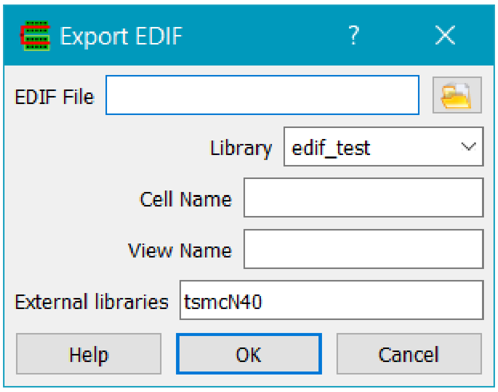


Figure 33 - Export EDIF

EDIF File specifies the file name to export to. *Library* , *Cell Name* and *View Name* set the design to export, which defaults to the open cellView. *External libraries* specifies reference libraries that will not be exported as libraries in EDIF, but as an external construct.

2.1.31 File->Print...

The **File->Print...** command prints the current design. The system printer options form is displayed allowing the user to specify paper size, landscape/portrait mode etc. The design is printed directly as it appears onscreen, so e.g. rulers etc. will be rendered. A white background should be chosen for printing on normal paper, and layer colours chosen carefully to give best results.

2.1.32 File->Export Graphics...

The **File->Export Graphics...** command dumps the current window to a PNG, JPEG or SVG format file. PNG format is smaller and has superior image quality to JPEG, at least for layout data. SVG (Scalable Vector Format) can be scaled and/or zoomed without loss of image quality and is more suitable for schematics/symbols.

2.1.33 File->Run Script...

The File->Run Script... command runs a python script. Python output is written to the message dock window.

2.1.34 File->Edit ascii file...

The **File->Edit ascii file...** command opens a file chooser dialog and allows you to view and make simple edits to any ascii file.

2.1.35 File->Exit

The **File->Exit** command exits Glade. Any designs opened are checked for changes before exiting. If there are cells which have been edited, a list of the edited cells is displayed in the Save Cells dialog.

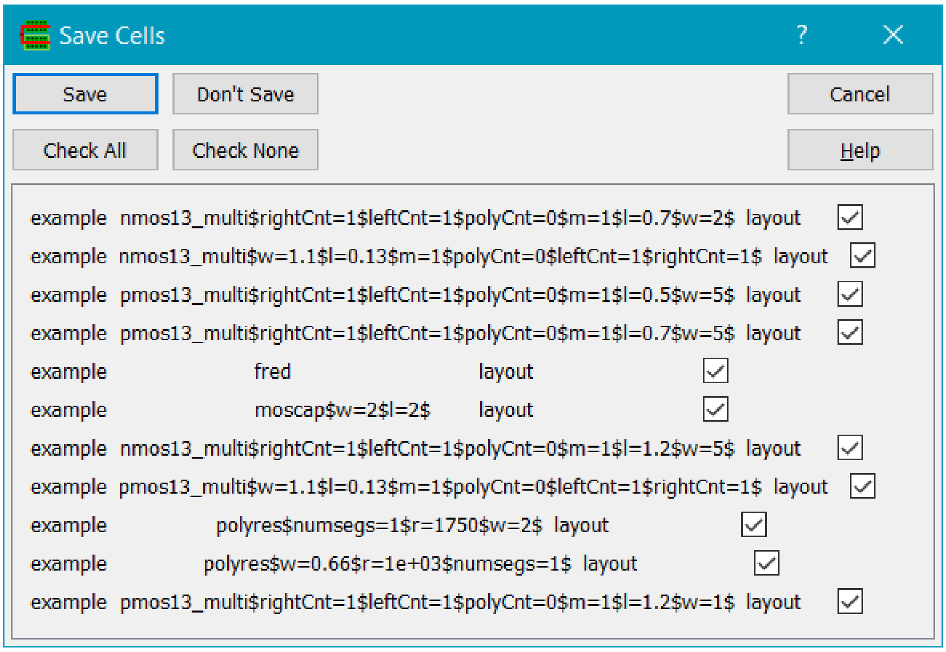


Figure 34 - Save Cells

If *Save* is clicked, all checked cells are saved and the program exits. If *Don't Save* is clicked, no cells are saved and the program exits. If *Cancel* is clicked, no cells are saved and the program does not exit. If *Check All* is clicked, all the cells in the cell list are checked. If *Check None* is clicked, all the cells in the cell list are unchecked.

The Save Cells dialog is also displayed if the Glade window is closed via the window manager close button and there are edited cells that are unsaved, and may be displayed if a crash occurs and Glade is able to perform an orderly shutdown.

2.2 The Tools Menu

2.2.1 Tools->LSW

The **Tools->LSW** command toggles the display of the LSW.

The LSW (Layer Selection Window) is used to control layer display in Glade. It comprises a dockable dialog box with a scrollable panel of layers - one for each layer defined in the technology file - plus some system defined layers. Each layer in the LSW has 3 parts: a color box on the left which displays the layer line and fill style; a layername box in the centre which displays the layer's name, and a purpose box on the right which displays the layer's purpose, abbreviated to 3 characters (for example 'drawing' becomes 'dwg', 'pin' becomes 'pin', 'boundary' becomes 'bdy' and 'net' is represented as 'net').

The LSW shows user-defined layers and the system layers. System layers include the following:

- Layers y0-y9, used for temporary display purposes
- Layers annotate (purpose drawing, drawing1-9), used for schematic/symbol labels
- mpp - Used internally for MPP objects. Do not draw on this layer.
- boundary - used for cell boundaries for LEF cells and the DEF design boundary
- region - used to display DEF regions
- row - used to display rows from DEF
- marker - used for flagging DRC errors
- device - used for symbol shapes
- wire - used for schematic wires
- pin - used for schematic and symbol pins
- text - used for autogenerated text labels e.g. as a result of importing LEF
- hilite - used for displaying flightlines e.g. for connectivity
- select - used to highlight selected objects
- mingrid - used to draw the minor grid
- majgrid - used to draw the major grid
- axes - used to draw the axes
- cursor - used for the box or crosshair cursor
- viaInst - for via instances that are shown unexpanded
- instance - for instances that are show unexpanded
- backgnd - the background display colour (defaults to black, but can be set to any colour)

At the top of the LSW are four buttons NS (None selectable), NV (None visible), AS (All Selectable), AV (All visible) which allow all layer selectability/visibility to be set at once. Below this are 4 buttons M1 (save to memory 1), R1 (recall from memory 1), M2 (save to memory 2) and R2 (recall from memory 2). These allow the current layer selectability / visibility to be saved and recalled for frequent changes. As changes are made that affect the display (changing colour, fill pattern or layer visibility) the display is automatically updated.

The LSW also has a menu bar, the menu bar has the Edit menu and Display menu. Edit commands are as follows.

2.2.1.1 Edit->Create Layer

The **Edit->Create Layer** command displays the Create Layer dialog box.

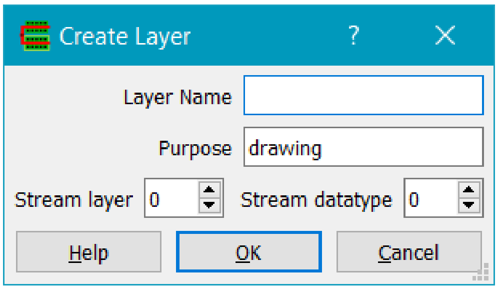


Figure 35 - Create Layer

Layer Name specifies the name of the layer to be created. *Purpose* specifies the purpose of the layer. The default is 'drawing'. *Stream Layer* sets the layer number used in GDS file import/export. *Stream datatype* sets the datatype number used in GDS file import/export.

2.2.1.2 Edit->Delete Layer

The Edit->Delete Layer command displays the Delete Layer dialog box.

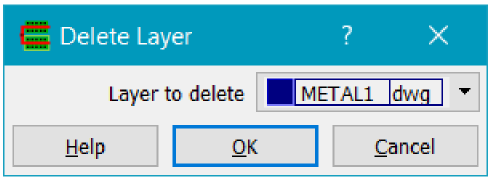


Figure 36 - Delete Layer

Layer to delete specifies the name of the layer to be deleted. A check is made of all cells in the library to see if there are any shapes on that layer. If there are, a dialog is shown asking the user if they really want to delete the layer, as all shapes in all cells in the library on that layer will be deleted, along with the layer itself. If the user chooses to cancel, the layer is not deleted, nor are any shapes on that layer.

Display commands are as follows:

2.2.1.3 Display->Show User Layers

The **Display->Show User Layers** command toggles the display of the user definable layers in the LSW. The menu item is checked when user layers are displayed.

2.2.1.4 Display->Show System layers

The **Display->Show System Layers** command toggles the display of the system layers in the LSW. The menu item is checked when system layers are displayed.

2.2.1.5 Display->Show Valid layers

The **Display->Show Valid Layers** command toggles the display of the valid layers in the LSW. The menu item is checked when valid layers are displayed.

2.2.1.6 Display->Show CellView Layers

The **Display->Show CellView Layers** command shows only the layers used in the current open cellview.

2.2.1.7 Display->Show Viewport layers

The **Display->Show Viewport Layers command** shows only the layers that currently overlap the viewport.

2.2.1.8 Display->Show All Layers

The **Display->Show All Layers** command shows all the layers defined in the techFile.

2.2.1.9 Display->System layers visible

The **Display->System Layers Visible** command makes the system layers visible.

2.2.1.10 Setting Layer Colours.

Left clicking the mouse on the colour box allows layer colours to be set. A left mouse button (LMB) click brings up the colour selection palette (the OS native colour picker is used)

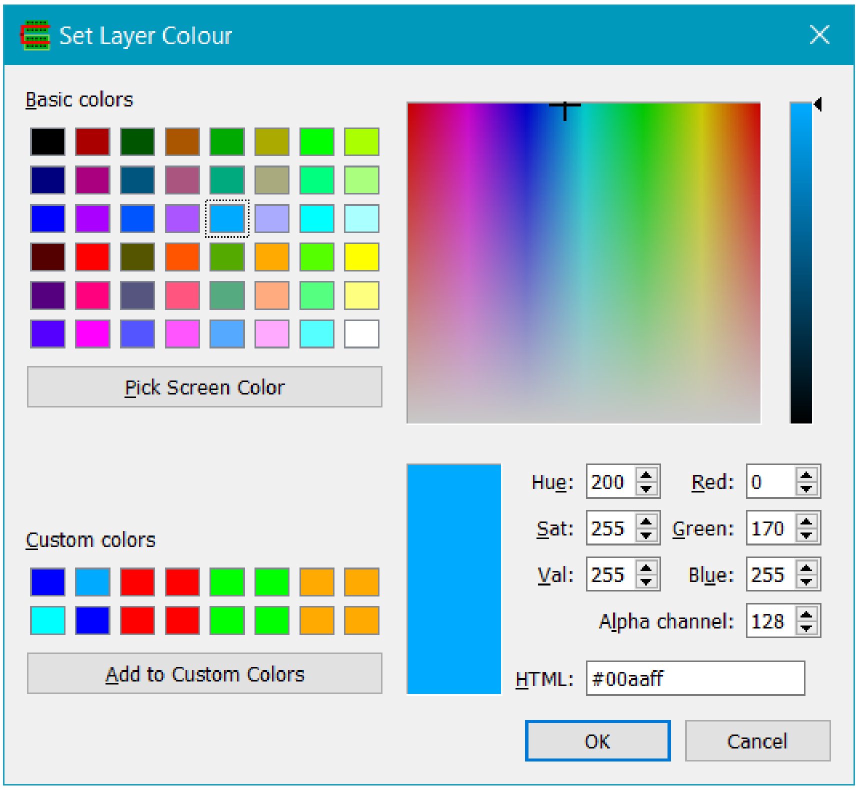


Figure 37 - Set Layer Colour

The layer colour can be chosen by clicking on the desired colour or typing in RGB or HSV numbers. The Alpha channel controls layer transparency. A value of 255 sets a layer opaque, values less than this make the layer transparent.

2.2.1.11 Setting layer Stipple Patterns

Right clicking the mouse on the colour box allows layer fill and line styles to be set. A right mouse button (RMB) click brings up the stipple pattern editor for that layer:

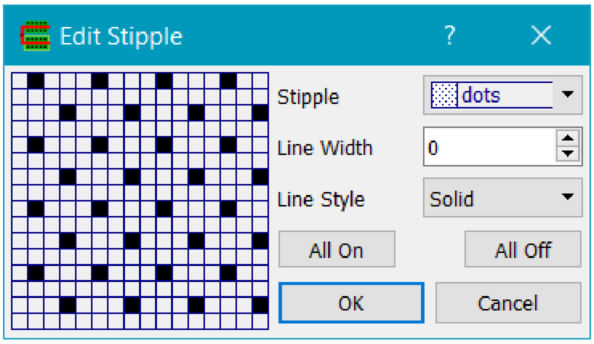


Figure 38 - Edit Stipple

The stipple pattern of the layer can be either edited manually by left mouse clicking in the 16 x 16 grid to toggle the pixels, or an existing stipple pattern can be chosen from the *Stipple* combo box, and then clicking the OK button. Clicking on the *All On* button turns on all pixels; clicking on the *All Off* button turns off all pixels. *Line Width* sets the line width of the border of the fill pattern - a value of 0 means use a single pixel line. *Line Style* sets the linestyle e.g. solid, dotted, dashed etc.

2.2.1.12 Setting layer Selectability and Visibility

Using the mouse on the LSW's layer box allows visibility and selectability to be toggled. Middle mouse button clicking (MMB) on the layername box toggles selectability On the Mac, there is no middle mouse button, so shift-right mouse button can be used instead.. Whan a layer is not selectable, the layer widget for that layer is grayed out in the LSW. Right mouse button clicking (RMB) on the layername box toggles layer visibility. When a layer is invisible, its color box is hidden.

2.2.1.13 Setting the current layer

Left mouse button clicking on the LSW's layername box makes that layer the current editing layer. A rectangle in brown highlights the current layer in the LSW. The current layer is used by the Create Label, Create Path, Create Polygon, Create Rectangle, Create Circle, Create Ellipse and Create Arc commands.

2.2.1.14 Setting the layer name

Double left mouse button clicking on the LSW's layer name box brings up a dialog box that allows the layer name to be edited.

2.2.1.15 Setting the layer purpose name

Double middle mouse button clicking on the LSW's layer name box brings up a dialog that allows the layer purpose name to be edited.

2.2.1.16 Querying layer properties

Double right mouse button clicking on the LSW's layer name box brings up a dialog box showing the layer properties. Currently the layer properties that can be changed are:

- GDS2 layer number
- GDS2 datatype
- Layer minimum width
- Layer minimum space
- Layer Pitch

- Layer Direction
- Layer Resistance
- Layer Area Cap
- Layer Edge Cap
- Layer Order
- Layer Dim Factor (%)

2.2.1.17 Setting the layer order

Layers are drawn in the order they are shown in the LSW, from the top down. The default layer order is the same as the order in the techFile. Layer order can be changed by left clicking and dragging a layer in the LSW to a new location. If the techFile is exported the layers will be written in the new order.

2.2.2 Tools->Message Window

The **Tools->Message Window** command toggles the display of the Output message window. The Output message window can be used for entering Python commands and displays messages from Glade. All messages in the Output window are written to the log file, called glade.log.

2.2.3 Tools->Library Browser

The Tools->Library Browser command toggles the display of the library browser.

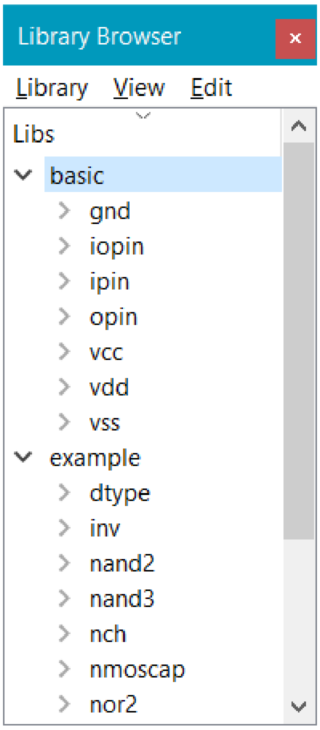


Figure 39 - Library Browser

Use the library browser to open cellViews, rename cellViews, copy cellViews or delete cellViews. The library browser shows the library name(s), the cell names and their view names as a tree. You can interact with the library browser using the left and right mouse buttons. Left clicking on an entry will expand or collapse that entry; left double clicking on a view name opens that cellView.

If you right click over the name of a library, a popup menu is displayed with the following menu items:

- **Save Library** - saves the library.
- **Save Library As...** - saves the library to a new disk file.
- **Close Library** - closes the library.
- **Rename Library** - renames the library.
- **Create CellView** - creates a new cellView in the library.
- **Tree View** - toggles tree view vs flat view display of the library contents.
- **Case Sensitive** - toggles case sensitive sorting of cell names.
- **Find...** - Allows you to search for a cell by name
- **Refresh** - Refreshes the library browser's contents.

If you right click over a cell name, a popup menu is displayed with the following menu items:

- **Delete Cell** - deletes the cell and all its views from the library.
- **Rename Cell** - renames the cell and updates all references to it and its cellViews within the library.
- **Copy Cell** - copies the cell and all its views to a new library/cell.

If you right click over the view name of a cell, a popup menu is displayed with the following menu items:

- **Open CellView** - opens the cellView
- **Delete CellView** - deletes the cellView and purges it from memory. Note this does not delete the cell in the library saved to disk.
- **Rename CellView** - renames the cellView and updates all references to it within the library.
- **Copy CellView**- copies the cellView to a new cell and/or view name
- **Properties** - displays the cellView’s properties.

The library browser has the following menu commands:

- **Library->New Lib** - Creates a new library.
- **Library->Open Lib** - Opens a library.
- **Library->Save Lib As...** - Saves a library to disk.
- **Library->Exit** - exits the library browser and closes the window.
- **View->Refresh** - Refreshes the library browser display.
- **View->Tree View** - toggles tree view vs flat view display of the library contents.
- **View->Case Sensitive** - toggles case sensitive sorting of cell names.
- **Edit->Find...** - Allows you to search for a cell by name.

2.2.4 Tools->Hierarchy Browser

The **Tools->Hierarchy Browser** command displays the current edit cell's hierarchy in the hierarchy browser dock window.

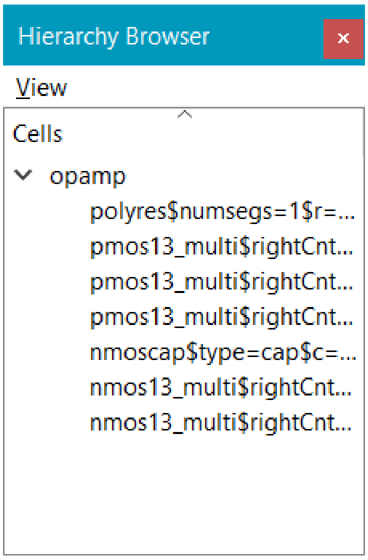


Figure 40 - Hierarchy Browser

The Hierarchy browser shows the design hierarchy. The root cell is shown with its subcells displayed by their cell names. In addition to expanding or collapsing the list items by left clicking on the 'v' boxes, several other operations can be performed.

Left mouse double clicking on any cell name will descend into that cell and it will be displayed in the browser as the new root cell.

Right mouse clicking on any subcell name will show that instance's properties (and hence the instance name).

Right mouse clicking on the root cell name will show a popup menu with the following menu items:

- **Ascend** - ascends to the parent cell of the current cell.
- **Refresh** - refreshes the hierarchy browser.

The hierarchy browser has the following menu commands:

- **View->Inst View** – toggles the hierarchy browser between display of instance names and cell names.
- **View->Case Sensitive** – toggles case sensitivity.
- **View->Refresh** – refreshes the hierarchy browser contents.

2.2.5 Tools->Net Browser

The Tools->Net Browser command displays the current cell's nets in the net browser dock window.

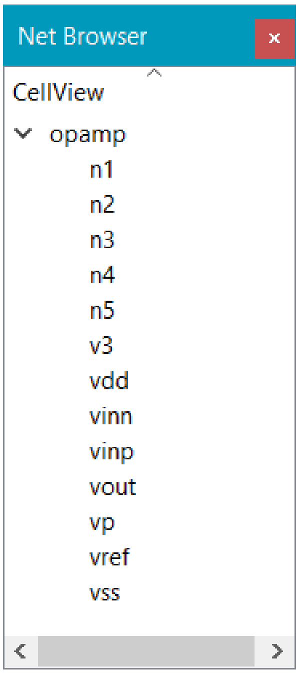


Figure 41 - Net Browser

The cell name is shown as the root with its nets displayed.

Left mouse double clicking on a net will select all shapes of the net.

Right mouse button clicking on a net name will display a popup menu with the following menu items:

- **Select All Insts** - Selects all the instances that connect to the net.
- **Select Driver Inst** - Selects the instance(s) that have output pin(s) connected to the net, i.e. drive the net.
- **Select Load Insts** - Selects the instances that have input pins connected to the net, i.e. are loads of the net.

2.2.6 Tools->Add Marker

The **Tools->Add Marker** command adds a marker at a specified location.

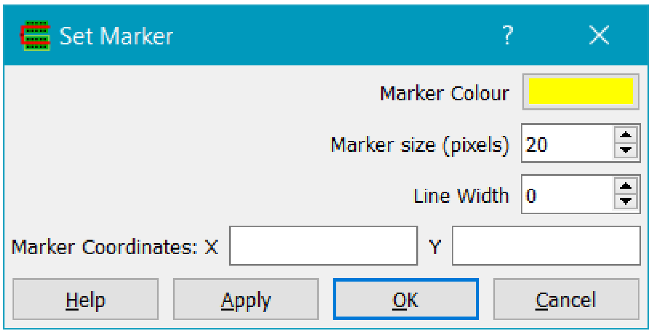


Figure 42 - Add Marker

The marker colour can be changed by the *Marker Colour* button, which displays the current marker colour. *Marker size* sets the size of the marker in pixels, so the marker size remains unchanged with zoom in/out operations. *Line Width* sets the linewidth; a linewidth of 0 or 1 is a single pixel line. The *Marker Coordinates* are the XY location of the marker in microns.

Markers are useful for setting temporary reference points in layout. Like rulers, they are not persistent i.e. they are not stored in any output format.

2.2.7 Tools->Clear Markers

The **Tools->Clear Markers** command clears all markers.

2.2.8 Tools->Netlist View

The **Tools->Netlist View** command opens the Netlist View window. This is a dock window to display a Spice/CDL netlist for netlist driven layout.

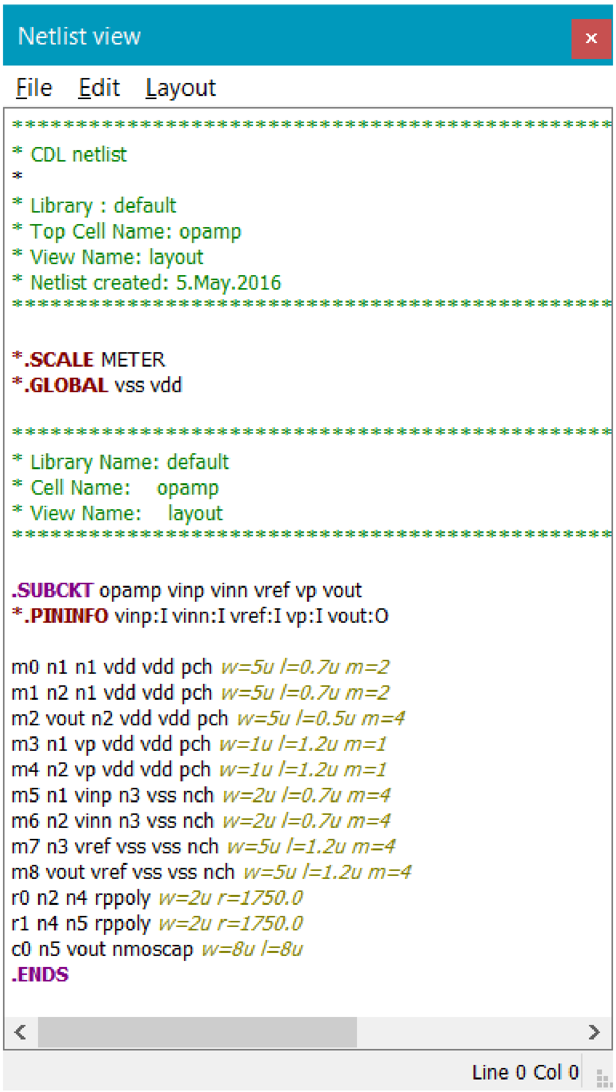


Figure 43 - Netlist View

The Netlist View has several menu items.

2.2.8.1 File->Open

The **File->Open** command loads a Spice or CDL file. The file is displayed in the Netlist View window with syntax highlighting.

2.2.8.2 File->Save
The **File->Save** command saves the current open file.

2.2.8.3 File Save As...
The **File->Save As...** Saves the current open file to a (new) file.

2.2.8.4 File->Close
The **File->Close** command closes the current open file.

2.2.8.5 Edit->Undo
The **Edit->Undo** command undoes an edit.

2.2.8.6 Edit->Redo
The **Edit->Redo** command redoes an undone edit.

2.2.8.7 Edit->Cut
The Edit->Cut command deletes the selected text.

2.2.8.8 Edit->Copy
The **Edit->Copy** command copies the selected text to the clipboard.

2.2.8.9 Edit->Paste
The **Edit->Paste** command pastes the text from the clipboard to the current cursor location.

2.2.8.10 Edit->Find...
The **Edit->Find...** command finds the specified text.

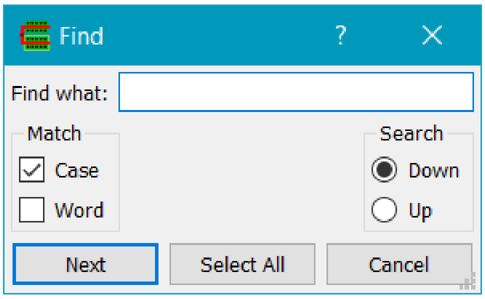


Figure 44 - Find Text

2.2.8.11 Edit->Goto Line...
The **Edit->Goto Line...** command moves the cursor to the specified line number.

2.2.8.12 Layout->Map Devices
The **Layout->Map Devices** command displays the Map dialog.

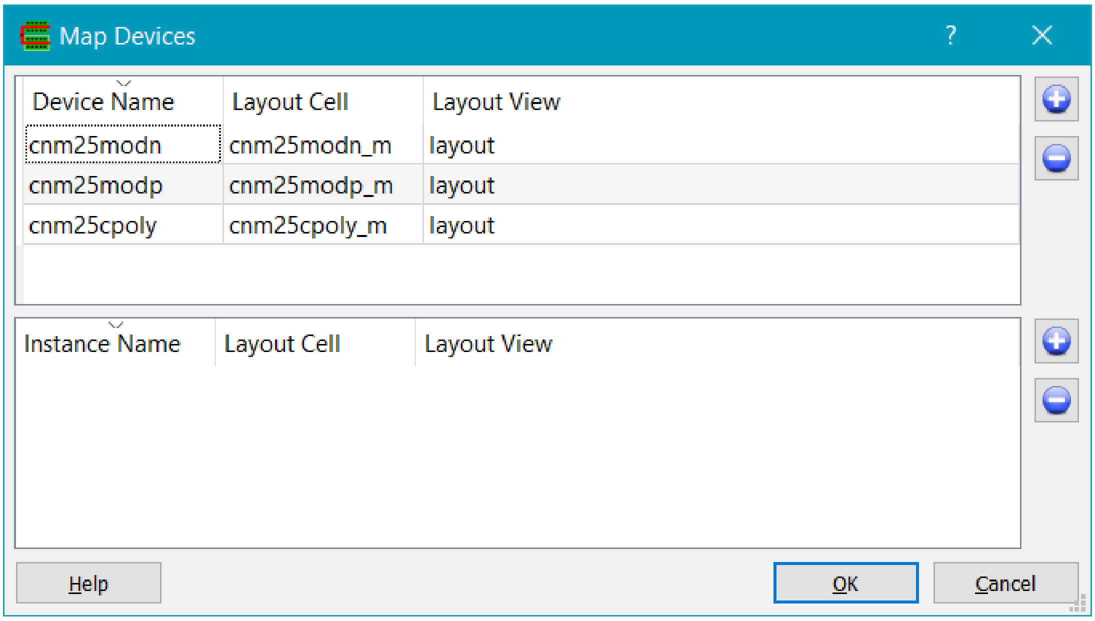


Figure 45 - Map Devices

The top table widget shows the device names found in the netlist in the first column. The second and third column contains the cellName and viewName of the cellView to map this device to in the layout. The lower level table widget shows the instance names found in the netlist in the first column. The second and third column contains the cellName and viewName of the cellView to map this instance to in the layout. Instance name mapping overrides device name mapping.

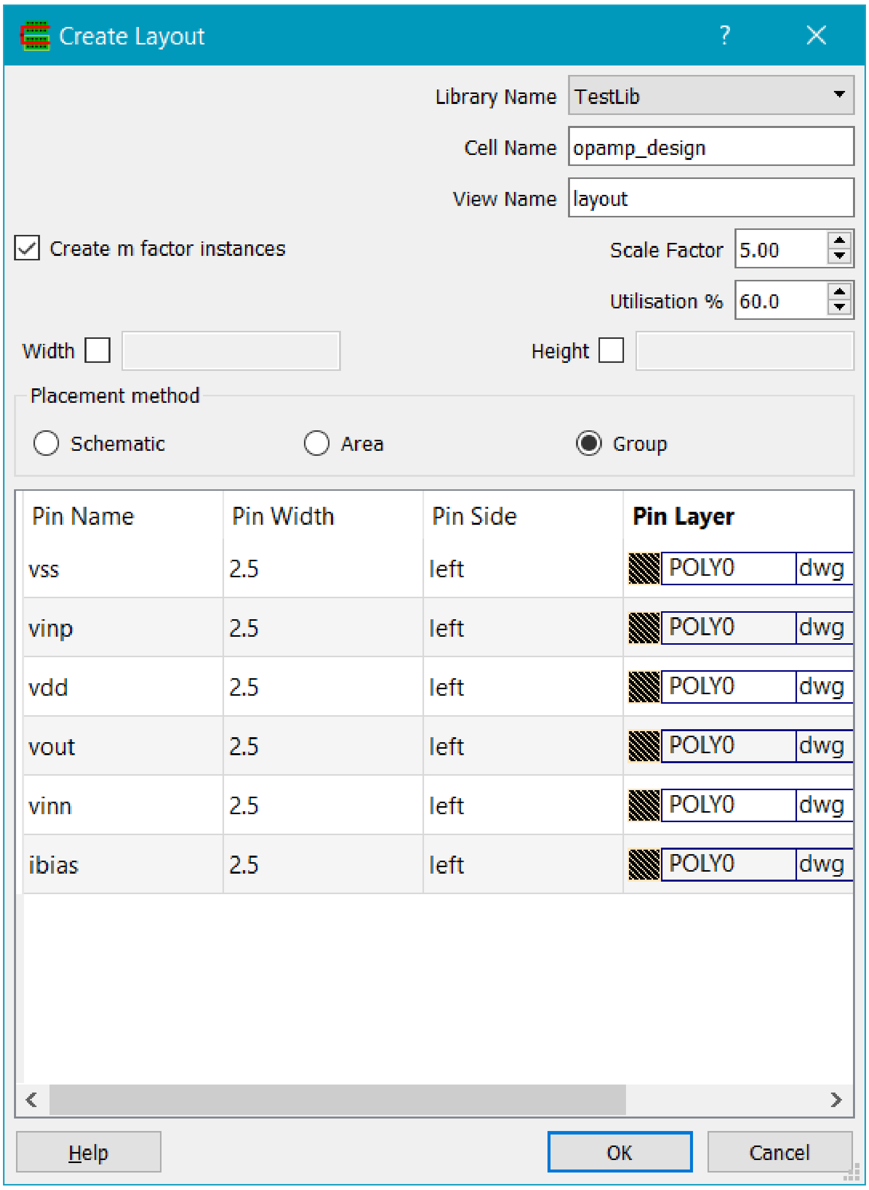
Device mapping defaults can be set in the techFile. For example,

MAP nch TO nmos layout ;

Maps the netlist device name to the layout cellView 'nmos13_multi layout'.

2.2.8.13 Layout->Gen Layout

The **Layout->Gen Layout** command displays the Create Layout dialog.



The target cellView is specified using the *Library Name* / *Cell Name* / *View Name* fields. If *Create m factor instances* is set, then if a netlist instance has a property 'm', multiple instances of the cell will be created in the layout based on the value of the property, and the m property is not passed to the layout PCell. If not checked, the m property is passed to the layout PCell, if the PCell is required to handle this itself.

Scale Factor is not used when generating layout from a netlist.

Utilisation is used to create the cell boundary layer in the resulting layout view. The area of all the layout instances is summed, and divided by 100/utilisation%. If *Width* is specified, the cell boundary will be rectangular with the specified width, and height will be computed from the area/width. If *Height* is specified, the cell boundary rectangle will have the specified height and the width will be computed from the area/height. If both *Width* and *Height* are specified, then the cell boundary rectangle will use the specified width and height.

Placement method can only be *Area* when generating layout from a netlist. *Area* arranges the layout cells by type (PMOS/NMOS/resistor/capacitor).

The pin field allows pin width, side and layer to be specified for each pin. Pins are placed abutting the cell boundary rectangle according to their side.

2.2.8.14 Layout->Clear Hilite

The **Layout->Clear Hilite** command clears existing netlist/layout hilites.

2.3 The Window Menu

The Window menu is used to manage open design windows. It is dynamically built and updated as windows are added or removed.

2.3.1 Window->Tab Style

The **Window->Tab Style** command changes the windowing mode to tab windows. Existing windows will be closed.

2.3.2 Window->MDI Style

The **Window->MDI Style** command changes the windowing mode to MDI (Multiple Document Interface) windows. Existing windows are closed.

2.3.3 Window->Close

The **Window->Close** command closes the current active window.

2.3.4 Window->Close All

The **Window->Close All** command closes all open windows.

2.3.5 Window->Tile

For MDI window mode, the **Window->Tile** command tiles the windows. Two open windows will be tiled horizontally; three will be tiled with one on the left, and two stacked vertically on the right etc.

2.3.6 Window->Cascade

For MDI Window mode, the **Window->Cascade** command arranges the window in a cascading fashion from the top left.

2.3.7 Window->Next

The **Window->Next** command changes the active window to the next open window in the window list.

2.3.8 Window->Previous

The **Window->Previous** command changes the active window to the previous open window in the window list.

2.4 The Help Menu

2.4.1 Help->Contents…

The **Help->Contents...** command displays the online help information.

2.4.2 Help->Index…

The **Help->Index…** command displays the online help index.

2.4.3 Help->About

The **Help->About** command displays about information.

2.5 Layout Menus

2.5.1 View->Fit

The **View->Fit** command zooms the display to fit the currently cellView's bounding box. The bounding box is scaled according to the **View->Pan/Zoom Options** Fit% value.

2.5.2 View->Fit+

The **View->Fit+** command zooms to 10% bigger than the displayed cell's bounding box.

2.5.3 View->Zoom In

The **View->Zoom** In command zooms in on the current cell by a factor of two. You can also zoom in by rotating the mouse wheel forward, or by pressing the right mouse button and dragging an area from lower left to upper right you want to zoom in to.

2.5.4 View->Zoom Out

The **View->Zoom Out** command zooms out on the current cell by a factor of two. You can also zoom out by rotating the mouse wheel backwards, or by pressing the right mouse button and dragging an area from upper right to lower left. The zoom factor is the current viewport size divided by the drag rectangle size.

2.5.5 View->Zoom Selected

The View->Zoom Selected command zooms to fit the window around the selected set.

2.5.6 View->Pan

The **View->Pan** command moves the centre of the display to the entered point. Note that panning can also be achieved by dragging with the middle mouse button held down; the pan is done in real time.

2.5.7 View->Pan to Point

The **View->Pan to Point** command displays a dialog box in which the X and Y coordinates to pan to can be entered. The display is then centred on these coordinates.

2.5.8 View->Redraw

The **View->Redraw** command redraws the screen. The display in Glade is double buffered, so redrawing consists of copying the back buffer to the front. This is fast, especially in the OpenGL version of Glade which is accelerated by hardware when using modern graphics cards. If however the display changes (e.g. by changing the layers that are visible in the LSW) then the back buffer is drawn again. However, this is still fast.

2.5.9 View->Ruler

The **View->Ruler** command draws a ruler.

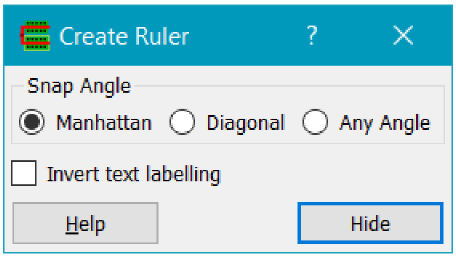


Figure 46 - Create Ruler

Manhattan, 45 degree and all angle rulers are supported. The popup dialog (toggle using F3 bindkey) allows the ruler snap angle to be changed while entering the ruler. Checking the Invert text labelling box puts the major/minor ticks on the opposite side of the ruler to normal. This can be useful when measuring edges of shapes with dense fills where the ruler text is not easily visible. To more accurately measure distances between shapes, you can turn gravity on, then the start and end points of the ruler can snap to the shape edges. You can zoom and pan while drawing rulers.

2.5.10 View->Delete Rulers

The **View->Delete** Rulers command deletes all rulers.

2.5.11 View->View level 0

The **View->View Level 0** command sets the display levels to 0. No contents of cells are visible.

2.5.12 View->View level 99

The **View->View Level 99** sets the display levels to 99. Cells up to 99 levels of hierarchy will have their contents displayed.

2.5.13 View->Previous View

The **View->Previous View** command sets the viewport to the last view before a pan/zoom/fit etc.

2.5.14 View->Cancel Redraw

The **View->Cancel Redraw** command cancels the current redraw. This can be useful on very large designs.

2.5.15 View->Display Options

The View->Display Options command displays the Display Preferences dialog.

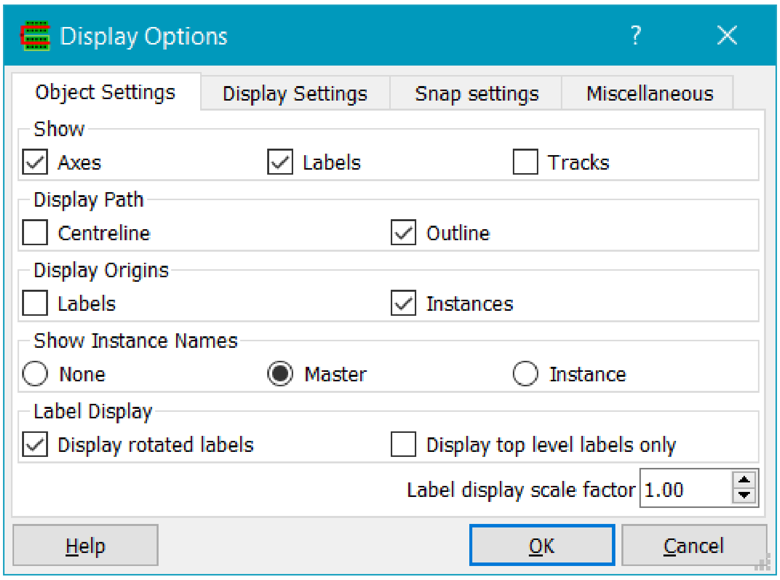


Figure 47 - Display Options (Object Settings)

Show Axes shows the X=0 and Y=0 axes.

Show Labels toggles the display of text labels. By default, text label display is turned off as in non-OpenGL display mode, drawing text labels can be slow if there are many labels.

Show Tracks displays the track grid for each layer for DEF based designs.

Display Path Centreline shows the centreline of path objects. *Display Path Outline* draws the outline of the path, based on its real width.

Display Origins - Labels shows the origin of text labels as a small cross. *Display Origins - Instances* shows the origin of instances as a small cross. Note that instance origins are only displayed if their bounding box is shown, i.e. they are at the display stop level.

Show Instance Names can be set to *None*, *Master* or *Instance*. With *Master*, the instance's master cell name is shown inside the instance bounding box. With *Instance*, the instance name is shown inside the instance bounding box.

Label Display allows finer control of text labels. *Display Rotated Labels* if checked displays text rotated as per its database orientation e.g. from GDS2. When unchecked, labels are displayed with no rotation (horizontally). *Display top level labels only* if checked will only display labels at the top of the cell hierarchy. Labels contained in lower levels will not be shown.

Label display scale factor will scale the displayed labels according to the scale factor set.

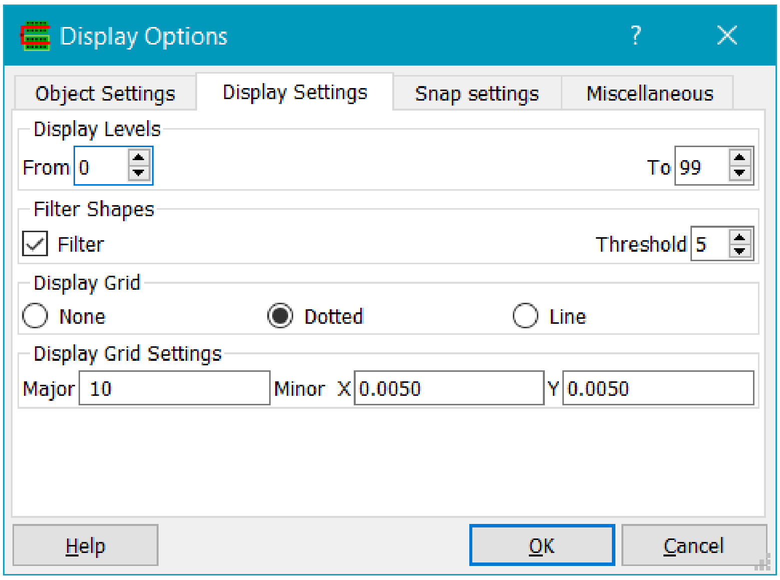


Figure 48 - Display Options (Display Settings)

Display Levels sets the levels of hierarchy that are displayed. A display level of 0, for example, means only display shapes and instance bounding boxes in the current cell. Although the view level 99 command turns on viewing of 99 levels of hierarchy, there's really no limit.

Filter Shapes controls filtering of objects to speed redraw. When zoomed out, it makes no sense drawing objects that are so small that they contribute nothing to the visible display. So with filtering enabled, objects with a size smaller than the threshold (in pixels) are not drawn. Turning filtering off is the same as setting the *Threshold* to 0. Note that path outlines are also subject to filtering; if the path width is less than the threshold then only the path centreline is drawn. Filtering can have a vast effect on redraw speed on large designs. The default filter level is 5 which is a good compromise of detail versus performance.

Display Grid controls the display grid which can be one of *None*, *Dotted* or *Line*. The display major grid is drawn using the LSW majgrid layer; the minor grid is drawn using the LSW mingrid layer.

Display Grid Settings. The *Minor X* and *Y* values set the dot or line spacing and are drawn using the mingrid layer. The *Major* grid spacing is the number of minor grids per major grid dot or line; it should be an integer, typically 5 or 10. The major grid is drawn using the majgrid layer.

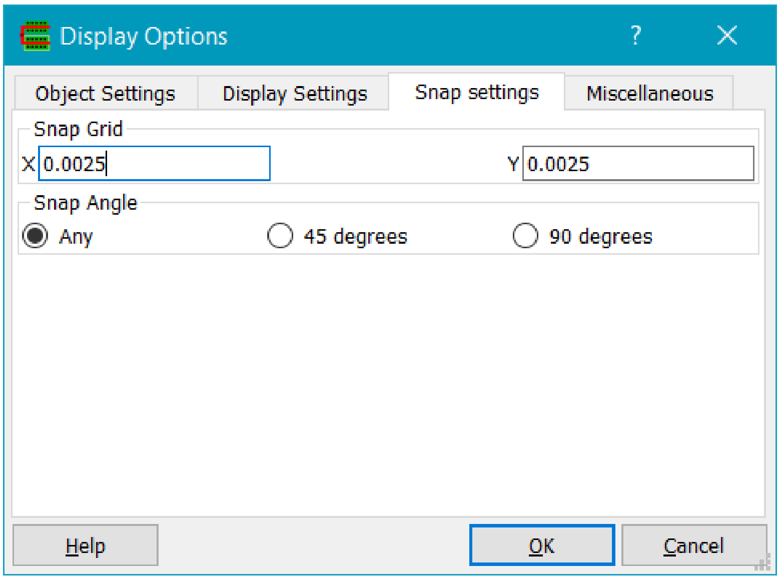


Figure 49 - Display Options (Snap Settings)

Snap Grid controls cursor snapping. The cursor is snapped to the values specified in X and Y. Snapping is modified by gravity; see the Selection Options dialog.

Snap Angle controls the angle that data can be entered for some shape creation and also for rulers. *Any* allows all angles; *45 degrees* and *90 degrees* snap accordingly.

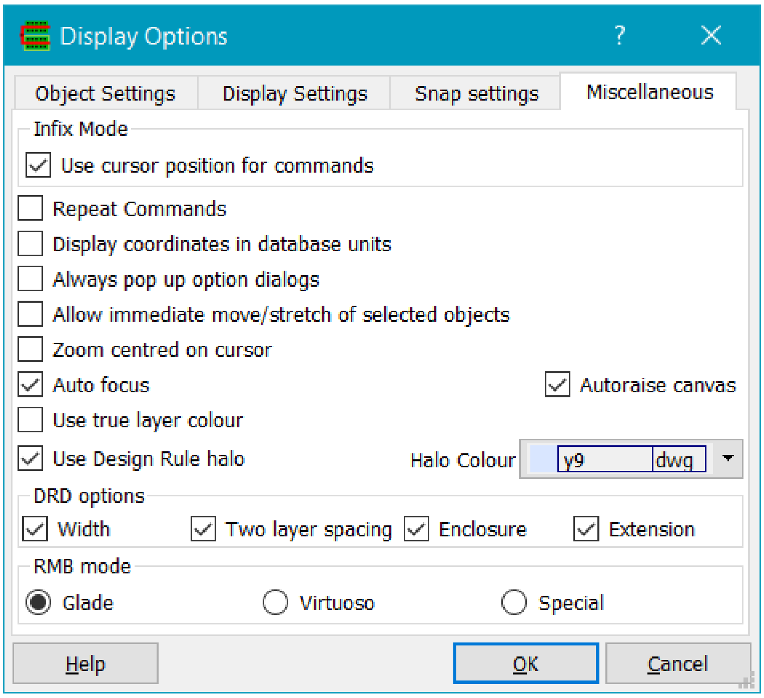


Figure 50 - Display Options (Miscellaneous)

Infix Mode is used for commands which can take the current mouse position rather than relying on the user to click on the first point of the command.

Repeat commands will keep repeating a command until ESC is pressed.

Display coordinates in database units shows coordinates in DB units, rather than microns. This can be useful when working with e.g. DEF files where the ascii coordinates in the file are in DB units.

Always popup option dialogs when checked will always show option dialogs for forms such as Create Path. These option dialogs can be shown and hidden by toggling the F3 key. If Always popup option dialogs is not checked, then the option forms will not be shown automatically (but can still be shown by pressing F3). This is useful when entering e.g. a lot of polygons.

Allow immediate move/stretch of selected objects will let selected objects be moved by the cursor without issuing a move/select command. The cursor changes according to the object. To use, select an object in full mode, or an edge/vertex in partial mode. The cursor will change to a 4-way arrow (for full mode select) or a 2-way arrow (for partial mode select). Then left click and drag to move or stretch the object. The object is deselected afterwards, so to repeat the command, select another object.

Keep immediate move selected will keep objects selected after an immediate move/stratch; otherwise all objects will be deselected.

Zoom centred on cursor sets the centre of the zoom to the cursor position; otherwise zoom in/out is centred on the viewport. This affects both zoom in/out and mouse wheel zoom.

Auto focus sets input focus to the canvas whenever the mouse moves over it. If this option is unchecked, then the user has to explicitly click on the main window in order to e.g. use bindkeys after any operation that transfers focus to another window. Some window managers may override this operation because they provide control of focus directly.

Auto raise raises the canvas window the mouse is over automatically. If this option is not set, the canvas must be explicitly clicked on to make it the active window for accelerator key input.

Use true layer colour will use the layer colour/fill for drawing during Create/Move/Copy/Stretch commands, rather than an outline drawn in the cursor layer colour..

Use Design Rule Halo will perform real time DRC and highlight shapes that would give rise to DRC MINSPACE violations. The violating shapes have a halo drawn round them which is the MINSPACE distance away from their edges/vertices. Checking is done for shapes (not yet instances) that are created/moved/copied/stretched, against existing shapes either at the top level of the hierarchy or also lower levels. Hierarchy check depth is controlled by the display stop level, i.e. checking will be performed to the depth of hierarchy that is displayed. *DRD options* controls which rules are checked; *Width* if checked turns on width checking of shapes, using the MINWIDTH rule for the layer*, Two layer spacing* turns on spacing checks between two different layers, using two layer MINSPACE rules. *Enclosure* and *Extension* enable checking minimum enclosure/minimum extension rules using the techFile MINENC / MINEXT rules.

Halo Colour is the layer colour used to draw the DRC violation halo.

RMB mode sets the operation of the right mouse button. It can be set to *Glade* mode (dragging the mouse down zooms in, dragging it up zooms out), *Virtuoso* mode (dragging the right mouse in any direction zooms in) or *Special* mode (dragging the right mouse down zooms in, dragging it up left zooms out, dragging it up right does a window fit).

2.5.16 View->Selection Options

The View->Selection Options command displays the Selection Options form.

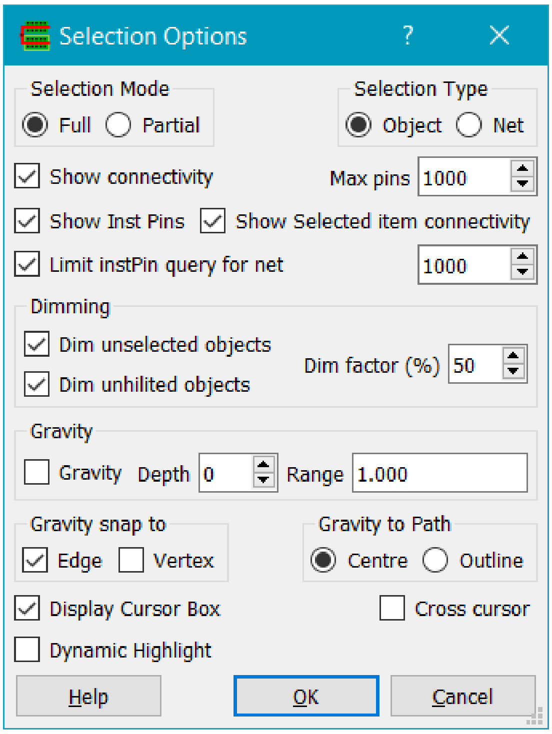


Figure 51 - Selection Options

Selection Mode is set to *Full* or *Partial*. Bindkey F4 toggles between these modes. *Full* mode selects the entire object. *Partial* mode selects an edge or vertex of an object. Use *Partial* mode to select edges or vertices for subsequent stretch commands.

Selection Type can be either *Object* mode or *Net* mode. In *Object* mode, only the object selected becomes part of the selected set. In *Net* mode, if a selected object is part of a net then all shapes that are part of that net are selected.

Show connectivity displays flightlines between instance pins that have connectivity. Max pins sets the limit to the number of pins that a connectivity flightline is drawn. Show Inst Pins shows instance pins as well as nets when *Show Connectivity* or *Show selected item connectivity* is checked. *Show selected item connectivity* shows connectivity flightlines when an object is selected that has net connectivity.

Limit instPin query for net will limit the number of instance pins shown in the Query Net dialog. This is useful when querying power/ground nets which may have hundreds of thousands of special net pins. Such a number of pins means the dialog is slow to build. Regular net pins are always shown.

Dim unselected objects will dim all unselected objects if any object(s) are selected. Unselected objects will be dimmed according to the *Dim factor* specified. This is useful when selecting an object e.g. a net by name in a large design and you want to display the selected object clearly. *Dim unhilited* objects will dim all unhighlighted objects, if highlighting is used e.g. by the Trace Net command.

Gravity when enabled will snap the cursor box to the nearest shape edge or vertex. The *Range* field determines how far an object edge can be from the current cursor position for gravity to take effect. Gravity works for all shapes and also the bounding boxes of instances at the current level of hierarchy only. *Depth* sets how far down the physical hierarchy shapes will be snapped to; for example with depth=0 only shapes in the current cell will be snapped to. Note that the grid snapping as set in the display options dialog overrides gravity snapping: in other words gravity snapping will snap to the nearest coordinate on grid if an object's edge is not on grid.

Gravity snap to sets whether snapping to edges or vertices is carried out when gravity is on.

Gravity to Path sets whether gravity snaps to path centrelines or edges.

Display Cursor Box shows a small square box in the LSW cursor colour, centered on the cursor, which is snapped to the current snap grid, or snapped to the nearest edge within gravity distance if gravity is on.

Cross Cursor when checked will display the cursor as a crosshair rather than as a box.

Dynamic Highlight highlights the object that will be selected if the left mouse button is clicked in Full selection mode. In Edge selection mode it highlights selectable edges, and in Vertex selection mode it highlights selectable vertices.

2.5.17 View->Pan/Zoom Options...

The View->Pan/Zoom Options... command displays the pan / zoom options dialog.

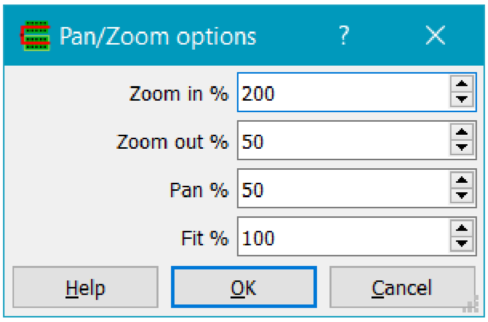


Figure 52 - Pan / Zoom Options

Zoom in % sets the percentage that a zoom in changes the current magnification. For example 200% zooms in by a factor of two. *Zoom out %* sets the percentage that a zoom out changes the current magnification. For example 50% zooms out by a factor of two. *Pan %* sets the percentage of screen width that is panned by the pan keys (left/right/up/down keys). For example 50% means shift the viewport half of the current viewport width. *Fit %* sets the percentage of screen width occupied by the current cellView's bounding box when a Fit command is issued. It can be in the range 10%-100%.

2.5.18 Edit Menus

Note that there are several function keys that can be used during editing. F1-F6 are hard coded and cannot be reassigned like bindkeys. On some platforms e.g. Mac, the function keys by default are assigned to special actions by the OS (for example raising/lowering the brightness of the display). It is possible to switch to normal Fn key mode operation (e.g. on the Mac by the **Settings->Keyboard** dialog).

- Escape key - aborts the current command.
- Return key - completes a Create Path, Create Polygon, Create MPP, Create Wire or Reshape command. The current cursor position is used as the last point. This is usually easier than double clicking to complete these commands.
- Backspace key - deletes the last vertex during a Create Path, Create Polygon, Create MPP, Create Wire or Reshape command.
- F1 key - opens the help browser.
- F2 key - toggles the Selection Options 'Gravity Mode' on/off.
- F3 key - toggles the command option dialogs.
- F4 key - toggles between Full and Partial selection modes. See Selection.
- F5 key - shows the Enter Coordinate dialog. For any command that normally takes a mouse click to enter a coordinate, F5 allows the user to specify the coordinates through the Enter Coordinates dialog box instead. For example, if you want to create a rectangle with coordinates (0.0, 0.0) (2.0, 3.0), click on the Create Rectangle icon, then press F5 and enter the first pair of coordinates and press OK. Then press F5 again and enter the second pair of coordinates.
- F6 key - toggles the Selection Options 'Display Connectivity' mode on/off.

Also note that double clicking the left mouse button will add a final path/mpp point, or add a final polygon point, or terminate the Reshape command.

2.5.19 Edit->Undo

The **Edit->Undo** command undoes the last edit made. Multiple undos can be carried out. Currently the only operations that can be undone are Delete, Move, Move Origin, Copy, Rotate, Stretch, Create, Merge, Chop, Flatten, Align, Reshape.

2.5.20 Edit->Redo

The **Edit->Redo** command redoes the last undo. Multiple redos can be carried out. Currently the only operations that can be redone are Delete, Move, Move Origin, Copy, Rotate, Stretch, Create, Merge, Chop, Flatten, Align, Reshape.

2.5.21 Edit->Yank

The **Edit->Yank** command copies the selected set into a yank buffer. The objects can then be pasted into another cellView (or even the same cellView) using the Paste command.

2.5.22 Edit->Paste

The **Edit->Paste** command pastes a copy of the items in the yank buffer into the current cellView.

2.5.23 Edit->Delete

The **Edit->Delete** command deletes the current selected set. Deletes can be undone.

2.5.24 Edit->Copy

The **Edit->Copy** command copies the current selected set. The F3 key will toggle the Copy options dialog.

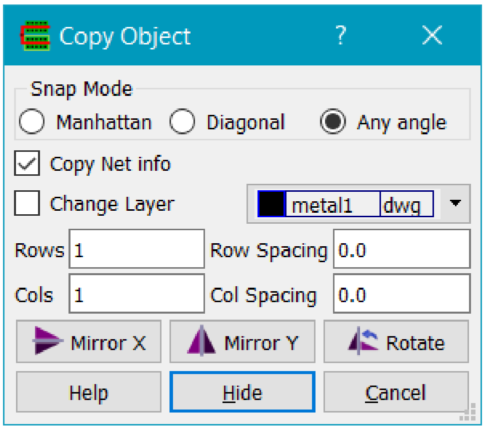


Figure 53 - Copy

Snap Mode can be set to *Manhattan*, *Diagonal* or *Any Angle*. *Copy Net info* if checked will copy a shape's net connectivity. If a shape is being copied, *Change Layer* will allow the layer of the new shape to be changed to the one selected by the layer chooser. If *Rows* or *Cols* is set to a number greater than 1, an array of objects will be copied with the spacing set by *Row Spacing* and *Col Spacing*. If *Mirror X* is pressed (or the 'x' key) during a copy, the object is mirrored in the X axis. If *Mirror Y* is pressed (or the 'y' key) during a copy, the object is mirrored about the Y axis. If *Rotate* is pressed (or the 'r' key) the object is rotated 90 degrees anticlockwise.

If infix mode is on, the current cursor position is used for the reference coordinate. Else you will be prompted to enter the reference coordinate. During a copy operation, the object(s) are shown as outlines and delta coordinates (dX/dY) from the initial position are shown on the status bar.

2.5.25 Edit->Move

The Edit->Move command moves the current selected set. The F3 key will toggle the Move options dialog.

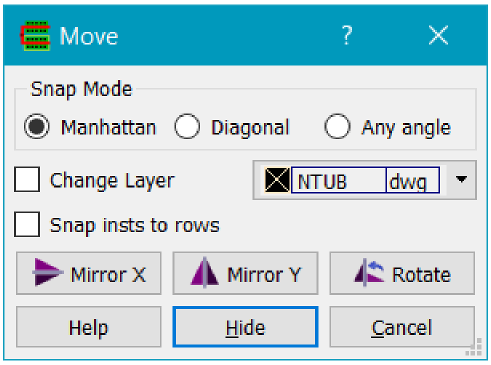


Figure 54 - Move

Snap Mode can be set to *Manhattan*, *Diagonal* or *Any Angle*. If a shape is being moved, *Change Layer* will allow its layer to be changed to the one selected by the layer chooser. If moving instances, *Snap insts to rows* will snap instances to row objects if they exist. If *Mirror X* is pressed (or the 'x' key) during a copy, the object is mirrored in the X axis. If *Mirror Y* is pressed (or the 'y' key) during a copy, the object is mirrored about the Y axis. If *Rotate* is pressed (or the 'r' key) the object is rotated 90 degrees anticlockwise.

If infix mode is on, the current cursor position is used for the reference coordinate. Else you will be prompted to enter the reference coordinate. During a move operation, the object(s) are shown as outlines and delta coordinates (dX/dY) from the initial position are shown on the status bar.

2.5.26 Edit->Move By...

The **Edit->Move By...** command moves the current selected set by the distance specified in the Move By dialog.

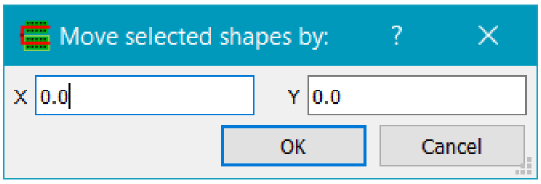


Figure 55 - Move By

2.5.27 Edit->Stretch

The **Edit->Stretch** command stretches the current selected edge or vertex. The F3 key will toggle the Stretch options dialog.

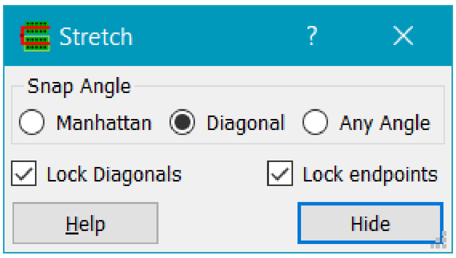


Figure 56 - Stretch

Snap Angle can be set to *Manhattan*, *Diagonal* or *Any Angle*. If objects as well as edges or vertices are selected, they are moved by the stretch distance. If *Lock Diagonals* is checked, diagonal edges will be locked to 45 degrees, otherwise moving an edge adjacent to a diagonal may make the diagonal edge become any angle. *Lock Diagonals* has no effect when stretching vertices. If *Lock endpoints* is checked, then stretching a path segment at the beginning or ending of a path will split the path at the start or end vertex, keeping the start/end vertex fixed and stretching the other part of the split segment.

If infix mode is on, the current cursor position is used for the reference coordinate. Else you will be prompted to enter the reference coordinate. During a stretch operation, the object edge(s)/vertex(vertices) are shown as outlines and delta coordinates (dX/dY) from the initial position are shown on the status bar.

2.5.28 Edit->Reshape

The **Edit->Reshape** command reshapes the currently selected edge of a polygon or path.

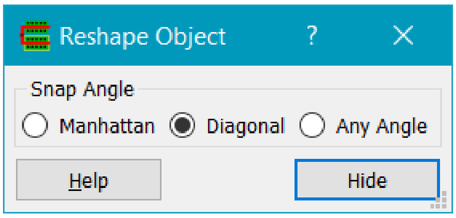


Figure 57 - Reshape

Snap Angle can be set to *Manhattan*, *Diagonal* or *Any Angle*.

To reshape an object, first select an edge of a polygon or the centreline of a path (in partial selection mode). Then enter vertices you wish to add to the edge. The original start and end points of the edge will be unchanged. Vertices can be added according to the *Snap Angle*.

Double click or press return to complete reshaping the edge. Pressing backspace will back up one vertex. Although Reshape only works with paths and polygons, you can convert any object e.g. a rectangle to a polygon using the Edit->Convert to Polygon command.

2.5.29 Edit->Round Corners

The **Edit->Round Corners** command rounds the corners of a rectangle or polygons.

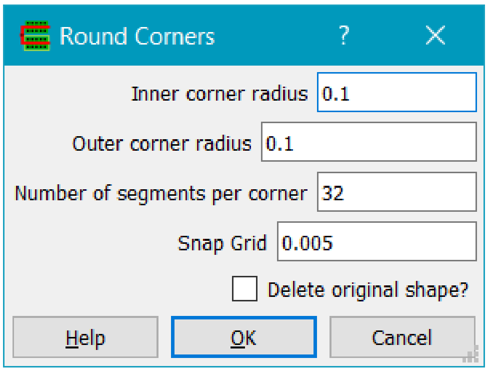


Figure 58 - Round Corners

You must first select a shape to round. *Inner Corner Radius* sets the radius of curvature in microns of inner corners; *Outer Corner Radius* sets the radius of outer corners. *Number of segments per corner* sets the precision of the generated curve which is made up of segments (straight lines). *Snap Grid* sets the manufacturing snap grid to avoid off-grid vertices; if no snapping is required set the value to the user database resolution (usually 0.001um). If *Delete Original Shape?* is checked (the default), the original shape is deleted.

2.5.30 Edit->Add Vertex

The **Edit->Add Vertex** command adds a vertex to a selected path or polygon at the point given by the cursor. The vertex that has been added is selected, so it can be moved using a Stretch command.

2.5.31 Edit->Rotate

The **Edit->Rotate** command rotates the current selected set about a point, which the user is prompted to enter.

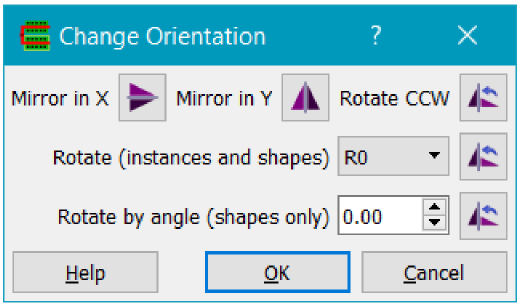


Figure 59 - Rotate

Mirror in X mirrors the objects about the X axis, *Mirror in Y* mirrors the objects about the Y axis. *Rotate CCW* rotates the objects counter clockwise. *Rotate (instances and shapes)* rotates instances according to the transform selected. *Rotate by angle* rotates shapes by any angle from -360.0 to +360.0 degrees; a positive angle corresponds to a clockwise rotation. Only shapes can be rotated by any angle; rectangles and squares get converted to polygons and are then rotated, while paths and polygons are maintained and their vertices are rotated.

Instance placement orientation can be changed by querying the instance's properties and changing the orientation.

2.5.32 Edit->Move Origin

The **Edit->Move Origin** command moves the origin of the current cell. Click on the point that you want to make the new origin, and all object coordinates in the current cell will be changed to make this point (0, 0).

2.5.33 Edit->Convert to Polygon

The **Edit->Convert to Polygon** command converts selected shapes into polygons. This command is useful in conjunction with the Edit->Reshape command above.

2.5.34 Edit->Boolean Operations...

The **Edit->Boolean Operations...** command performs boolean operations on layers.

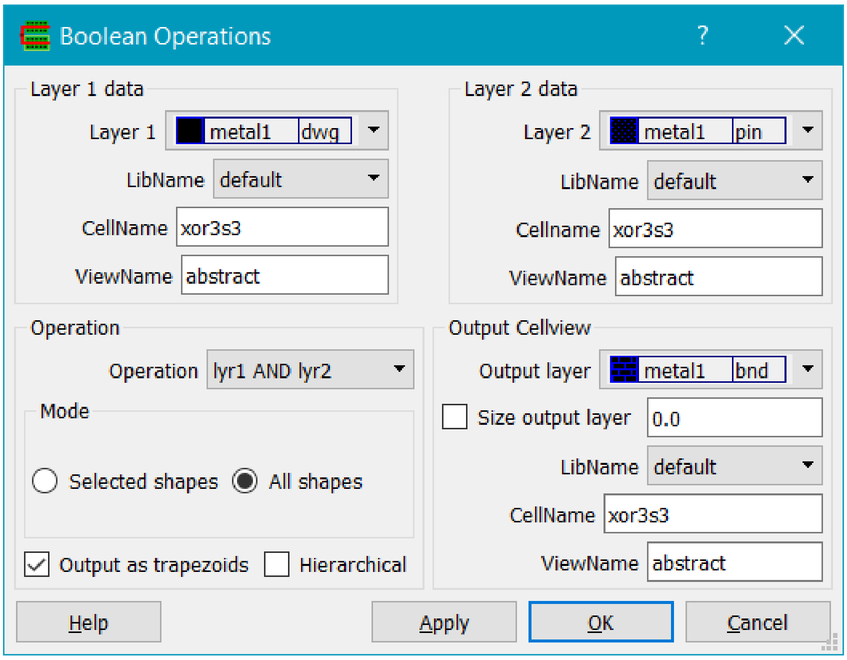


Figure 60 - Boolean Operations

Layer1 and *Layer2* are input layers, and *Output Layer* is the output layer. By default layer data is processed from the current open cellView, however it is possible to set the library/cell/view names of the cells containing Layer1 data and Layer2 data. Operations that can be performed are two layer AND, two layer OR, single layer OR (merge), single layer NOT, two layer NOT, two layer XOR, sizing and up/down sizing (first size up by a given amount, then size down by the same amount - useful for removing small gaps or notches) and selection (select all shapes on a layer that touch shapes on another layer). *Mode* allows either *Selected Shapes* on Layer1 and, if used, Layer2 to be processed only, else *All Shapes* for the layer(s) will be processed. If *Output data as trapezoids* is checked, the resulting layer is converted into trapezoids rather than complex polygons. If *Hierarchical* is checked, the design hierarchy is flattened and all shapes on the layer(s) are processed; else just shapes in the top level cellView are processed. The *Output CellView* is the destination for the generated data. By default this is set to the current cellView, but can be any cellView; if the cellView does not exist it will be created. If Size Output Layer is checked, the output layer can be also sized by an amount (except for the operation Size lyr1).

2.5.35 Edit->Tiled Boolean Operations...

The **Edit->Tile Boolean Operations...** command performs boolean operations on layers. It is useful when the data is too large to process with Edit->Boolean Operations... as it uses a tiling algorithm to process the data tile by tile.

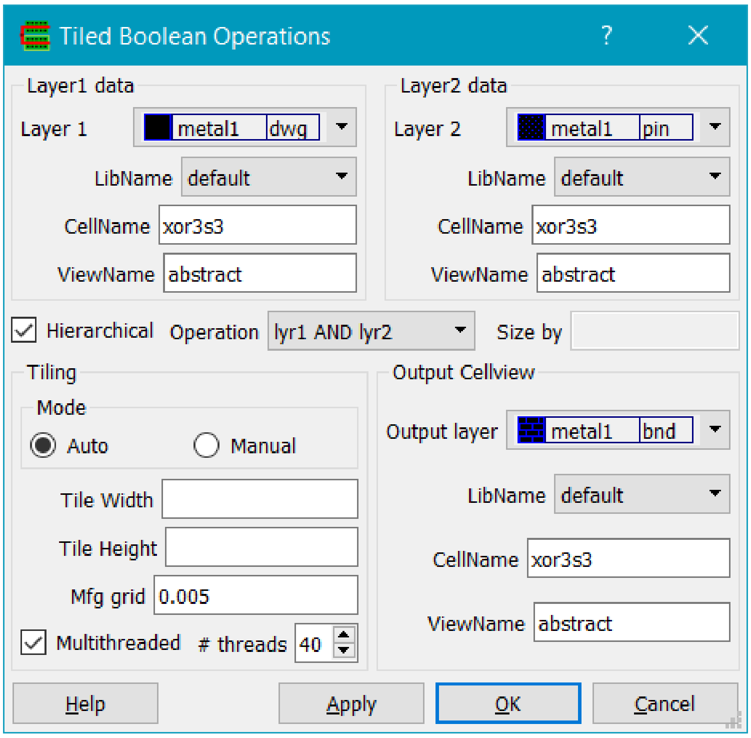


Figure 61 - Tiled Boolean Operations

Layer1 data and *Layer2 data* specify the input layer sources. The cellView for each layer defaults to the current displayed cellView, but can be changed e.g. to compare two cells using an XOR operation on the same layer, for example. *Operation* specifies the boolean operation to be performed. Currently only merge (single layer OR), OR, AND, ANDNOT, NOT, XOR and SIZE operations are supported. The *Output CellView* specifies the cellView that output shapes will be created on, according to the output layer specified.

If *Hierarchical* is checked, the cellView's data is flattened before the operation.

Tile size can be determined automatically if *Tiling Mode* is set to *Auto*. Else the tile width and height can be specified if *Tiling Mode* is set to *Manual*. The larger the tile size, the more physical memory will be used. For large designs with many levels of hierarchy, computing the best tile size can take a long time - so in this case manually setting the tile sizes is preferable. Typically a starting point of 500-1000um should be acceptable. Setting smaller tile sizes will use less memory, but may run longer.

Multithreaded specifies that the tiles are split and run on a multiple number of threads, which may speed up overall runtime at the expense of somewhat more memory usage. *# threads* defaults to the maximum number of threads that are feasible on your system. For example, a 4 core hyperthreaded Intel i7 processor will support 8 threads. Speed improvement is not linear with the number of threads due to IO and memory bottlenecks. Typically with 4 cores, about 3.5x speed improvement is gained.

2.5.36 Edit->Merge Selected

The **Edit->Merge Selected** command merges all selected shapes into polygons. Layers are preserved, i.e. only shapes on the same layer are merged. If you want to merge shapes on different layers, use the Edit->Booleans command with operation Layer1 OR layer2.

2.5.37 Edit->Chop

The **Edit->Chop** command chops a rectangle out of a selected shape.

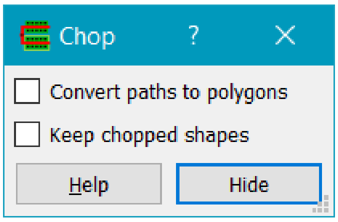


Figure 62 - Chop

First, select a shape. Then invoke the chop command and draw a chop rectangle. The shape will have the rectangle chopped out of it. If *Convert paths to polygons* is checked, paths will be converted to polygons before the chop takes place. Otherwise paths will be maintained and will be cut. If *Keep chopped shapes* is checked, the chop shapes from polygons are not deleted.

2.5.38 Edit->Align

The Edit->Align command aligns objects and optionally spaces them in the direction perpendicular to the alignment edge.

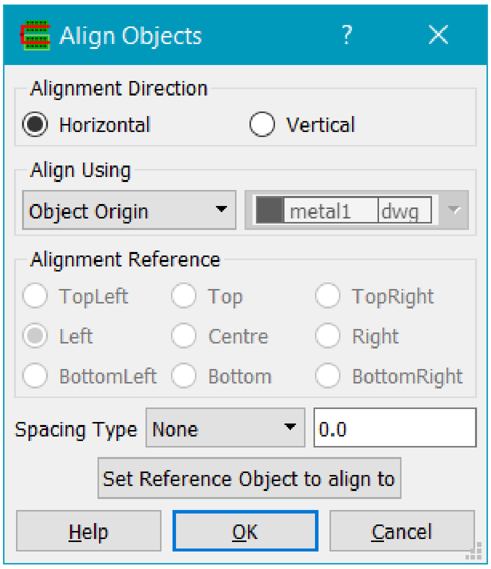


Figure 63 - Align

Alignment Direction is used when Align Using is set to Object Origin and can be horizontal or vertical. Horizontal will align objects horizontally e.g. by their left edges, and Vertical will align them vertically e.g. by their bottom edges. Alignment can be by Object Origin, Object bBox or Layer bBox.

Object Origin aligns according to the origin of an instance or array or the lower left of the bounding box of shapes. Figure 64 shows object alignment using Horizontal *Alignment Direction*.

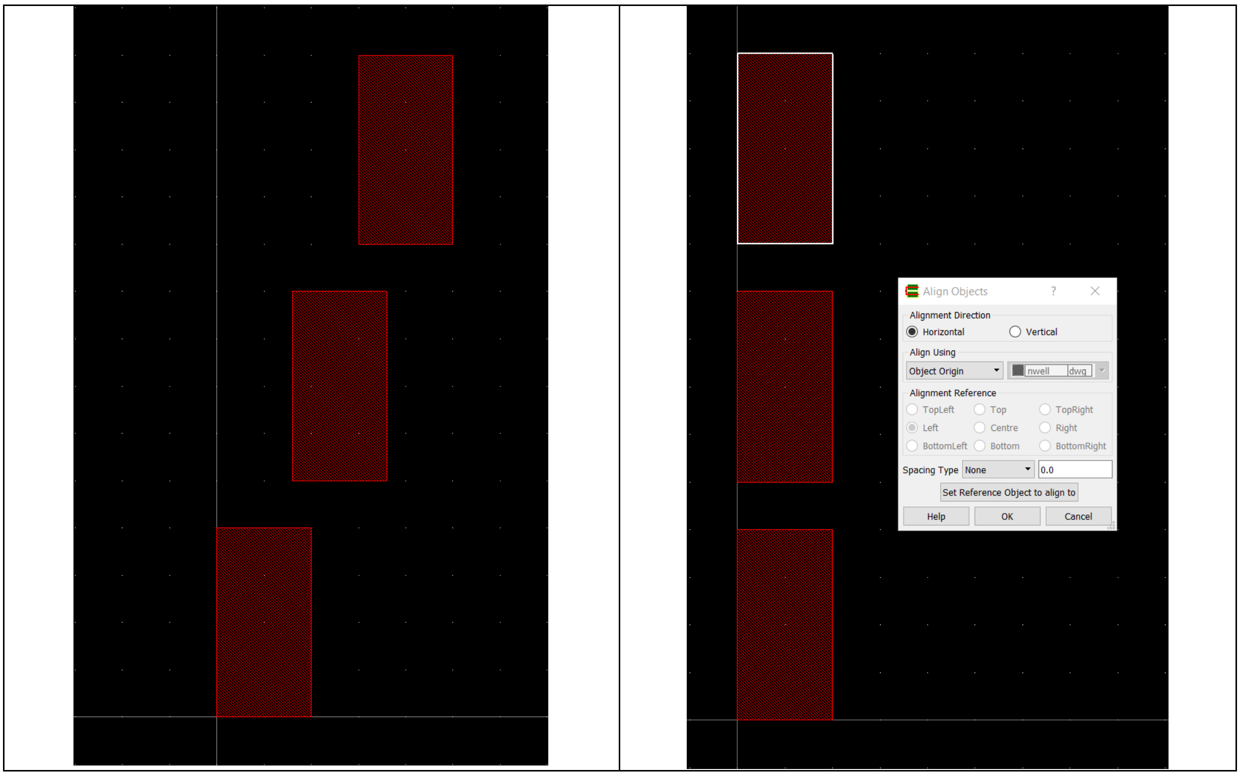


Figure 64 - Before and after horizontal align

Object bBox aligns according to the two object's bounding boxes, and ignores the *Alignment Direction*. When selected, Object bBox alignment will enable the *Alignment Reference* choices over which edge of the bounding box will be aligned. For example selecting *Top* will align the top edges of the objects; *Centre* will align the objects so they are centred on each other, and so on.

Layer bBox alignment is only applicable when aligning instances, and will align them according to a common layer in the instance, as given by the layer chooser.

Spacing Type controls object spacing during alignment. Any spacing is applied perpendicular to the *Alignment Direction*. It can be set to either *None*, *Space* or *Pitch*.

None sets object spacing to zero. For example if objects are aligned according to their origin horizontally, their position in the Y direction remains unchanged.

Space aligns objects so that the space between the reference object edge and the first object edge, the first object edge and the second object edge and so on is equal to the space value set. For Object bBox / Layer bBox alignment, spacing is only valid for left/right/top/bottom alignment reference. Figure 65 shows the effect of aligning objects using Horizontal alignment, with Spacing Type set to Spacing, and value 0.1um.

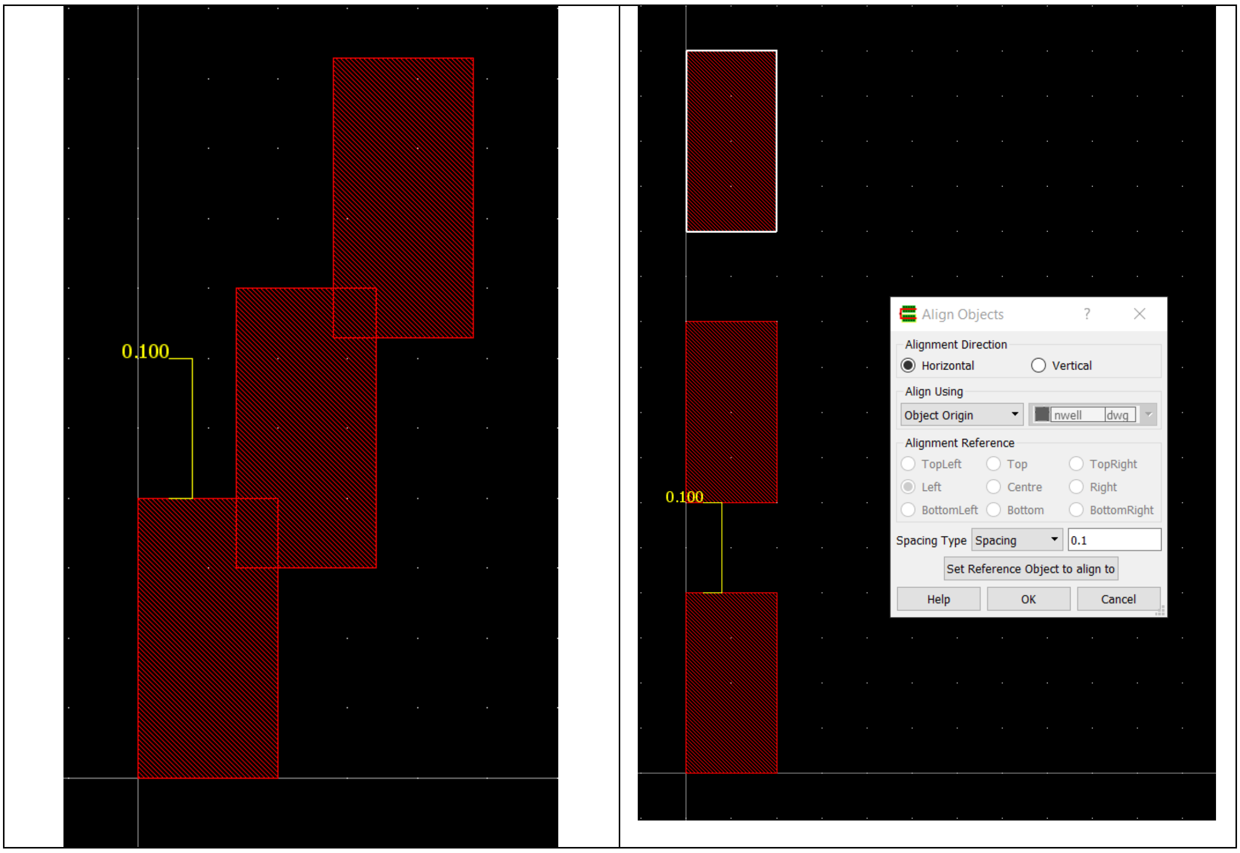


Figure 65 - Alignment with spacing

Pitch aligns objects so that the pitch between the reference object and the first object, the first object and the second object and so on is equal to the pitch value set. For Object bBox / Layer bBox alignment, pitch is only valid for left/right/top/bottom alignment reference.

To perform alignment, left click on *Set Reference Object to align to*. Then click on the objects you wish to align; OK or Cancel the dialog to finish an alignment sequence. Click again on Set Reference Object to start a new alignment sequence.

Note: Be sure to Cancel (ESC) or OK the Align command before carrying out a new command.

2.5.39 Edit-> Scale

The **Edit->Scale** command scales all objects in the current cellView by a simple linear scale factor.

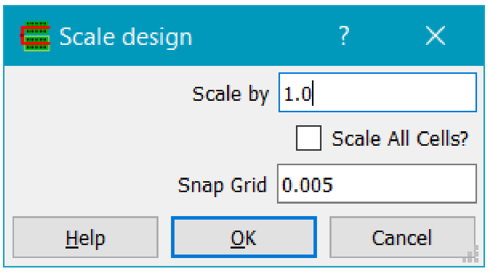


Figure 66 - Scale

Scale By sets the scaling factor; all coordinates are multiplied by this factor. If *Scale all cells?* is checked, all cells in the library will be scaled. Coordinates are snapped to the *Snap Grid*.

2.5.40 Edit->Bias

The **Edit->Bias** command biases shapes.

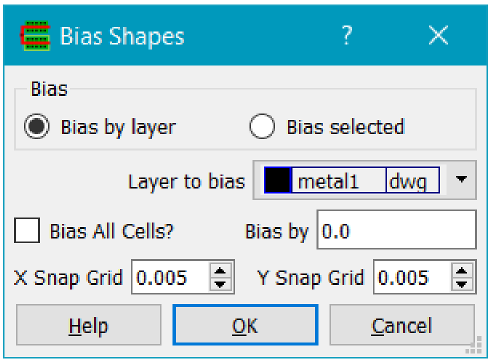


Figure 67 – Bias

Bias can be either *Bias by Layer*, which biases all shapes on the specified *Layer to bias*, or *Bias Selected* (selected shapes can be on any layer). *Bias by* sets the bias. A positive bias causes shapes to grow in size; a negative bias causes them to shrink.

If *Bias all cells?* is checked, all cells in the library will have the bias applied. Coordinates are snapped to the *X Snap Grid* and *Y Snap Grid*. Note that polygons with collinear or coincident points will not be biased correctly and a warning will be given.

2.5.41 Edit->Set Net

The **Edit->Set Net** command sets a selected shape's net.

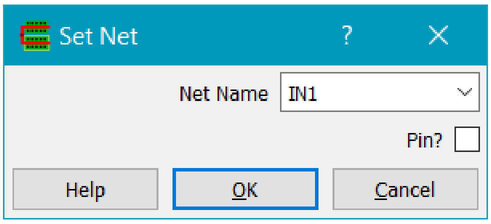


Figure 68 - Set Net

The *Net Name* combo box is filled with any existing net names in the cellView, or you can type in a net name to create that net. If *Set As Pin?* is checked, the shape(s) will become pin shapes.

2.5.42 Edit->Create Pins From Labels

The **Edit->Create Pins From Labels** command creates pin shapes from text labels.

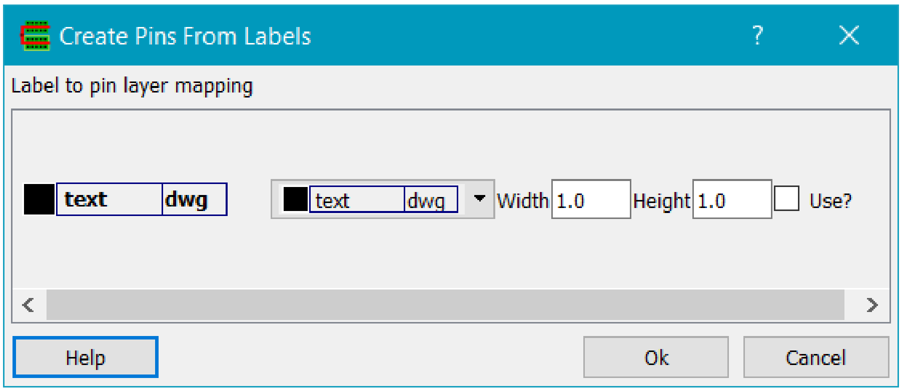


Figure 69 - Create Pins From Labels

All valid label layers are shown in the dialog. The first layer box shows the label layer. The second layer chooser allows control of the layer that pins will be generated on. Pins are created as rectangles centred on the label origin with the specified *Width* and *Height*. Pins are only created if the *Use?* option is checked.

2.5.43 Edit->Hierarchy->Ascend

The **Edit->Hierarchy->Ascend** command ascends one level of hierarchy, assuming you have previously descended into a cellView's hierarchy.

2.5.44 Edit->Hierarchy->Descend

The Edit->Hierarchy->Descend command descends into the selected instanceor tries to find an instance under the cursor to descend into if nothing is selected.

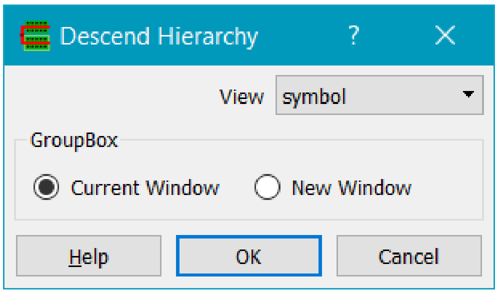


Figure 70 - Hierarchy Descend

View is the view of the instance to descend into; for example a schematic instance may have both a symbol view and a schematic (lower level of hierarchy) view. *Open In* controls the window used to display the cellView; *Current Window* uses the existing window, and the **Edit->Hierarchy->Ascend** command can be used to return to the previous cellView in the hierarchy. *New Window* opens a new window for the cellView, leaving the previous cellView window open.

2.5.45 Edit->Hierarchy->Create

The **Edit->Hierarchy->Create** command creates a new cell from the selected set.

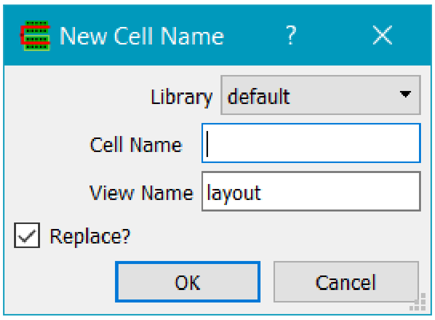


Figure 71 - Create Hierarchy

Library, *Cell Name* and *View Name* specify the new cellView to be created. By default the selected objects are deleted from the current cellView, and an instance of the new cellView is placed in the current cellView to replace them. If *Replace?* is checked, then the selected objects are deleted.

2.5.46 Edit->Hierarchy->Flatten

The **Edit->Hierarchy->Flatten** command flattens the current selected instances into the current cellView.

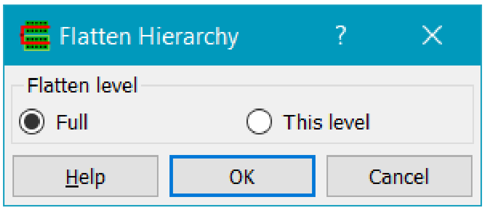


Figure 72 - Flatten Hierarchy

Flatten level controls the flattening process; with *Full* checked the complete hierarchy from the current level down to leaf cells is flattened. If *This Level* is checked, then only instances at the current level of hierarchy are flattened; lower levels of the hierarchy are preserved.

2.5.47 Edit->Edit in place->Edit in place

The **Edit->Edit In Place->Edit In Place** command allows editing a cell in place.

First select an instance that you want to edit. The Edit in place command will cause all subsequent selection and editing will be done in the master cell for that instance, but with the original top level cell displayed. The edit in place cell will be shown with layers of normal intensity, whereas all other shapes (of non-editable cells) will be shown dimmed, according to the dimming value set in the Selection Options dialog.

Edit in place is hierarchical, i.e. you can choose to edit in place a cell within another cell you are currently editing in place.

2.5.48 Edit->Edit in place->Return

The **Edit->Edit In Place->Return** command returns to the parent cell of the current edit in place cell.

2.5.49 Edit->Select->Inst by name

The Edit->Select->Inst By Name command Displays allows selection of instances based on their instance name.

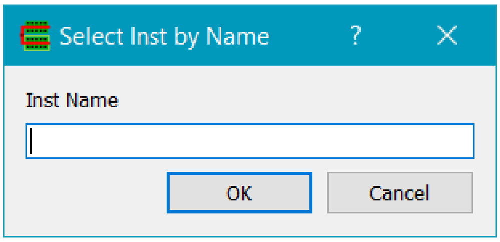


Figure 73 - Select Inst By Name

2.5.50 Edit->Select->Net by Name

The **Edit->Select->Net By Name** command allows selection of nets based on their name.

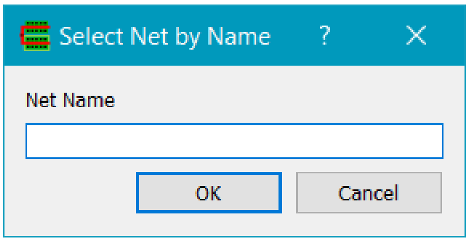


Figure 74 - Select Net By Name

2.5.51 Edit->Select->Select All

The **Edit->Select->Select All** command selects all currently selectable objects.

2.5.52 Edit->Select->Deselect All

The Edit->Select->Deselect All command deselects all the selected set.

2.5.53 Edit->Find

The **Edit->Find** command displays the Search dialog.

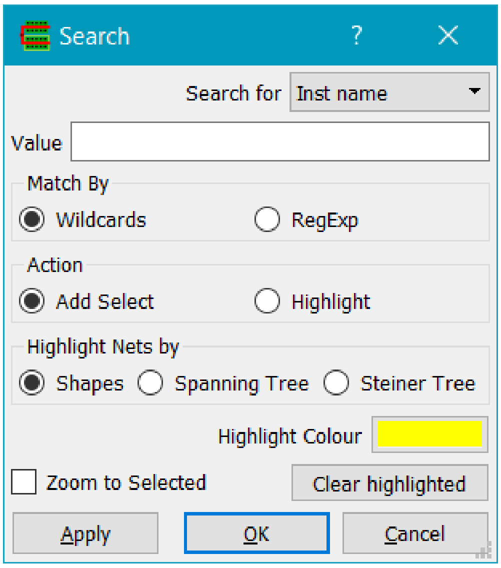


Figure 75 - Find

Find searches for instances by name, instances by master name (cell name), nets by name or text labels by name. Names can be matched by *Wildcard* (e.g. VDD* matches VDD1, VDD2, VDD) or by *RegExp* (regular expressions). Objects that match the selection criteria can be added to the selected set or highlighted. In the case of highlighted nets, they can be displayed either as the actual net shapes highlighted, or by a *Spanning Tree* between the instance pins of the instances the net connects to, or as *a Steiner tree*. This is useful, for example, in highlighting the connectivity of unrouted nets; the spanning tree is a good approximation to the path an autorouter will take; the Steiner tree is even better although can be slow on nets with many pins. The colour can be chosen using the *Highlight Colour* button. Optionally the display can *Zoom to Selected* object(s) and it is possible to clear all highlighted objects using the *Clear highlighted* button.

2.5.54 Edit->Properties->Query Object

The **Edit->Properties->Query Object** command queries the selected object.

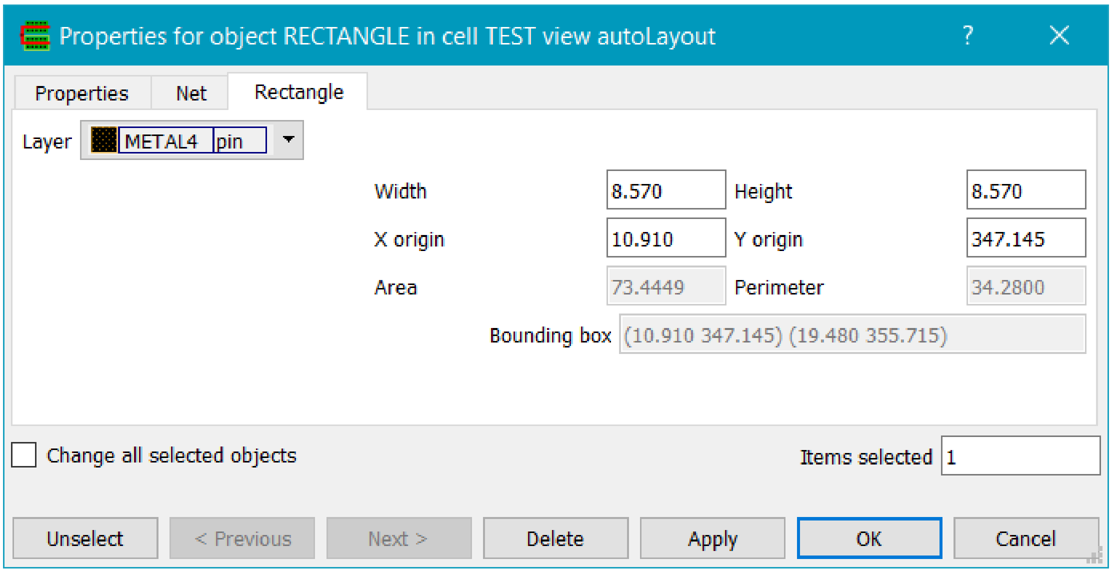


Figure 76 - Query Object

With nothing selected, the current cell's properties are queried. Otherwise you may query any selected object's properties and attributes, and cycle through the selected set using the *Previous* and *Next* buttons. You can delete a queried object using the *Delete* button. You can remove an object from the selected set with the *Unselect* button. If multiple objects are selected, *Change all selected objects* allows their common attributes to be changed. For example, if shapes are selected then the layer may be changed for all shapes. If the object has connectivity, a Net properties tab is added to the dialog. All objects may have user or system-defined properties which can be manipulated on the Properties tab page.

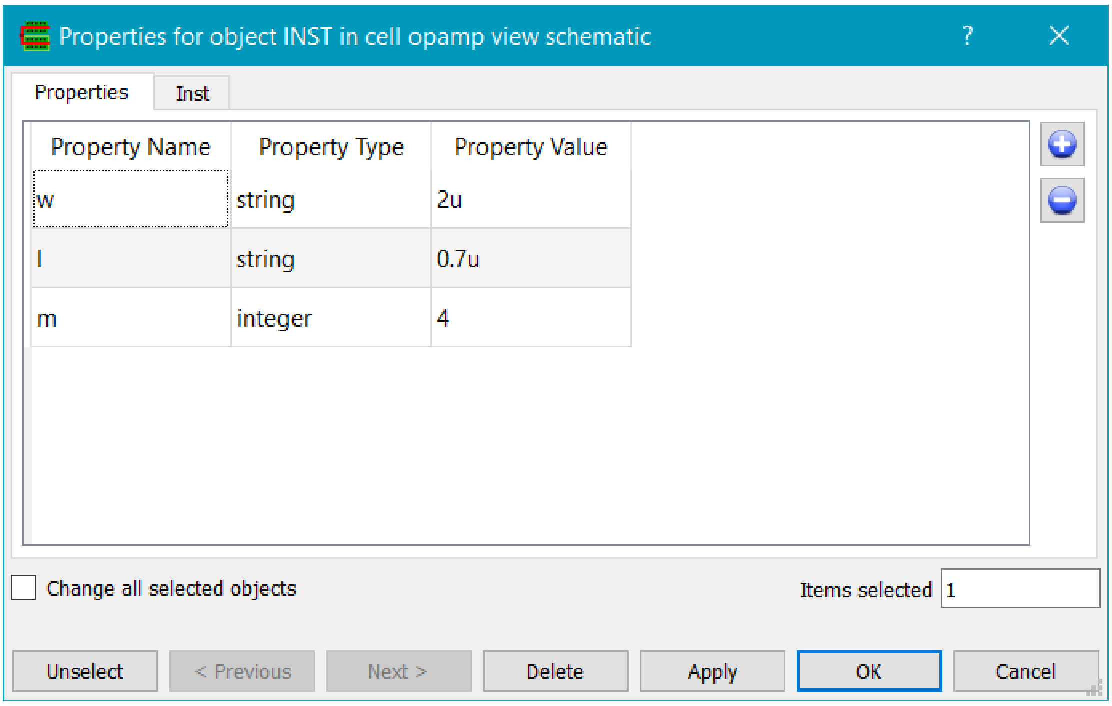


Figure 77 - Query Object Properties

Properties can be added as string, float, integer, boolean, list or orient. Click on the property name or value to change the text, or click on the type and select the type in the combo box that will appear. Click on the '+' button to add a (initially blank) property entry, or select a property and click on the '-' button to delete the property.

There is currently no undo capability if you delete a property.

2.5.55 Edit->Properties->Query CellView

The **Edit->Properties->Query CellView** command displays the Query dialog for the current cellView.

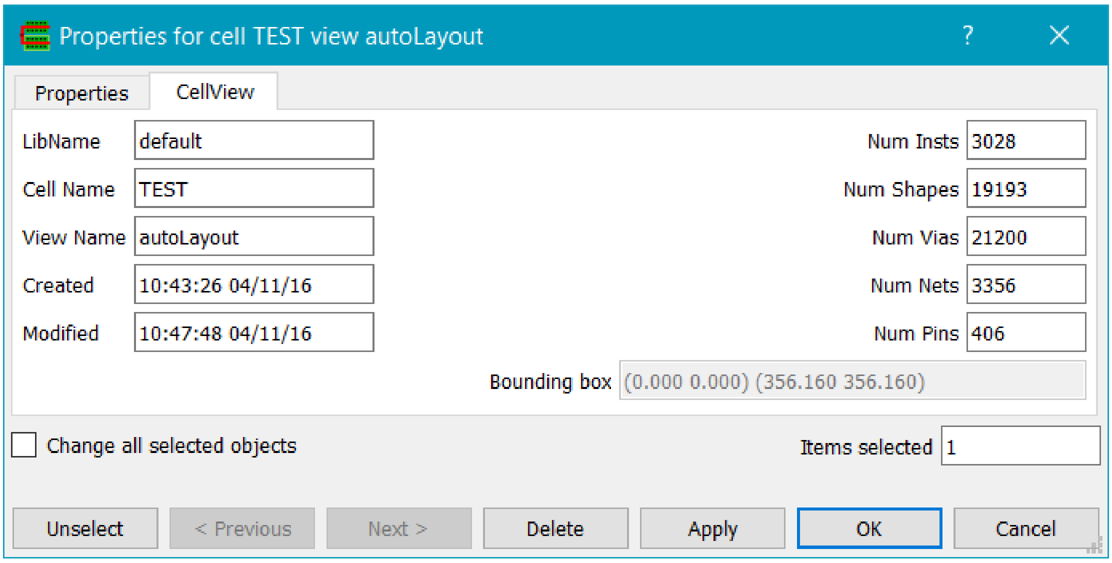


Figure 78 - Query CellView

2.5.56 Edit->Bindkeys

The Edit->Bindkeys command displays the Edit Bindkeys dialog.

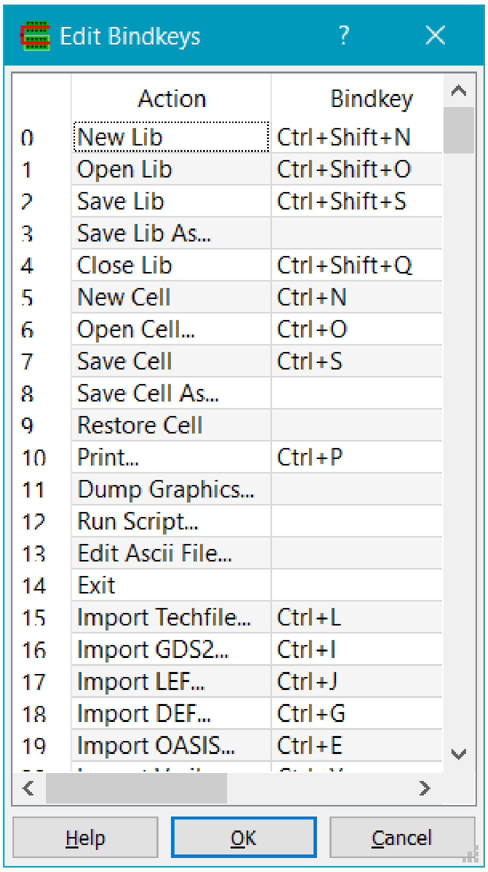


Figure 79 - Edit Bindkeys

All menus and toolbar buttons have actions. An action has a unique command associated with it; it also has an optional bindkey. For example the 'Open Cell' action by default has a bindkey Ctrl+O.

Bindkeys may be redefined by the user using the Edit Bindkeys command. This shows a table of all current bindkey assignments.

Clicking on the *Bindkey* entry in the table allows editing the bindkey for that *Action*. A single letter in uppercase indicates that key will be used. Modifier keys may be specified e.g. Shift+, Ctrl+, Alt+ and should precede the key, with no spaces.

Bindkeys are saved in the preferences file (~/.gladerc) and are loaded automatically every time Glade is run. A local .gladerc file will override values specified in the global ~/.gladerc file, so you have a project-specific subset of settings.

2.5.57 Create

Create commands for shapes (text, paths, polygons and rectangles) all work on the current layer as set in the LSW by left mouse clicking on the layer box. All the create commands pop up a dialog box which can be shown or hidden by pressing the F3 bindkey. Create commands can be terminated by hitting the Escape bindkey. Zooming and panning is possible during Create commands.

2.5.58 Create->Inst...

The **Create->Inst** command creates an instance or array in the current cell.

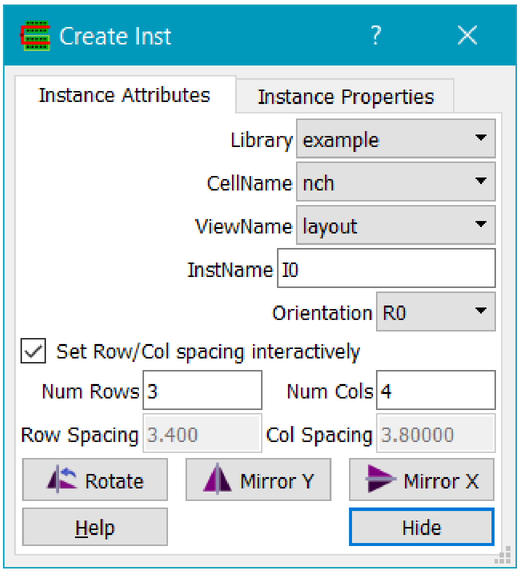


Figure 80 - Create Inst attributes

An instance is entered using a single left mouse click, which defines the origin of the instance. The instance master cell can be chosen from those present in the library using the *cellName* combo box and the *viewName* combo box. The instance's *InstName* is auto generated but can be changed by the user if required in the *instName* field. *Orientation* can be one of R0, R90, R180, R270, MX, MXR90, MY, MYR90. Arrays of instances can be generated if *Num Rows* and/or *Num Cols* is not 1; the spacing between rows and columns is set by *Row Spacing* and *Column Spacing*, unless *Set Row/Col spacing interactively* is checked. In that case, if *Num Rows* is greater than 1, the user is prompted for the location of the first instance of the second row (setting the rowSpacing), and if *Num Cols* is greater than 1, the user is prompted for the location of the first instance of the second column (setting the colSpacing). The instance bounding box is displayed during the command to assist in placement of the instance. *Rotate* (or the bindkey 'r' during instance placement) rotates the instance counter clockwise. *Mirror Y* (or the 'y' bindkey during instance placement) mirrors the instance about the Y axis. *Mirror X* (or the 'x' bindkey during instance placement) mirrors the instance about the X axis.

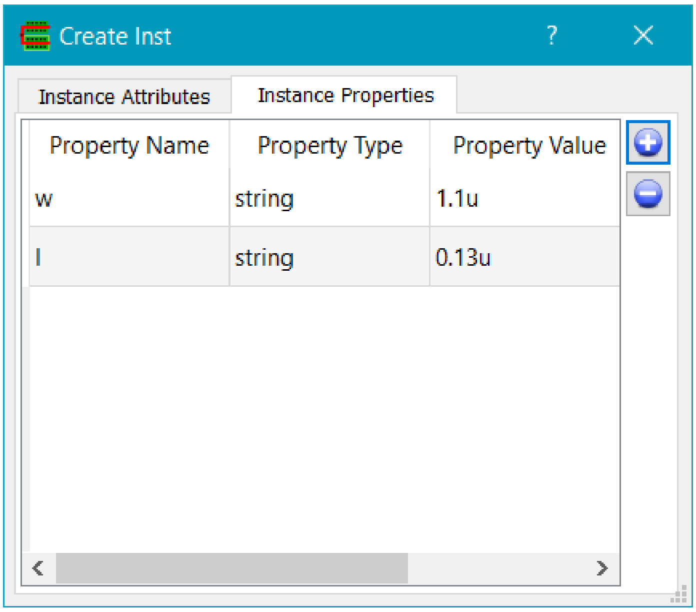


Figure 81 - Create Inst properties

The Instance Properties tab can be used to set properties on the instance, e.g. if the master cell is a PCell or a symbol. The '+' button adds a new property row. The *Property Name* column allows the property name to be edited. Clicking on the *Property Type* will display a combo box with the possible property types, e.g. string, integer, float etc. The *Property Value* column contains the property values.

2.5.59 Create->Rectangle

The **Create->Rectangle** command creates a rectangle in the current cell on the current layer. A rectangle is entered using two left mouse clicks, or a single click if Infix Mode is set. The first point (or the current cursor position if Infix mode is set) defines a vertex of the rectangle. A rubber band box is drawn showing the extent of the rectangle during mouse movement. Clicking on the second point completes the rectangle.

2.5.60 Create->Polygon

The **Create->Polygon…** command creates a polygon in the current cell on the current layer.

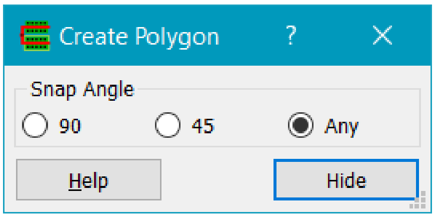


Figure 82 - Create Polygon

A polygon is entered using multiple left mouse clicks, with each click defining a vertex of the polygon. The polygon entry Snap Angle can be one of Manhattan, Diagonal or Any Angle. After two points have been entered, a dotted blue 'closure line' is shown. Hitting return while the closure line is displayed completes the polygon according to the closure line. Polygons can also be finished by double clicking on a point. Pressing the backspace key backs up the polygon by one vertex i.e. deletes the last point.

2.5.61 Create->Path...

The Create->Path... command creates a path in the current cell on the current layer.

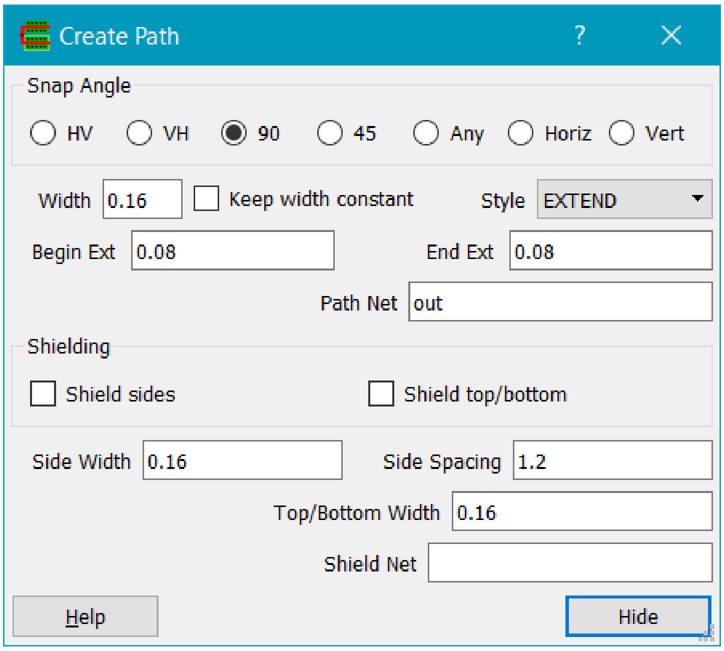


Figure 83 - Create Path

A path is entered using multiple left mouse clicks, with each click defining a path point. The path entry *Snap Angle* can be one of HV, VH, Any, 90, 45, Any, Horiz or Vert. HV will create a path with horizontal segment first, then vertical. VH will create a path with vertical segment first, then horizontal. 90 restricts entry to Manhattan, 45 to diagonal, and Any to all angles. Horiz and Vert restrict paths to horizontal and vertical segments only. The path width can be set in the *Width* field, and the path *Style* can be set to one of Truncate, Round, Extend, VarExtend or Octagonal. In the case of Extend, the path extent is set to half the path width. In the case of VarExtend, independent beginning and ending extensions can be set. Path entry is terminated by hitting the return key, or by double clicking on the final point required. Pressing the backspace key backs up the path by one vertex i.e. deletes the last entered point.

If the layer is a routing layer (has its FUNCTION set in the techFile to ROUTING) then the 'u' and 'd' keys can be used to switch up or down to the next routing layer during path entry. A via will be placed if a valid via between the two layers exists. If an existing net shape is selected, the *Path Net* and Layer are pre-set based on the net shape, and the *Width* field is set to that layer's minimum width. Note that if a via is entered during path entry, the previous path segment(s) are committed, so pressing the Esc key to interrupt the path command terminates it, preserving the already entered path segments. This is useful if you want to route one path to connect with another on an adjacent layer.

If the first or last point of a path is entered inside an instPin (e.g. of a PCell), then the path point is snapped to the centre of the instPin.This can considerably speed up wiring of layout generated by the schematic Layout->Create Layout command.

Shield sides, if checked, results in shield paths generated to the sides of the entered path, in the same layer as the path. The width of the side shield paths is set by *Side Width* and the spacing from the shield to the path is set by *Side Spacing*. *Shield top/bottom*, if checked, results in shield paths on top of and below the entered path, with the bottom shield on the next routing layer below the path, and the top shield on the next routing layer above the path. The routing order is as set in the techFile by the layer FUNCTION statements - the first layer with a FUNCTION of ROUTING is the lowest layer. The widths of the top and bottom shields are set by *Top/Bottom Width*. If *Shield Net* is set then the shields will be assigned to the net name specified; if it does not exist it will be created.

2.5.62 Create->Label...

The **Create->Label...** command creates a label in the current cell.

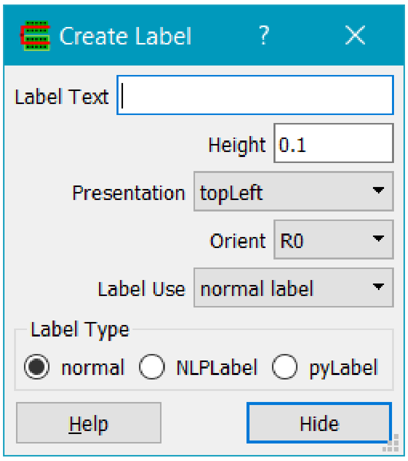


Figure 84 - Create Label

The label is created on the current layer with name given by the *Label Text* field. A label is entered using a single left mouse click, which defines the label's origin. *Height* defines the label's height. *Presentation* is the position of the origin relative to the text and can be one of topLeft, topCentre, topRight, centreLeft, centreCentre, centreRight, bottomLeft, bottomCentre or bottomRight. *Orient* can be one of R0, R90, R180, R270, MX, MXR90, MY, MYR90. *Label Use* sets the use of the label; for layout and schematic views this should be 'normal label'. For symbols, choosing a different use sets the layer the label is created on. *Label Type* sets the label type. A *normal* type label displays its label text as is. A *NLPLabel* label has the label text evaluated as an NLP expression. A *pyLabel* has its text evaluated as a Python expression.

2.5.63 Create->MultPartPath...

The Create->MultiPartPath... command creates a Multi Part Path (MPP).

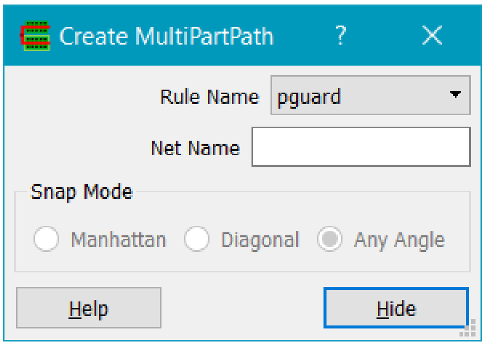


Figure 85 - Create MPP

The MPP is created according to a MPP *Rule Name*. Optionally a MPP may be assigned a *Net Name*. Currently MPP paths must be Manhattan only. MPPs when exported to GDS / OASIS etc. are converted to polygons i.e. they are flattened.

A MPP is defined in the techFile e.g. as below:

```
//
// MultiPartPath rules
//
MPP nguard LAYER nwell drawing WIDTH 1.80 BEGEXT 0.90 ENDEXT 0.9 ;
MPP nguard LAYER od drawing WIDTH 1.18 BEGEXT 0.59 ENDEXT 0.59 ;
MPP nguard LAYER nimp drawing WIDTH 1.54 BEGEXT 0.77 ENDEXT 0.77 ;
MPP nguard LAYER cont drawing WIDTH 0.16 BEGEXT -0.08 ENDEXT 0.08 SPACE 0.18
LENGTH 0.16 ;
MPP nguard LAYER metal1 drawing WIDTH 0.60 BEGEXT 0.30 ENDEXT 0.30 ;
```

A MPP is like a path in that it is defined as a set of vertices. A MPP may contain several layers. Each layer must have a nonzero WIDTH and a BEGEXT and ENDEXT which may be negative, positive or zero. The layer is justified to the segments of connected vertices i.e. it extends a half width either side of the path. The BEGEXT is the distance past the first vertex of the path that the layer starts, the ENDEXT is the distance past the last vertex of the path that the layer stops. If the layer has a SPACE and a LENGTH then it is assumed to be a repetitive contact structure, i.e. rectangles with WIDTH and LENGTH separated by SPACE are generated.

2.5.64 Create->Pin...

The **Create->Pin...** command creates a pin in the current cell.

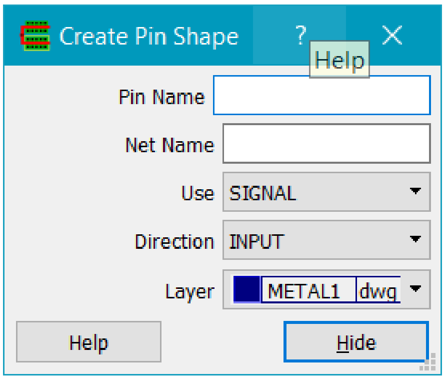


Figure 86 - Create Pin

Pin Name is the name of the pin, *Net Name* is the name of the net that the pin belongs to. If the net does not exist it will be created. If an existing net shape is selected, *Net Name* is seeded with the selected net's name. *Use* determines the pin's type. *Direction* sets the pin's direction. The pin is created on the *Layer* selected by the layer chooser field.

2.5.65 Create->Via...

The **Create->Via...** command creates a via in the current cell.

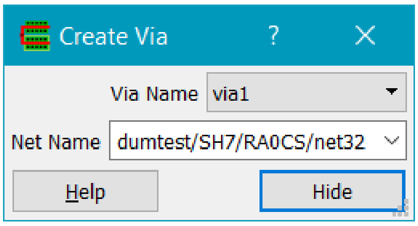


Figure 87 - Create Via

Via Name is the name of an existing via in the library. *Net Name* is the name of the net that this via is assigned to.

2.5.66 Create->Circle...

The **Create->Circle...** command creates a circle in the current cell on the current layer.

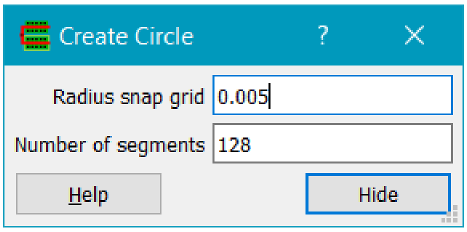


Figure 88 - Create Circle

Circles are entered using two left mouse clicks, or a single click if Infix mode is set. The first point (or the current cursor position if Infix mode is set) defines the centre of the circle, and the second point is a point on the circumference of the circle. *Radius snap grid* is the snap grid (usually the manufacturing grid) to snap the circle's radius to. *Number of segments* is the number of line segments used to represent the circle on export to GDS2 or OASIS.

2.5.67 Create->Ellipse...

The **Create->Ellipse...** command creates an ellipse in the current cell on the current layer.

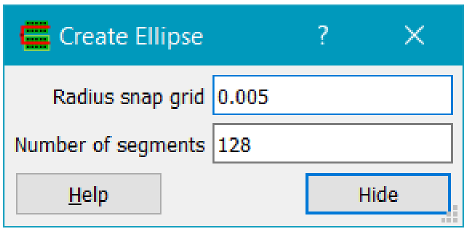


Figure 89 - Create Ellipse

Ellipses are entered using two left mouse clicks, or a single click if Infix mode is set. The first point (or the current cursor position if Infix mode is set) defines one corner of the ellipse's bounding box, and the second point defines the opposite corner of the ellipse's bounding box. *Radius snap grid* is the snap grid (usually the manufacturing grid) to snap the ellipse's bounding box to. *Number of segments* is the number of line segments used to represent the circle on export to GDS2 or OASIS.

2.5.68 Verify

2.5.69 Verify->Check...

The Verify->Check... command displays the Check dialog.

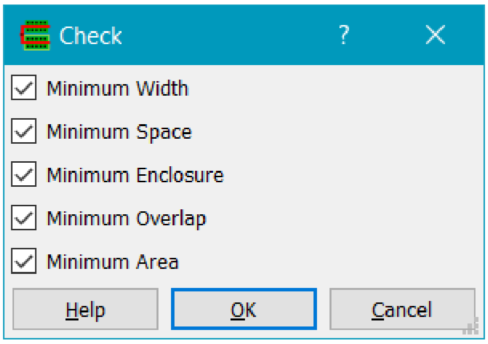


Figure 90 - Verify Check

Width and spacing error checks can be performed if e.g. layer minWidth and minSpace properties have been set in the techFile or via loading LEF.

2.5.70 Verify->Check Offgrid...

The Verify->Check Offgrid... command displays the Check Offgrid dialog.

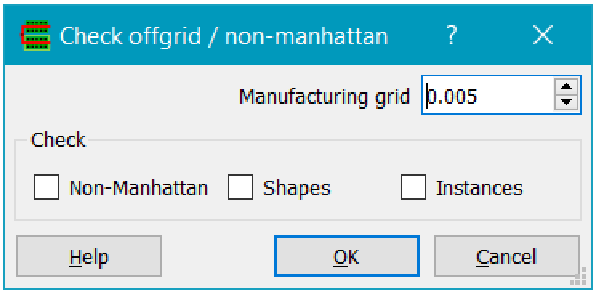


Figure 91 - Check Offgrid

If *Non-Manhattan is checked*, it will check and select any paths or polygons that contain non-Manhattan edges. If *Shapes* is checked, it will check if any shape vertices are not on the specified *Manufacturing grid*. If *Instances* is checked, it will check if instance origins are on grid, and in the case of arrays that the rowSpacing and colSpacing values are an integer multiple of the manufacturing grid. The command works on the top level cell only currently.

2.5.71 Verify->DRC->Run...

The **Verify->DRC->Run...** command displays the Run DRC dialog.

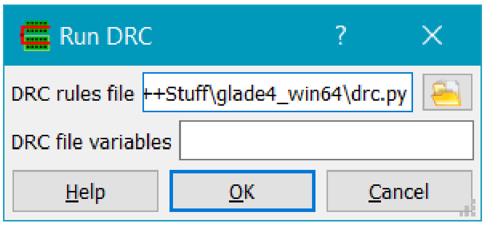


Figure 92 - Run DRC

DRC rules file specifies a python rules file to run. Any existing DRC violation markers are erased. If an environment variable GLADE_DRC_FILE is set, the *DRC rules file* will be set to the value of GLADE_DRC_FILE (which must be a full path name). *DRC file variables* can be set using the DRC file variables entry; both the name and value are passed to Python as strings. Options should be in the form of name=value, separated by a space. If an environment variable GLADE_DRC_VARS is set, the variables will be set to the value of this env var.

2.5.72 Verify->DRC->View Errors...

The **Verify->DRC->View Errors...** command shows the DRC error marker dialog.

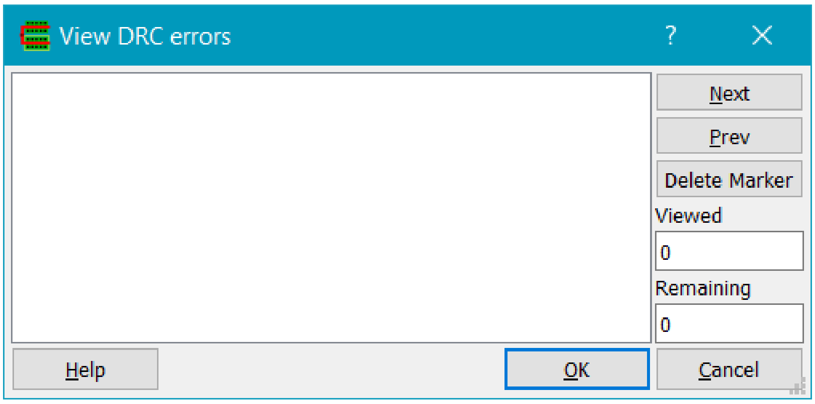


Figure 93 - View DRC errors

Click on a rule violation in the left hand list box to select the type of violation you wish to view. The first violation marker will be selected and zoomed in on. Subsequently you can use the *Next* and *Prev* buttons to step through the list of violation markers, zooming in on each new marker. The *Delete Marker* button will delete the currently viewed error marker. *Viewed* is the number of violations viewed so far; *Remaining* is the total number of violations remaining (the starting number less the number deleted using Delete Marker).

The current DRC marker is highlighted, and other shapes can be dimmed if the Selection Options Dim highlighted shapes option is set.

2.5.73 Verify->DRC->Clear Errors

The Verify->DRC->Clear command clears all errors on the drcMarker layer.

2.5.74 Verify->Extract->Run...

The Verify->Extract->Run... command displays the Run Extraction dialog.

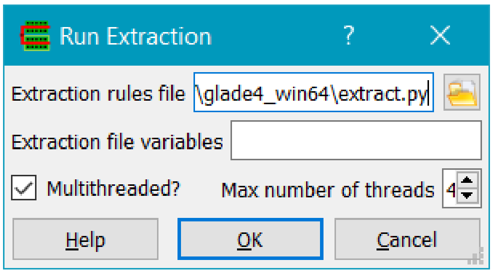


Figure 94 - Run Extraction

Extraction rules file specifies a python rules file to run. If an environment variable GLADE_EXT_FILE is set, the extraction rules file will be set to the value of GLADE_EXT_FILE (which must be a full path name). *Extraction file variables* can be set using the Extraction file options entry. Options should be in the form of name=value, separated by a space; both the name and value are passed to Python as strings. *Multithreaded* if checked runs connectivity analysis in the number of threads specified in *Max number of threads*, which defaults to the maximum number of logical cores the machine can use. If an environment variable GLADE_EXT_VARS is set, the variables will be set to the value of this env var.

2.5.75 Verify->LVS->Run...

The **Verify->LVS->Run...** command displays the Run LVS dialog. Glade uses Gemini for LVS, which is run with netlists generated from the extracted view and schematic view or netlist.

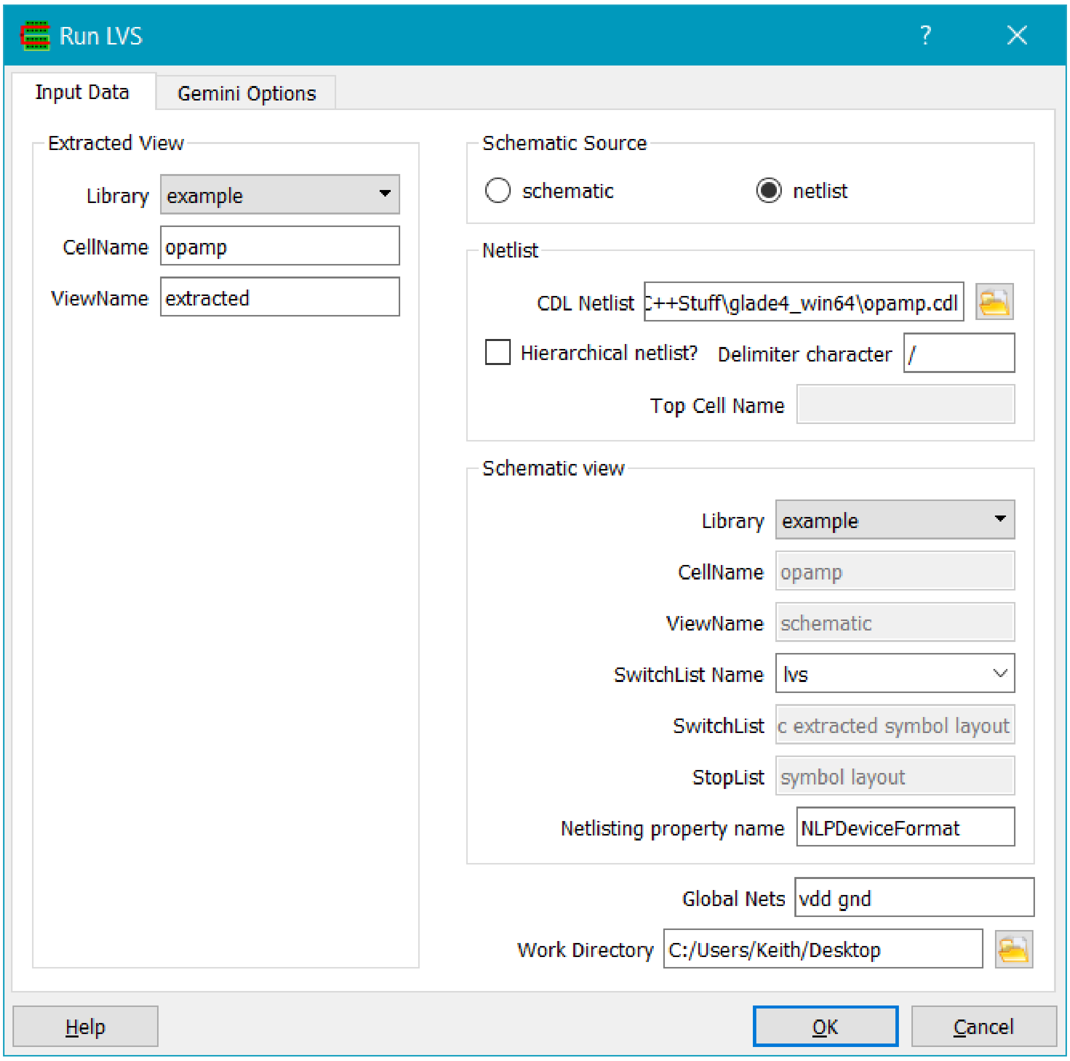


Figure 95 - Run LVS

Extracted View specifies the name of an existing extracted view in the Library/CellName/ViewName fields and is pre-set to the current cellView. *Schematic Source* can be either *schematic* or *netlist*.

If Schematic Source is *Netlist*, specify the name of the schematic netlist in the *CDL Netlist* field. This can be pre-set via an environment variable GLADE_NETLIST_FILE.

If *Hierarchical netlist?* is checked, then a hierarchical netlist will be flattened before passing to the LVS engine. *Delimiter character* specifies the delimiter between hierarchical names of nets and instances. *Top Cell Name* must be specified for a hierarchical netlist; it is the top level .subckt name that corresponds to the design to be verified.

If *Schematic Source* is *Schematic*, specify the Library/CellName/ViewName for the schematic, and the *SwitchList* and *StopList* for netlisting.

Switch List and *Stop List* set the switch and stop lists for the netlister during hierarchical netlisting, and are space-delimited lists of view names. Switch and stop lists are named in *SwitchList Name*. To create a new name group, edit the *SwitchList Name* and set the Switch List and Stop List. The new named group will be saved in the gladerc.xml preferences file.

Global Nets specifies global nets for the CDL netlist file.

The netlister property name can be set in the *Netlisting property name* field. It is a space-delimited list of names of the property to be used for netlisting; the first found is used, else the property name 'NLPDeviceFormat' is used.

Working Directory specifies a directory where temporary files are written. The extracted view is netlisted to a CDL file which is compared to the specified schematic netlist. Match info and/or discrepancies are written to the log file.

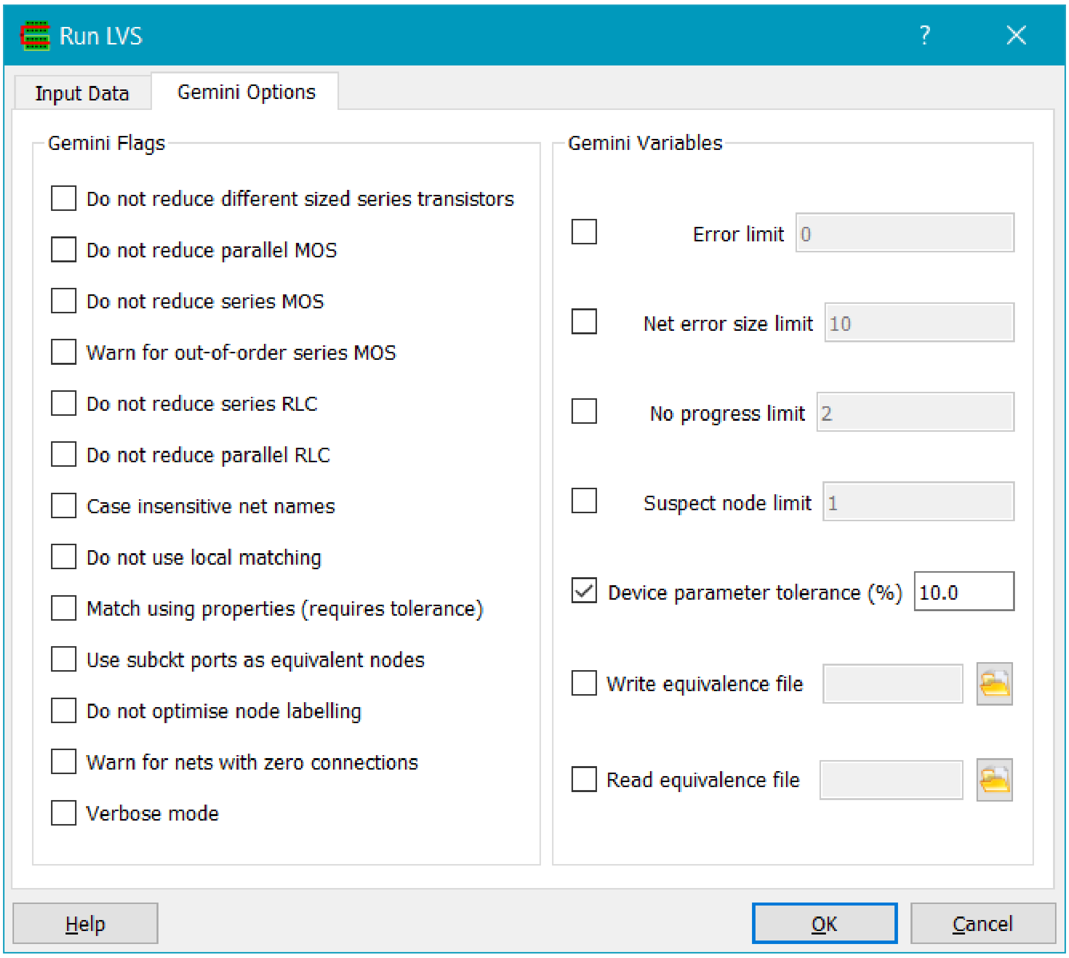


Figure 96 - Run LVS Options

The *Gemini Options* tab allows specification of Gemini options.

Do not reduce different sized series transistors - Normally series MOS devices are merged; checking this option prevents the merge if they have different L/W.

Do not reduce parallel MOS - Parallel (or multi-fingered) MOS devices with the same S/D/G/B nets are normally collapsed into a single device whose width is the sum of the individual widths; checking this option prevents merging.

Do not reduce series MOS - This option prevents series MOS devices from being merged.

Warn for out-of-order series MOS - If series merged transistors match, but have different gate net order, then a warning will be issued.

Do not reduce series RLC – This option prevents series R, L or C devices from being reduced into a single device. In the case of series R and L, their values are summed for the reduced device. In the case of series C, the reciprocals are summed.

Do not reduce parallel RLC – This option prevents parallel R, L or C devices from being reduced into a single device. In the case of parallel R or L, the reciprocals of their values are summed. In the case of C, their values are summed.

Case insensitive net names - Normally net names are case sensitive i.e. clk and Clk are different nets; checking this option treat them as the same net.

Do not use local matching - this option stops Gemini from using the local matching algorithm to speed up checking. It is normally never required.

Match using properties - Gemini does not consider device properties (e.g. W, L) when matching devices. Checking this option can resolve some symmetric circuits which have different device properties. The Device Parameter Tolerance option must also be checked.

Use subckt ports as equivalent nodes - Gemini defaults to matching netlists without any initial equivalence points. If checked, then if both netlists have .subckt/.ends lines, the port names in the .subckt line will be used for equivalence. Errors will be generated if the number of ports is different, or if the port names do not match. This option can resolve some symmetric circuits where the difference is with the port names.

Do not optimise node labelling - Gemini will try and optimise node labelling to assist matches. This option is normally never required.

Warn for nets with zero connections - Gemini will report an error for nets with no connections. Not very useful.

Verbose mode - Gemini will generate extra information in the message window while running.

Error limit - Sets the allowed number of errors. The default is zero.

Net error size limit - Gemini will report this number of devices connected to unmatched nets. Typically if e.g. power/ground nets mismatch, a lot of errors can be generated. The default limit is to report 10.

No progress limit - Sets the iteration limit if no progress is made relabelling nodes to try and find a match.

Suspect node limit - Sets the limit of suspect nodes allowed.

Device parameter tolerance - if checked, device properties e.g. W/L, R, C, L are checked according to the specified tolerance, else properties are not checked.

Write equivalence file - Writes a file containing equivalent net names between the layout and schematic. Each line has the format '= <name1> <name2>'.

Read equivalence file - Reads a equivalence file containing net names to be considered as initially matched in the format '= <name1> <name2>. This can sometimes help with circuits with symmetry, or many errors. Note that incorrectly specifying match names can result in wrong results.

Gemini will write mismatch information to the message window, and to the open extracted view as error markers for nets/devices.

2.5.76 Verify->Import Hercules Errors

The Verify->Import Hercules Errors command imports a Hercules error file.

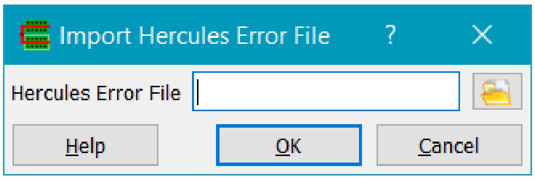


Figure 97 - Import Hercules Errors

The DRC error viewer can then be used to step through the errors.

2.5.77 Verify->Import Calibre Errors

The Verify->Import Calibre Errors command imports a Calibre error file.

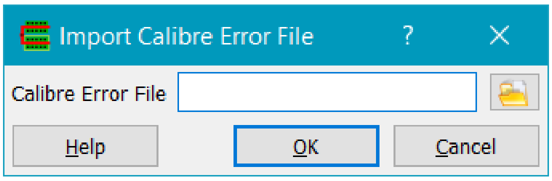


Figure 98 - Import Calibre Errors

The DRC error viewer can then be used to step through the errors. Both flat and hierarchical Calibre error files are supported.

2.5.78 Verify->Compare Cells...

The Verify->Compare Cells... command allows comparison of layers from two different cellViews using a multithreaded XOR operation. This can be useful for checking the changes made between two different versions of a design.

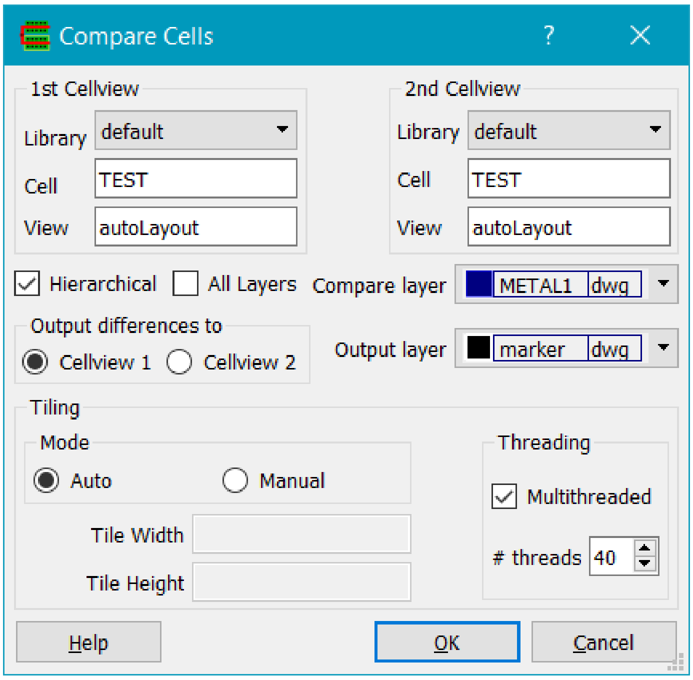


Figure 99 - Compare Cells

The data to be compared is specified by the *1st CellView* and *2nd CellView*. They are compared by the Compare layer using an XOR. If *Hierarchical* is checked, the cells are flattened for the compare, else only shapes on *the Compare layer* at the top level will be compared. If *All Layers* is checked, all the layers are compared, else just the *Compare Layer* .The results are output to the cellView specified in the *Output differences to field*. Results are written to the *Output layer* specified - if marker is used (the default), then differences can be viewed using the DRC->View Errors command.

Comparison is done by multithreaded tiling of the original data to handle large designs. For setting tile sizes and multithreading options, see the Tiled Boolean Operations command.

Note: if you want to compare two cells by importing e.g. GDS2 or OASIS files, you MUST use import a techFile for each import with the same GDS layer/datatype to layer/purpose mapping. Failing to do this (e.g. importing two GDS2 files without importing any techFile) will most likely result in the GDS layer/datatype of one file being assigned to a different internal layer number from that of the second file. This is because internal layer numbers represent layer/datatype or layer/purpose pairs as a single number, and the mapping is assigned in the order the pairs are encountered.

2.5.79 Verify->Trace Net

The Verify->Trace Net command displays the Trace Net dialog.

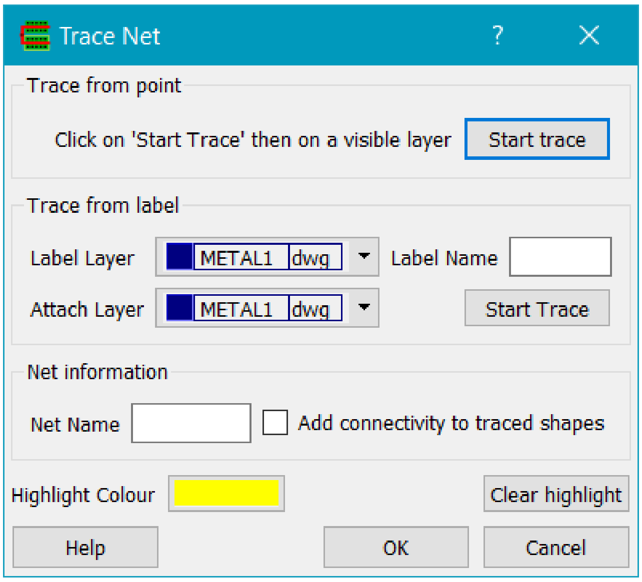


Figure 100 - Trace Net

Trace Net traces connectivity either from a start point or from a text label. To use the net tracer, the technology file must have layers with their CONNECT attributes defined. For example:

CONNECT poly drawing BY contact drawing TO metal1 drawing ;
CONNECT metal1 drawing BY via1 drawing TO metal2 drawing ;
CONNECT metal1 drawing TO metal1 drawing ;

In the first two cases, connection of the poly layer to the metal1 layer is through a via layer. In the second case, the two layers connect without any via layer.

To use the net tracer to trace from a shape, click on the *Trace from point - Start Trace button*. Tracing will continue until no more connecting shapes are found. Tracing may be aborted by clicking on the red abort button next to the progress bar on the status bar.

To use the net tracer to trace from a text label, set the *Label Layer* and the *Attach Layer* (which can be the same layer). Enter the *Label Name* and click on *Trace from label - Start Trace*. Note that currently, only labels on the top level of the cellView hierarchy can be traced from.

If the checkbox *Add connectivity to traced shapes* is set, then a net with the name given by *Net Name* will be created f it does not already exist, and all traced shapes will be assigned to that net. NB entering a *Label Name* for tracing from a text label will automatically set the *Net Name* field.

Clicking on *Highlight Colour* will alllow changing the highlight colour. *Clear highlight* clears all highlighted shapes. Using the selection options dialog to dim unhilited objects can make the trace result clearer.

2.5.80 Verify->Set Layer Stack

The Verify->Set Layer Stack command displays the layer connectivity used for the Trace Net command (and also settable in the techFile as described above, or set during import of LEF/DEF).

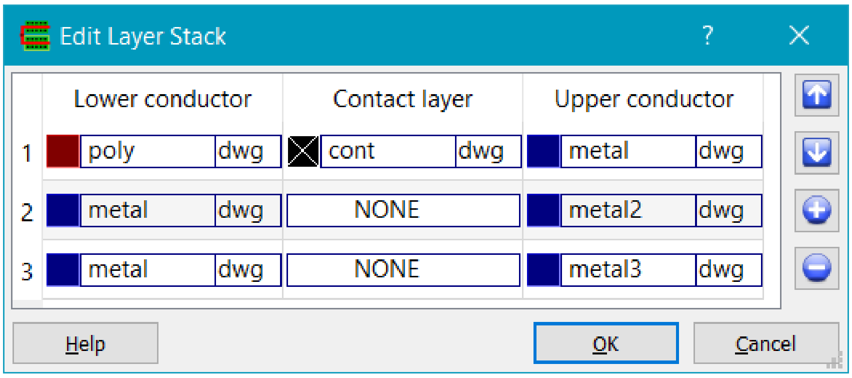


Figure 101 - Set Layer Stack

The dialog displays lower conducting layers, optional contact layers and upper conduction layers. So in the above dialog, poly connects to cont which connects to metal2. metal2 connects to metal, and metal also connects to metal3.

To edit a layer, double click on it and the icon will change to a combo box as shown in the second column, third row of the dialog above. You can set the layer to any layer including a special layer 'NONE' which when used for the contact layer means there is no explicit contact layer between the conducting layers.

To add a row, use the '+' buttton. To delete a row, select the row by single clicking on it, then use the '-' button. To move a row of layers up, select a row and use the 'up' bottom. Similarly to move a row down, select the row and use the 'down' button.

2.5.81 Verify->Short Tracer...

The Verify->Short Tracer... command displays the short tracer dialog.

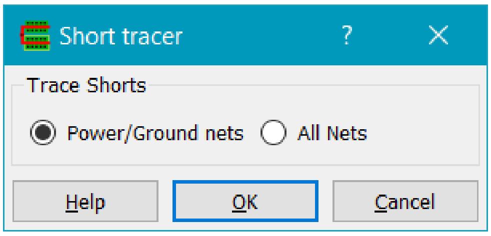


Figure 102 - Short Tracer

The Short Tracer can be used for DEF or similar designs that have connectivity. Either *Power/Ground nets*, or *All Nets* can be checked for touch/overlap against shapes connected to a different net. The bounding box of the shorting region is reported, and this bounding box is written to the marker layer so that the DRC->View Errors dialog can be used to step through the errors, zooming to each short location. Currently only top level nets are checked against other top level nets; in other words the check is done flat.

2.6 Schematic Menus

2.6.1 View

2.6.2 View->Fit

See Layout View menu

2.6.3 View->Fit+

See Layout View menu

2.6.4 View->Zoom In

See Layout View menu

2.6.5 View->Zoom Out

See Layout View menu

2.6.6 View->Zoom Selected

See Layout View menu

2.6.7 View->Pan

See Layout View menu

2.6.8 View->Redraw

See Layout View menu

2.6.9 View->Ruler

See Layout View menu

2.6.10 View->Delete Rulers

See Layout View menu

2.6.11 View->Cancel Redraw

See Layout View menu

2.6.12 View->Display Options

The View->Display Options command displays the Display Preferences dialog.

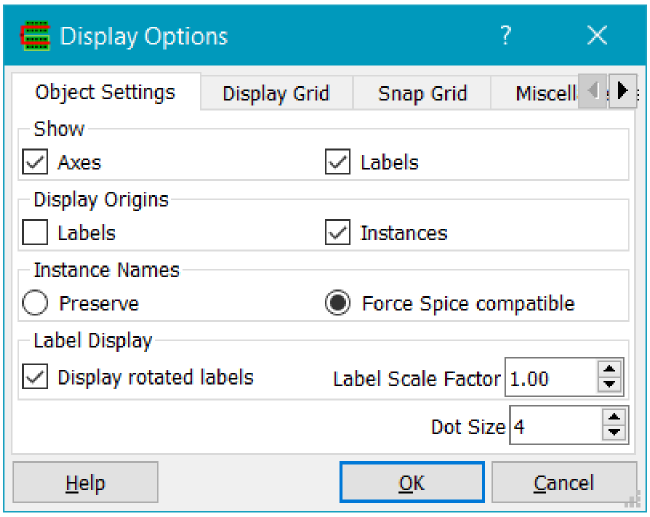


Figure 103 - Display Options (Object Settings)

Show Axes shows the X=0 and Y=0 axes.

Show Labels toggles the display of text labels. By default, text label display is turned off as in non-OpenGL display mode, drawing text labels can be slow if there are many labels.

Display Origins - Labels shows the origin of text labels as a small cross. *Display Origins - Instances* shows the origin of instances as a small cross.

Instance Names can be set to *Preserve* or *Force Spice Compatible*. With *Preserve*, the instance names are kept as is. With *Force Spice Compatible*, the first character of the instance name will be changed during Check to a Spice type e.g. M for MOS devices, R for resistors etc. depending on the instance master's type property.

Label Display allows finer control of text labels. *Display Rotated Labels* if checked displays text rotated as per its database orientation. When unchecked, labels are displayed with no rotation (horizontally).

Label display scale factor will scale the displayed labels according to the scale factor set. A different scale factor can be used for schematics and layout.

Dot Size sets the dot size when creating solder dots, either through interactive wiring or via the Create Solder Dot command.

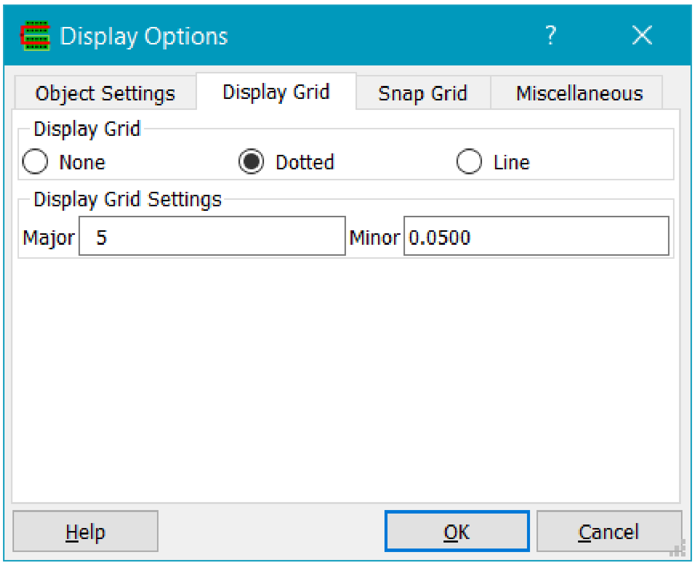


Figure 104 - Display Options (Display Settings)

Display Grid controls the display grid which can be one of *None*, *Dotted* or *Line*. The display major grid is drawn using the LSW majgrid layer; the minor grid is drawn using the LSW mingrid layer.

Display Grid Settings. The *Minor* grid spacing sets the dot or line spacing and are drawn using the mingrid layer. The *Major* value is the number of minor grids per major grid dot or line; it should be an integer, typically 5 or 10. The major grid is drawn using the majgrid layer.

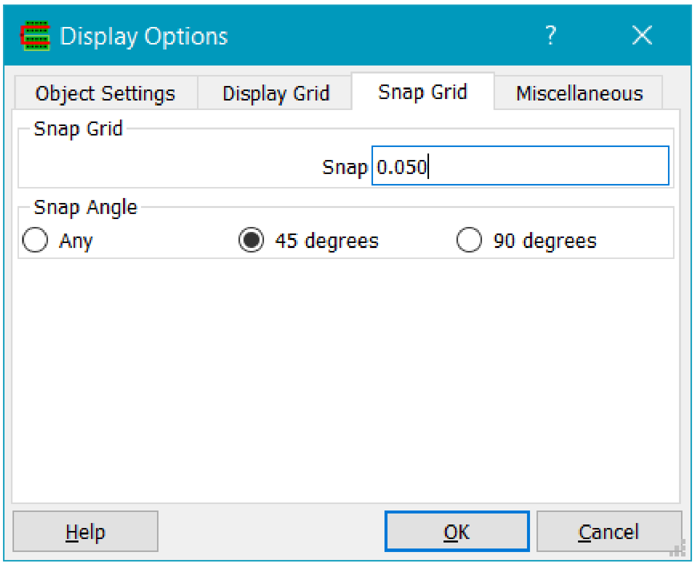


Figure 105 - Display Options (Snap Settings)

Snap Grid controls cursor snapping. The cursor is snapped to the value specified. Snapping is modified by gravity; see the Selection Options dialog.

Snap Angle controls the angle that data can be entered for some shape creation and also for rulers. *Any* allows all angles; *45 degrees* and *90 degrees* snap accordingly.

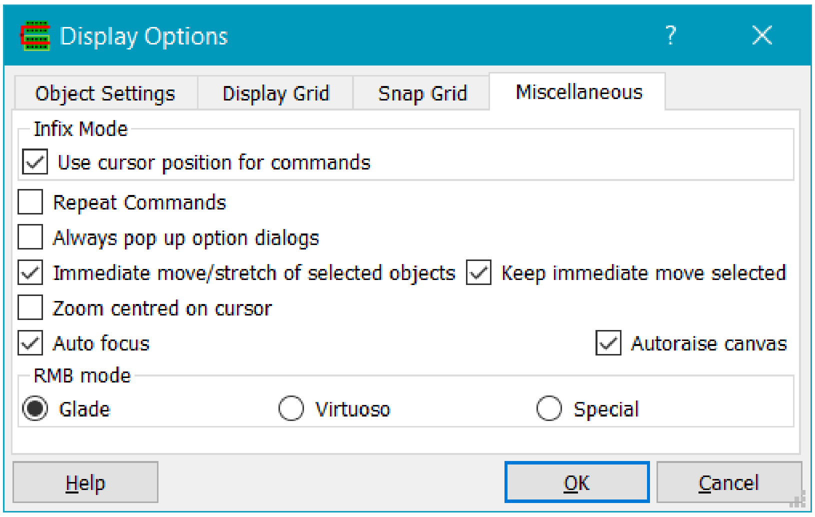


Figure 106 - Display Options (Miscellaneous)

Infix Mode is used for commands which can take the current mouse position rather than relying on the user to click on the first point of the command.

Repeat commands will keep repeating a command until ESC is pressed.

Display coordinates in database units shows coordinates in DB units, rather than microns. This can be useful when working with e.g. DEF files where the ascii coordinates in the file are in DB units.

Always popup option dialogs when checked will always show option dialogs for forms such as Create Path. These option dialogs can be shown and hidden by toggling the F3 key. If Always popup option dialogs is not checked, then the option forms will not be shown automatically (but can still be shown by pressing F3). This is useful when entering e.g. a lot of polygons.

Immediate move/stretch of selected objects will let selected objects be moved by the cursor without issuing a move/select command. The cursor changes according to the object. To use, select an object in full mode, or an edge/vertex in partial mode. The cursor will change to a 4-way arrow (for full mode select) or a 2-way arrow (for partial mode select). Then left click and drag to move or stretch the object. The object is deselected afterwards, so to repeat the command, select another object.

Keep immediate move selected will keep objects selected after an immediate move/stretch; otherwise all objects will be deselected.

Zoom centred on cursor sets the centre of the zoom to the cursor position; otherwise zoom in/out is centred on the viewport.

Auto focus sets input focus to the canvas whenever the mouse moves over it. If this option is unchecked, then the user has to explicitly click on the main window in order to e.g. use bindkeys after any operation that transfers focus to another window. Some Window managers may override this operation because they provide control of focus directly.

Auto raise raises the canvas window the mouse is over automatically. If this option is not set, the canvas must be explicitly clicked on to make it the active window for accelerator key input.

RMB mode sets the operation of the right mouse button. It can be set to *Glade* mode (dragging the mouse down zooms in, dragging it up zooms out), *Virtuoso* mode (dragging the right mouse in any direction zooms in) or *Special* mode (dragging the right mouse down zooms in, dragging it up left zooms out, dragging it up right does a window fit).

2.6.13 View->Selection Options

See Layout View menu

2.6.14 View->Pan/Zoom Options...

See Layout View menu

2.6.15 Edit

2.6.16 Edit->Undo

See Layout View menu

2.6.17 Edit->Redo

See Layout View menu

2.6.18 Edit->Yank

See Layout View menu

2.6.19 Edit->Paste

See Layout View menu

2.6.20 Edit->Delete

See Layout View menu. When deleting wires, solder dots and labels associated with the wire will be deleted.

2.6.21 Edit->Copy

See Layout View menu. Copying vector instances will also copy the vector information to the new instance name.

2.6.22 Edit->Move

See Layout View menu

2.6.23 Edit->Move By...

See Layout View menu

2.6.24 Edit->Move Origin

See Layout View menu

2.6.25 Edit->Stretch

See Layout View menu

2.6.26 Edit->Rotate

See Layout View menu

2.6.27 Edit->Set Net

The **Edit->Set Net** command sets a selected shape's net.

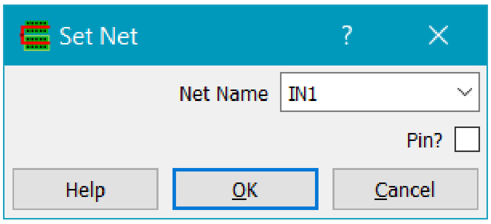


Figure 107 - Set Net

The *Net Name* combo box is filled with any existing net names in the cellView, or you can type in a net name to create that net. If *Set As Pin?* is checked, the shape(s) will become pin shapes.

2.6.28 Edit->Hierarchy->Ascend

The **Edit->Hierarchy->Ascend** command ascends one level of hierarchy, assuming you have previously descended into a cellView's hierarchy.

2.6.29 Edit->Hierarchy->Descend

The Edit->Hierarchy->Descend command descends into the selected instanceor tries to find an instance under the cursor to descend into if nothing is selected.

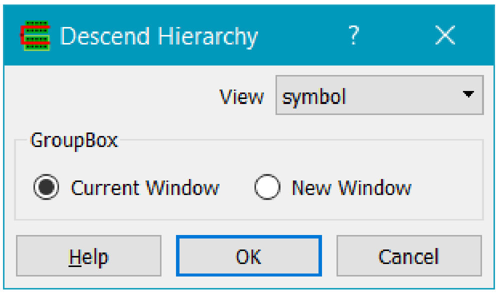


Figure 108 - Hierarchy Descend

View is the view of the instance to descend into; for example a schematic instance may have both a symbol view and a schematic (lower level of hierarchy) view. *Open In* controls the window used to display the cellView; *Current Window* uses the existing window, and the **Edit->Hierarchy->Ascend** command can be used to return to the previous cellView in the hierarchy. *New Window* opens a new window for the cellView, leaving the previous cellView window open.

2.6.30 Edit->Select->Inst by name

The Edit->Select->Inst By Name command Displays allows selection of instances based on their instance name.

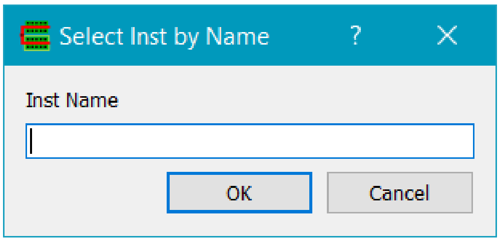


Figure 109 - Select Inst By Name

2.6.31 Edit->Select->Net by Name

The **Edit->Select->Net By Name** command allows selection of nets based on their name.

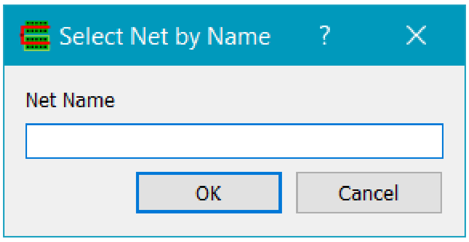


Figure 110 - Select Net By Name

2.6.32 Edit->Select->Select All

The **Edit->Select->Select All** command selects all currently selectable objects.

2.6.33 Edit->Select->Deselect All

The Edit->Select->Deselect All command deselects all the selected set.

2.6.34 Edit->Properties->Query Object

The **Edit->Properties->Query Object** command queries the selected object.

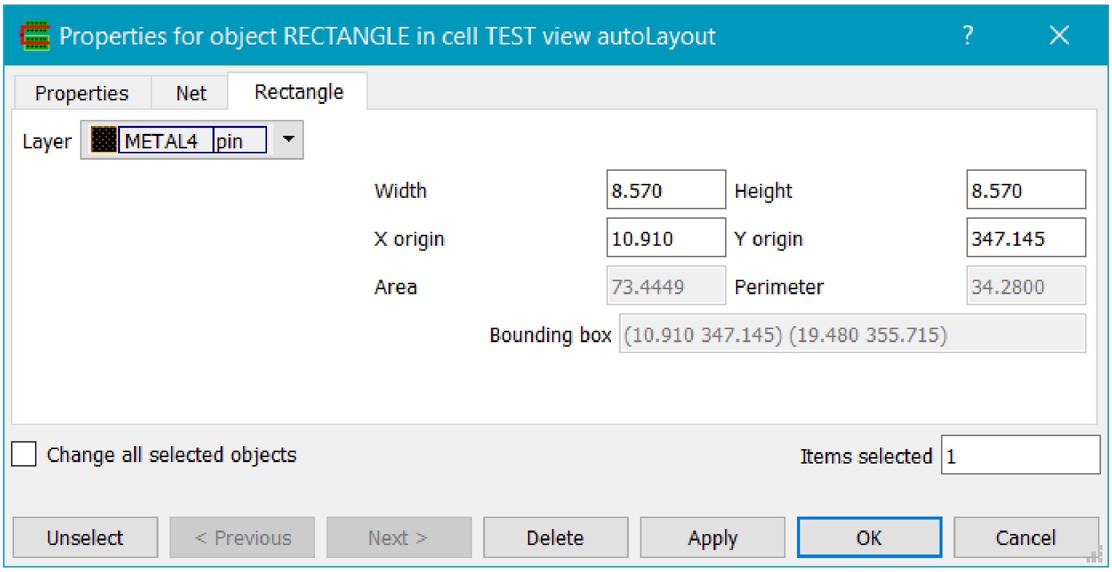


Figure 111 - Query Object

With nothing selected, the current cell's properties are queried. Otherwise you may query any selected object's properties and attributes, and cycle through the selected set using the *Previous* and *Next* buttons. You can delete a queried object using the *Delete* button. You can remove an object from the selected set with the *Unselect* button. If multiple objects are selected, *Change all selected objects* allows their common attributes to be changed. For example, if shapes are selected then the layer may be changed for all shapes. If the object has connectivity, a Net properties tab is added to the dialog. All objects may have user or system-defined properties which can be manipulated on the Properties tab page.

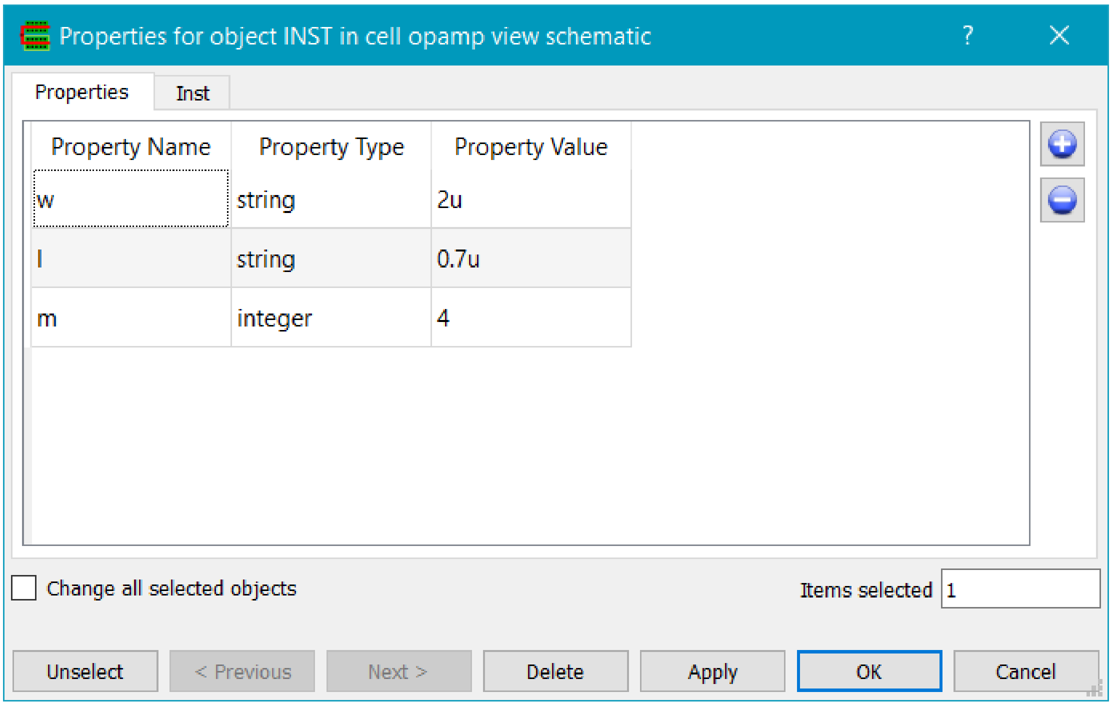


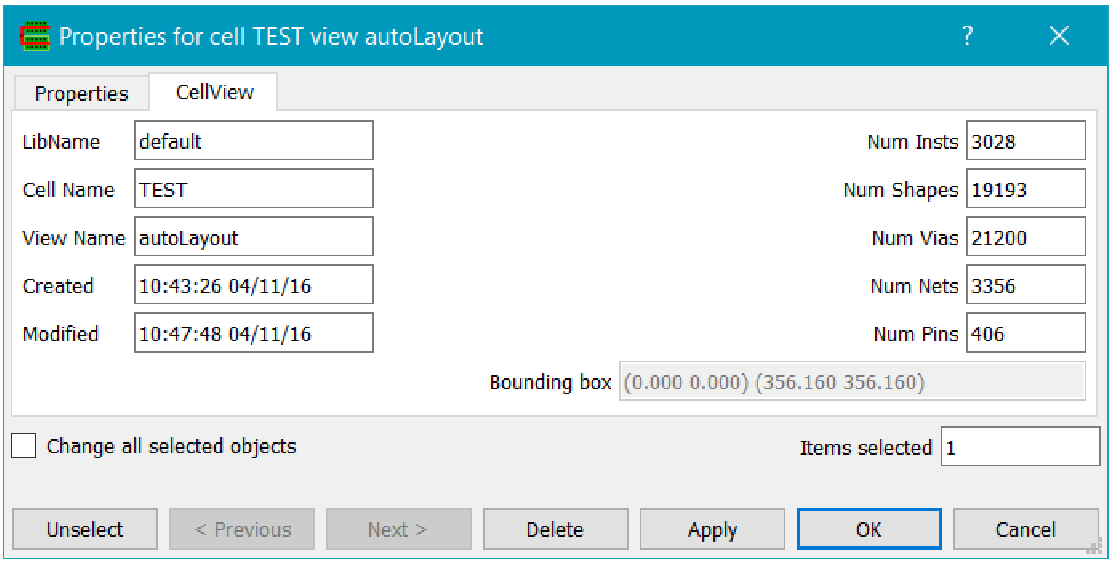
Figure 112 - Query Object Properties

Properties can be added as string, float, integer, boolean, list or orient. Click on the property name or value to change the text, or click on the type and select the type in the combo box that will appear. Click on the '+' button to add a (initially blank) property entry, or select a property and click on the '-' button to delete the property.

There is currently no undo capability if you delete a property.

2.6.35 Edit->Properties->Query CellView

The **Edit->Properties->Query CellView** command displays the query dialog for the current cellView.



2.6.36 Edit->Search...

The **Edit->Search...** command displays the Search dialog.

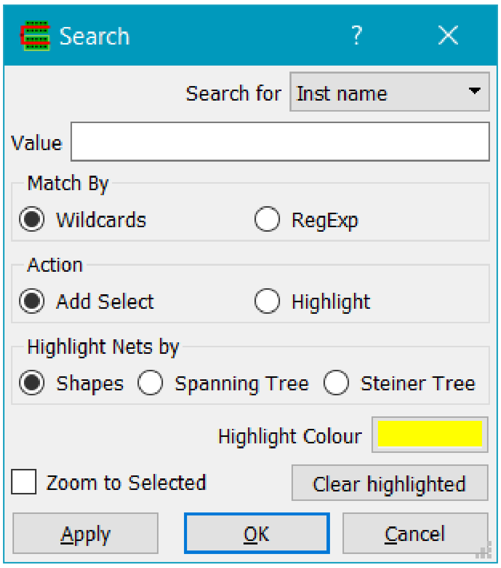


Figure 113 - Search

Find searches for instances by name, instances by master name (cell name), nets by name or text labels by name. Names can be matched by *Wildcard* (e.g. VDD* matches VDD1, VDD2, VDD) or by *RegExp* (regular expressions). Objects that match the selection criteria can be added to the selected set or highlighted. In the case of highlighted nets, they can be displayed either as the actual net shapes highlighted, or by a *Spanning Tree* between the instance pins of the instances the net connects to, or as *a Steiner tree*. This is useful, for example, in highlighting the connectivity of unrouted nets; the spanning tree is a good approximation to the path an autorouter will take; the Steiner tree is even better although can be slow on nets with many pins. The colour can be chosen using the *Highlight Colour* button. Optionally the display can *Zoom to Selected* object(s) and it is possible to clear all highlighted objects using the *Clear highlighted* button.

2.6.37 Edit->Bindkeys

The Edit->Bindkeys command displays the Edit Bindkeys dialog.

2.6.38 Create

2.6.39 Create->Instance…

The **Create->Instance…** command displays the Create Instance dialog.

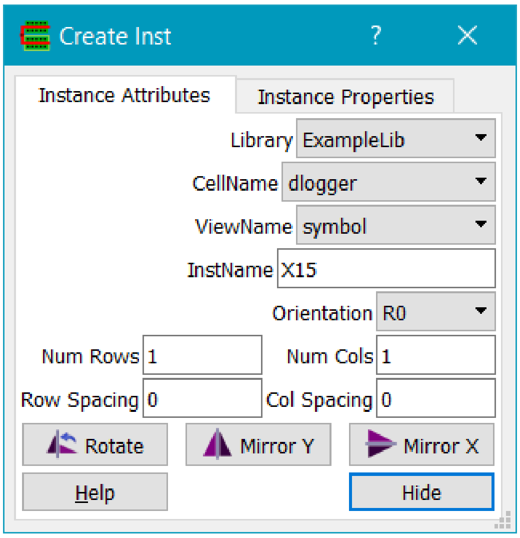


Figure 114 - Create Schematic Instance

An instance is entered using a single left mouse click, which defines the origin of the instance. The instance master cell can be chosen from those present in the library using the *cellName* combo box and the *viewName* combo box. The instance's *InstName* is auto generated but can be changed by the user if required in the *instName* field. *Orientation* can be one of R0, R90, R180, R270, MX, MXR90, MY, MYR90. Arrays of instances can be generated if *Num Rows* and/or *Num Cols* is not 1; the spacing between rows and columns is set by *Row Spacing* and *Column Spacing* respectively. The instance bounding box is displayed during the command to assist in placement of the instance. *Rotate* (or the bindkey 'r' during instance placement) rotates the instance counter clockwise. *Mirror Y* (or the 'y' bindkey during instance placement) mirrors the instance about the Y axis. *Mirror X* (or the 'x' bindkey during instance placement) mirrors the instance about the X axis.

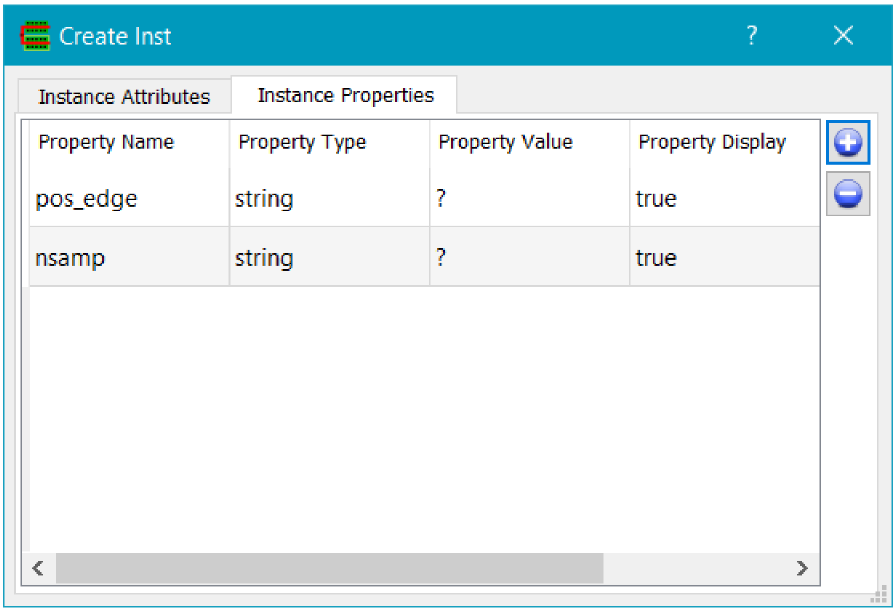


Figure 115 - Create Instance Properties

The Instance Properties tab can be used to set properties on the instance, e.g. if the master cell is a PCell or a symbol. The '+' button adds a new property row. The *Property Name* column allows the property name to be edited. Clicking on the *Property Type* will display a combo box with the possible property types, e.g. string, integer, float etc. The *Property Value* column contains the property values. If *Property Display* is set to true, the property is displayed in the schematic.

2.6.40 Create->Wire...

The **Create->Wire...** command displays the Create Wire dialog.

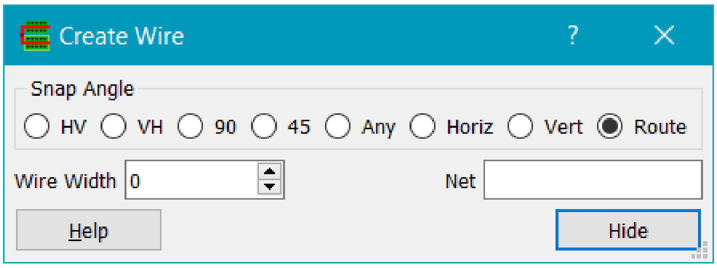


Figure 116 - Create Wire

Create a wire at the initial point (either the current cursor position, if infix mode is on) or by a first point entered by a left mouse click. Subsequent left mouse clicks add wire end points; use the backspace key to back up an entered point, and use the return key or double click to end a wire. If the wire starts or ends on another wire midpoint, a solder dot is automatically entered at the junction of the two wires. If the wire starts or ends on the endpoint of an existing wire, the two wires will be merged into a single continuous wire. If you click on a pin (either an IO pin or a device pin) or click on a wire, the wire entry is ended.

Snap Angle sets the snap direction when entering a wire.

- *HV* means the wire will be created with a horizontal segment followed by a vertical segment.

- *VH* means the wire will be created with a vertical segment followed by a horizontal segment. *90* means the wire will snap to Manhattan directions.
- *45* means the wire will snap to 45 degree directions.
- *Any* means the wire can have any direction.
- *Horiz* means the wire can only be entered in a horizontal direction.
- *Vert* means the wire can only be entered in a vertical direction.
- *Route* will use an autorouter to route a wire from the initial point to the current cursor position. The routing avoids obstructions (symbol boundary shapes and symbol pins); if the current cursor position is over an obstruction the routed path is shown dashed as a straight line from initial to current point, else it is shown in full as a solid line.

Wire Width sets the display width of the wire. A value of 0 or 1 means 1 pixel wide. *Net* can be used to pre-set the net name for the wire; it is not necessary in most cases as a subsequent Check CellView command will extract connectivity.

2.6.41 Create->Solder Dot

The **Create->Solder Dot** command creates a solder dot at the point entered by the cursor. If you want to connect two crossing wires, use a solder dot, else they are assumed to be bridging and not connected. The size of the dot can be set from the Display Options dialog.

2.6.42 Create->Label...

The **Create->Label...** command displays the Create Label dialog.

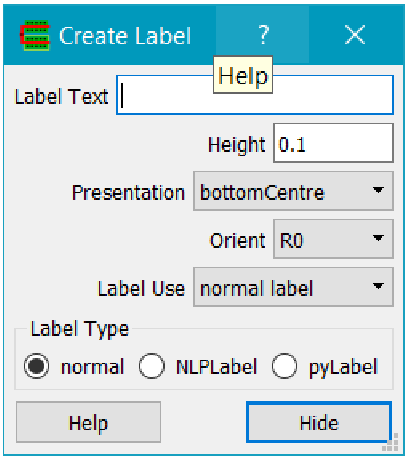


Figure 117 - Create Label

To label a wire with its net name, *Label Text* specifies the name of the label, along with *Height*, *Orientation*, and *Presentation*. The *Label Use* should be 'normal label' label, and the *Label Type* 'normal'.

2.6.43 Create->Pin...

The **Create->Pin...** command displays the Create Pin dialog.

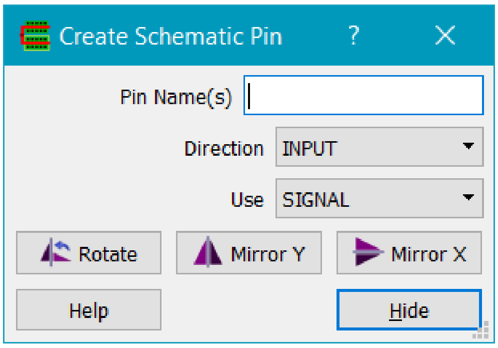


Figure 118 - Create Schematic Pin

A list of Pin Name(s) can be entered, separated by spaces. As each pin is positioned by left clicking, a pin of the first name in the pin name list is created, and that name is removed from the list of pin names. The pin Direction and pin Use can also be specified. Pins can be mirrored or rotated during entry. A pin is actually an instance of a pin from the 'basic' library; if this library cannot be opened when Glade starts an error will be reported and Create Pin will fail.

2.6.44 Create->Symbol

The **Create->Symbol** command displays the Create Symbol dialog.

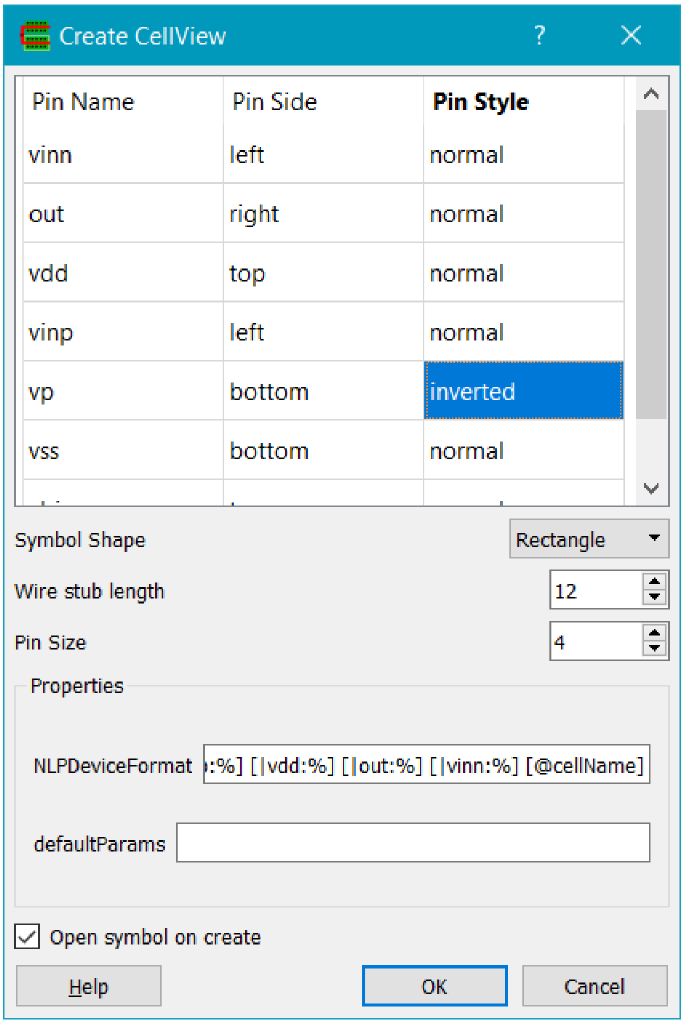


Figure 119 - Create CellView

This command creates a symbol view from the existing schematic. *Symbol Shape* sets the shape of the created symbol; valid options are Rectangle, Triangle and Circle. *Pins* shows the pins of the symbol, derived from the schematic pins. The sides of the symbol the pins are placed on are give by the *Left*/*Bottom*/*Right*/*Top* fields and consist of the pin names, delimited by spaces. Wire stub length is the length (in dbu) of the wires from the symbol body to the pins. Pin Size is the size of the pin rectangles in sbu and defaults to the same as the dot size used in schematics. *NLPDeviceFormat* is the property that is added to the symbol to control netlisting.

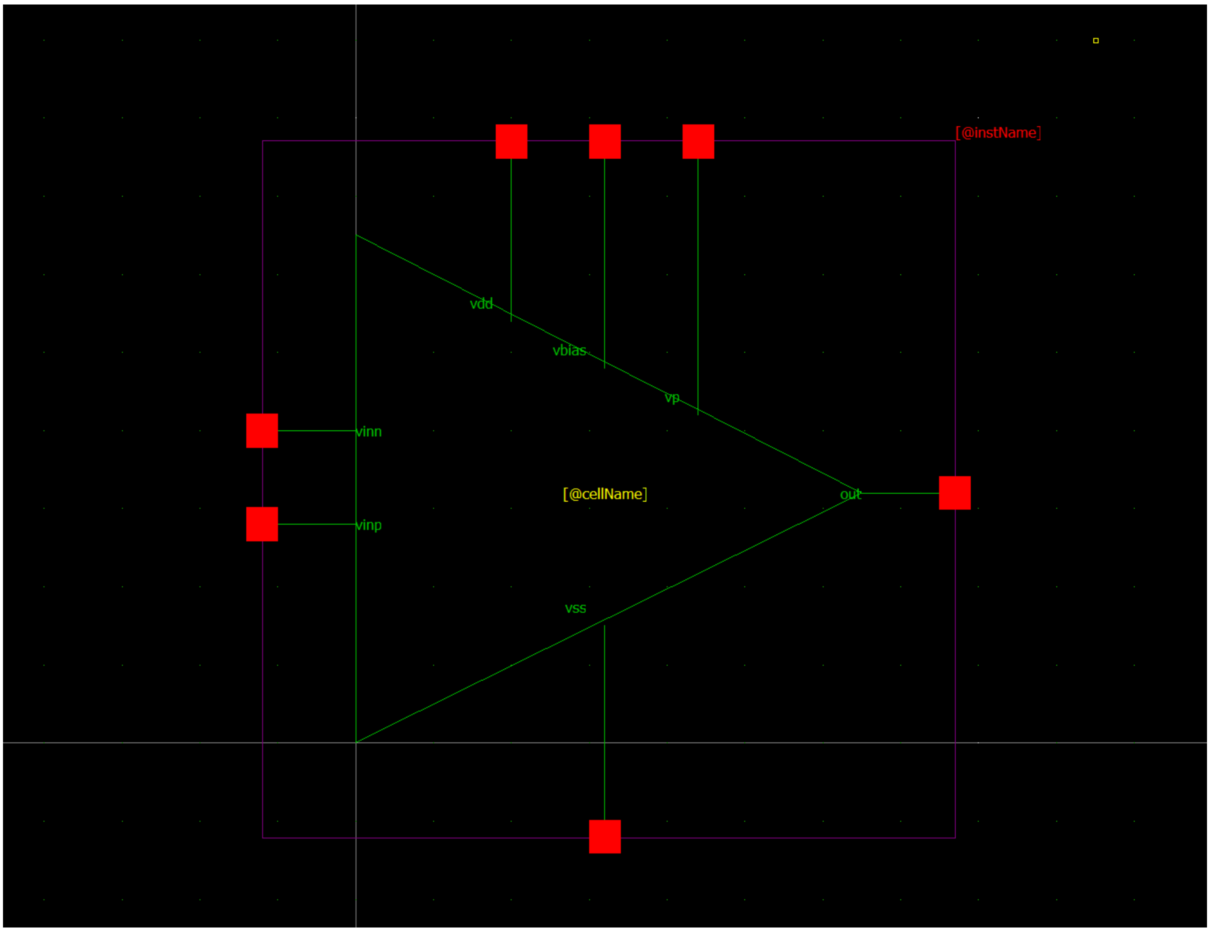


Figure 120 - Creating a symbol from a schematic cellView

2.6.45 Check

2.6.46 Check->Check CellView

The Check CellView command must be used after creating or editing a schematic to extract connectivity e.g. for netlisting. Various checks are performed including floating wires, floating pins and shorted wires, and the checks can be controlled using the Check Options dialog. Bus connections are checked for width and syntax. If errors are found, the number is reported and markers are written on the marker layer to the cellView.

2.6.47 Check->View Errors…

The Check->View Errors… command displays the schematic error viewing dialog.

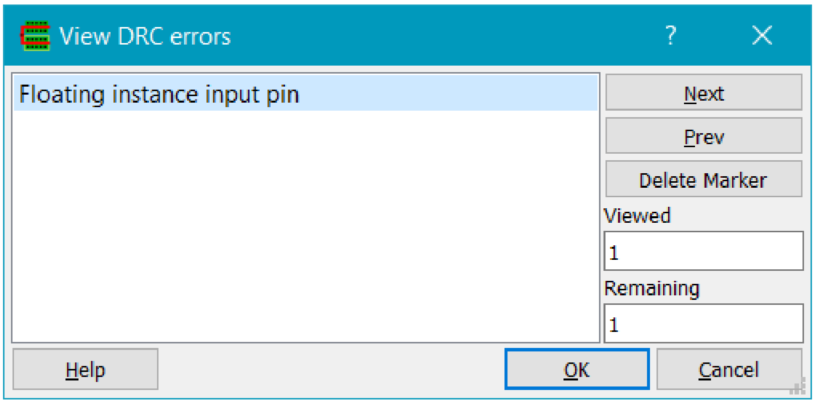


Figure 121 - View Errors

Errors are listed in the left hand panel; click on an error type to view the associated errors. Errors can be stepped through via the *Next* and *Prev* buttons. *Delete* will delete an error marker.

2.6.48 Check->Clear Errors

The **Check->Clear Errors** command clears all error markers.

2.6.49 Check->Check Options…

The **Check->Check Options…** command displays the Check Options dialog.

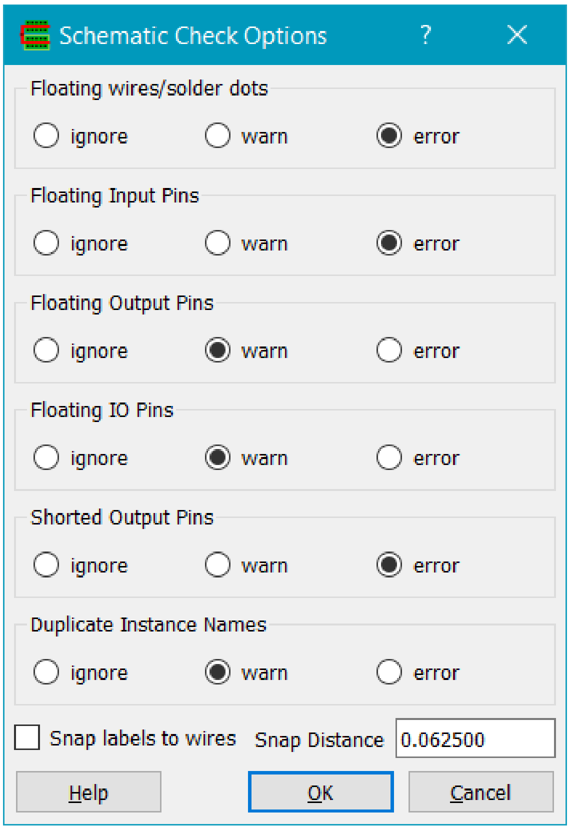


Figure 122 - Check Options

Checks can be set to be ignored, to give warnings, or to give errors. If errors occur, then the schematic cannot be netlisted until they are corrected and cleanly checked. *Snap labels to wires* will snap label origins onto wires, if they are closer than *Snap Distance*. This is useful for e.g. import EDIF where labels on schematics may not be positioned accurately due to grid issues.

2.6.50 Layout

The Layout menu commands facilitate generating layout from a schematic view.

2.6.51 Layout->Map Devices

The **Layout->Map Devices** command allows mapping a cell in the schematic to a different named cell (usually PCell) in the layout.

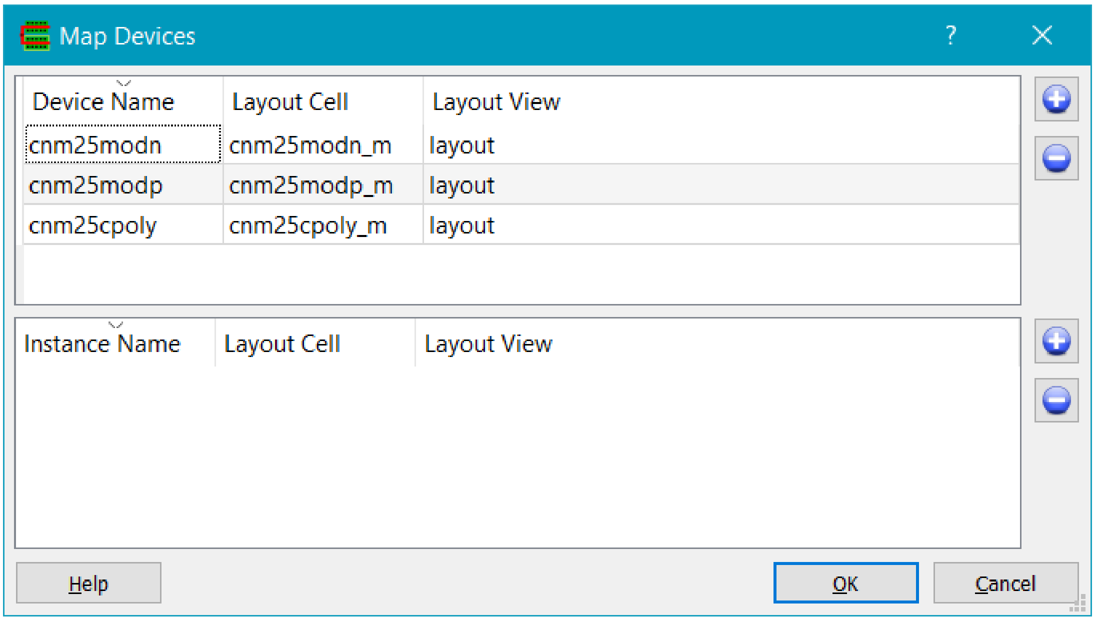


Figure 123 - Map Devices

In the above dialog, the entries in the Device Name panel of the table map a cell name such as cnm25modn in the schematic to a cell called cnm25modn_m in the layout. Entries in the Instance Name panel can map specific instances of a cell to a different layout cell.

Device mapping can be set up to pre-seed the dialog using entries in the Glade technology file:

```
MAP cnm25modn TO cnm25modn_m layout ;
MAP cnm25modp TO cnm25modp_m layout ;
MAP cnm25cpoly TO cnm25cpoly_m layout ;
```

2.6.52 Layout->Gen Layout

To create a layout view from a schematic, use the Create Layout command.

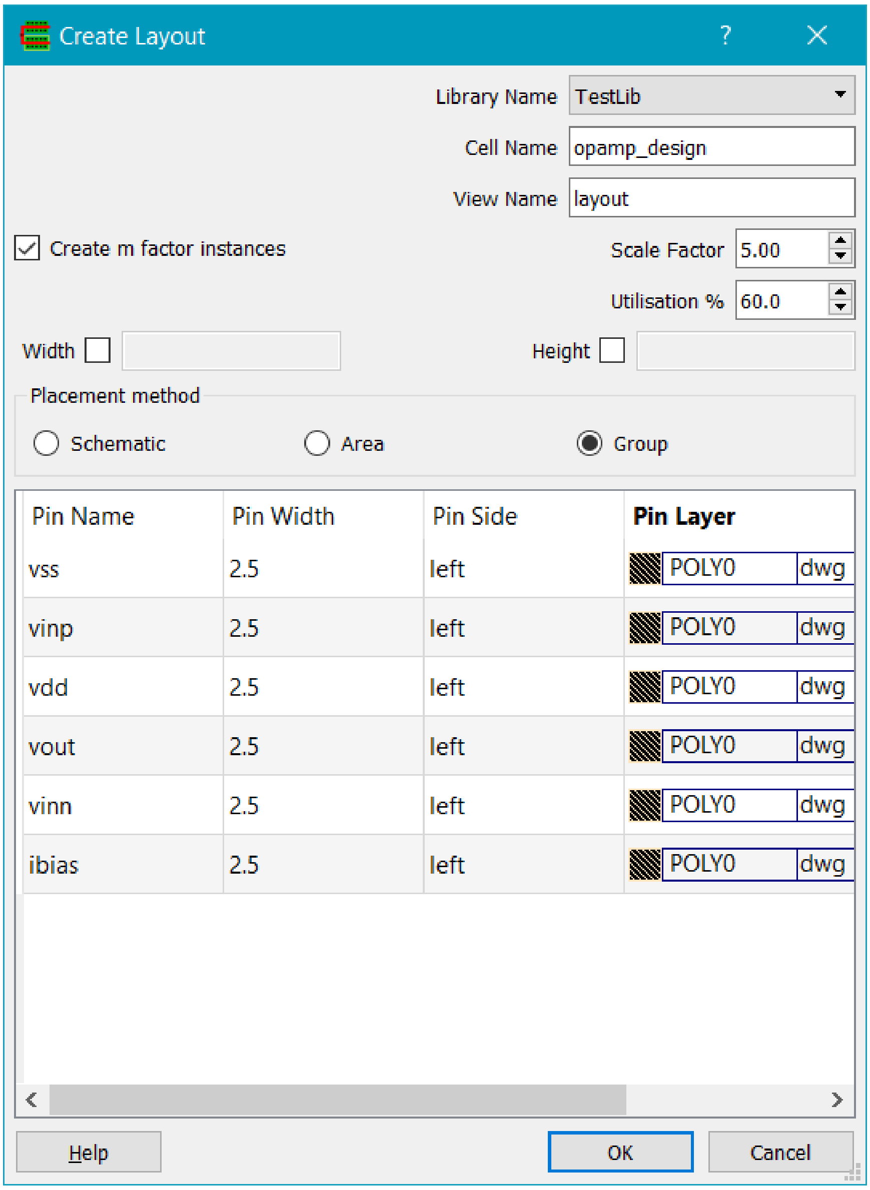


Figure 124 - Create Layout

The target cellView is specified using the Library Name / Cell Name / View Name fields. If *Create m factor instances* is set, then if a schematic instance has an integer property 'm', then multiple instances of the cell will be created in the layout based on the value of the property, and the m property is not passed to the layout PCell. If not checked, the m property is passed to the layout PCell, if the PCell is required to handle this itself.

Scale Factor is used when the *Placement method* is *Schematic*. It scales the instance origin coordinates by the factor, so the resulting layout mimics the schematic. The actual value required will depend on the target library cells.

Utilisation is used to create the cell boundary layer in the resulting layout view. The area of all the layout instances is summed, and divided by 100/utilisation%. If *Width* is specified, the cell boundary will be rectangular with the specified width, and height will be computed from the area/width. If *Height* is specified, the cell boundary rectangle will have the specified height and the width will be computed from the area/height. If both *Width* and *Height* are specified, then the cell boundary rectangle will use the specified width and height.

Placement method can be one of *Schematic*, *Area* or *Group*. *Schematic* placement uses the relative coordinates of the schematic instance origins to place the layout cells. *Area* arranges the layout cells by type (PMOS/NMOS/resistor/capacitor). *Group* will place cells according to group properties on the schematic, id they have been specified, or place by schematic for those that have no group properties.

The pin field allows pin width, side and layer to be specified for each pin. Pins are placed abutting the cell boundary rectangle according to their side.

2.6.53 Layout->Create Group

The **Create->Create Group** command displays the Create Group dialog.

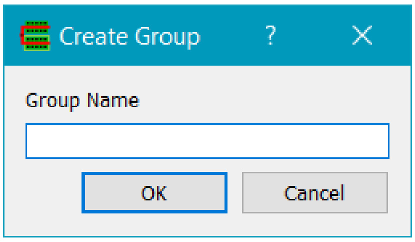


Figure 125 - Create Group

Group Name specifies the name of the group. The command takes a selected set of instances and creates a group for group placement in Gen Layout. A string property with name "group" and value given by the group name will be created on all the selected instances.

2.6.54 Layout->Add To Group

The ***Create->Add To Group*** command displays the Add to Group dialog.

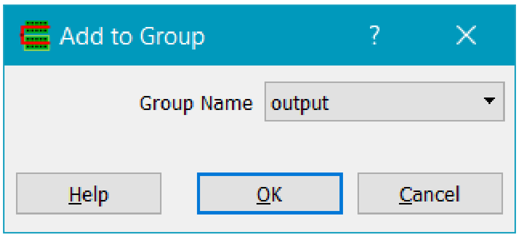


Figure 126 - Add To Group

The selected instances are added to the group specified by the *Group Name* field.

2.6.55 Layout->Rename Group

The **Create->Rename Group** command renames an existing group.

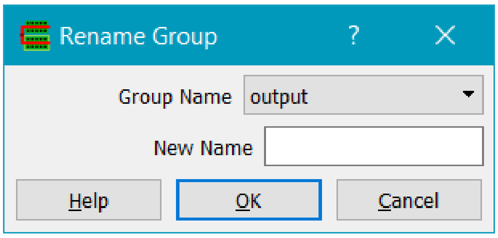


Figure 127 - Rename Group

Group Name is the name of the group to rename. *New name* is the new name for the group.

2.6.56 Layout->Remove From Group

The **Layout->Remove From Group** command removes the selected instances from the group. This command cannot be undone.

2.6.57 Layout->Delete Group

The **Layout->Delete Group** command displays the Delete Group dialog.

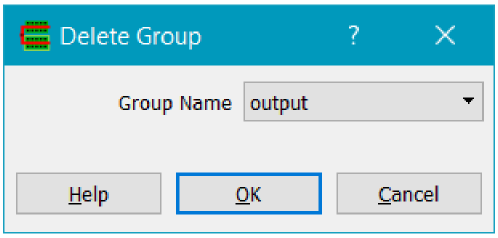


Figure 128 - Delete Group

The group specified by Group Name is deleted.

2.6.58 Layout->Edit Group

The **Layout->Edit Group** command allows setting the pattern for the layout of the group's instances.

Figure 129 - Edit Group

The instances of the group specified by *Group Name* are displayed as a grid, with different instance basenames in different colours (an instance basename is the name as seen on the schematic e.g. M6; the full name e.g. M6.0 consists of the basename and optionally the individual instances expanded by m-factor as <basename>.0, <basename>.1 etc). The *Rows* and *Cols* spinboxes can be used to change the generated array of devices; the size of the array is always greater than the number of actual instances. *Dummy Name* is the name of the cellView to use for adding dummy cells. *Group Name* sets the current group to edit. To change positions of instances, left click and drag an instance to a new position; the source and destination instances are swapped.

For groups with more than one instance basename, the centre of gravity of the instances are shown by a circle, coloured with the instance colour. If all instances centres of gravity coincide, then the layout pattern is shown as being common centroid with the label in green. If not, the label indicator is red.

Right clicking on an instance displays a context menu with options to mirror or rotate the device and to add dummies before or after the current selected instance, and to delete a currently selected dummy. Dummies are given the prefix IDMY to the instance name, followed by a period and a number which is incremented for each dummy that is added. Dummy cells are generated as instances of cells with the cell master specified by Dummy Name. Dummies are not (yet) backannotated to the schematic and are not assigned connectivity. Instance orientations are shown by the tab triangle which is in the top right for orientation R0, top left for MY etc.

Rows and *Cols* set the size of the group; if *Decouple Row/Col count* is not checked, as one is altered the other is also so that the overall cell count is approximately maintained. If *Decouple Row/Col count* is checked, then the number of rows and columns can be altered independently, however the number cannot be reduced below that which would give a total number of grid entries less than the number in the group. *Group Orientation* is a global orientation of the group and takes effect as a transformation of all instances after any instance-specific mirroring. If *Autoinsert Dummies* is checked, then dummy instances with type set by *Dummy Name* are added to the grid as it is resized.

The Left/Right/Up/Down arrows allow scrolling of the grid pattern. This allows e.g. adding a ring of dummies easily.

The group patterns are saved to the schematic cellView as a property with name equal to the group name. The value of this property is a string of the form "I0.0_0_0_0,I0.1_0_1_6" etc. where each field delimited by a comma represents the instance name, the row number and the column number, finally the orientation as a digit, delimited by an underscore.

2.6.59 Layout->Link To Layout

The Layout->Link to Layout command sets the mapping from schematic to layout. If you have two windows open in MDI mode, one for the schematic and one for the layout, this allows cross probing between layout instances and schematic instances. The corresponding instances are selected in the linked cellView, and are highlighted. Note that layout linking is automatically carried out when Gen Layout is run.

2.6.60 Layout->Clear Hilite

The **Layout->Clear Hilite** command clears any currently highlighted devices.

2.7 Symbol Menus

2.7.1 View

2.7.2 View->Fit

See Layout View menu

2.7.3 View->Fit+

See Layout View menu

2.7.4 View->Zoom In

See Layout View menu

2.7.5 View->Zoom Out

See Layout View menu

2.7.6 View->Zoom Selected

See Layout View menu

2.7.7 View->Pan
See Layout View menu

2.7.8 View->Redraw
See Layout View menu

2.7.9 View->Ruler
See Layout View menu

2.7.10 View->Delete Rulers
See Layout View menu

2.7.11 View->Cancel Redraw
See Layout View menu

2.7.12 View->Display Options…
See Schematic View menu

2.7.13 View->Selection Options…
See Schematic View menu

2.7.14 View->Pan/Zoom Options…
See Schematic View menu

2.7.15 Edit

2.7.16 Edit->Undo
See Schematic Edit menu

2.7.17 Edit->Redo
See Schematic Edit menu

2.7.18 Edit->Yank
See Layout View menu

2.7.19 Edit->Paste
See Layout View menu

2.7.20 Edit->Delete
See Schematic Edit menu

2.7.21 Edit->Copy
See Schematic Edit menu

2.7.22 Edit->Move
See Schematic Edit menu

2.7.23 Edit->Move By…
See Schematic Edit menu

2.7.24 Edit->Move Origin
See Schematic Edit menu

2.7.25 Edit->Stretch
See Schematic Edit menu

2.7.26 Edit->Rotate…
See Schematic Edit menu

2.7.27 Edit->Set Net…
See Schematic Edit menu

2.7.28 Edit->Select->Select All
See Schematic Edit menu

2.7.29 Edit->Select->Deselect All
See Schematic Edit menu

2.7.30 Edit->Properties->Query
See Schematic Edit menu

2.7.31 Edit->Properties->Query CellView
See Schematic Edit menu

2.7.32 Edit->Search…
See Schematic Edit menu

2.7.33 Edit->Edit Bindkeys…
See Schematic Edit menu

2.7.34 Create
Symbols require shapes on the 'device' layer to represent their structure, for example a zigzag line for a resistor. Symbols have pins to allow connectivity when placed in a schematic. Finally symbols have labels to display information such as instance name, model name etc.

2.7.35 Create->Create Line…
The **Create->Create Line…** command creates a line object.

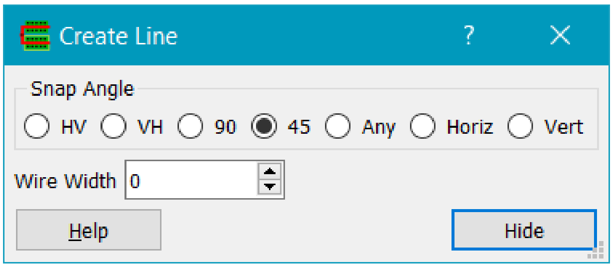


Figure 130 - Create Line

Lines are created on the device layer. In Infix mode, the first point is the current position of the cursor, else the first and subsequent points are prompted for. The backspace key can be used to delete the last entered point. Pressing Enter or left double clicking terminates a Create Line command. *Snap Angle* controls the entry mode. *Wire Width* sets the width of the line.

2.7.36 Create->Create Rectangle

The **Create->Create Rectangle** command creates a rectangle on the device layer.

2.7.37 Create->Create Polygon...

The **Create->Create Polygon...** command creates a polygon on the device layer.

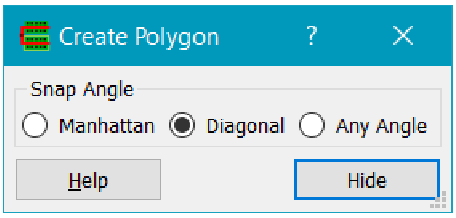


Figure 131 - Create Polygon

In Infix mode, the first point is the current position of the cursor, else the first and subsequent points are prompted for. The backspace key can be used to delete the last entered point. Pressing Enter or left double clicking terminates a Create Line command. *Snap Angle* controls the entry mode.

2.7.38 Create->Create Circle...

The **Create->Create Circle...** command creates a circle on the device layer.

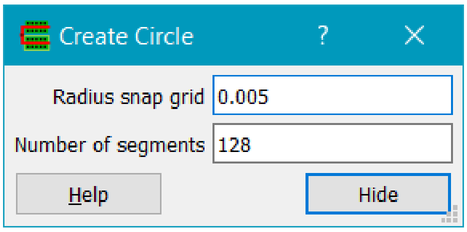


Figure 132 - Create Circle

In Infix mode, the current position of the cursor is used for the centre of the circle, else a point is prompted for. The second point is s point on the circumference of the circle. *Radius snap grid* is the snap grid of points on the circumference. *Number of segments* is the number of line segments used to represent the circle.

2.7.39 Create->Create Ellipse...

The **Create->Create Ellipse...** command creates a circle on the device layer.

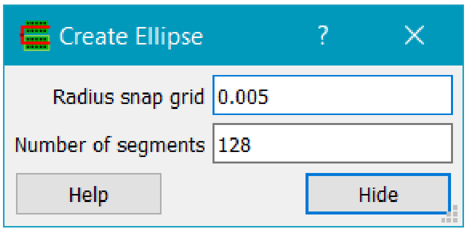


Figure 133 - Create Ellipse

In Infix mode, the current position of the cursor is used for the centre of the ellipse, else a point is prompted for. The second point is s point on the circumference of the ellipse. *Radius snap grid* is the snap grid of points on the circumference. *Number of segments* is the number of line segments used to represent the ellipse.

2.7.40 Create->Create Arc...

The **Create->Create Arc...** command creates an arc on the device layer. Arcs are entered using three left mouse clicks, or two clicks if Infix mode is set. The first click defines the starting reference of the arc. The second point defines the stopping reference. The third point defines the radius of the arc relative to the centre of a line between the first and second points.

2.7.41 Create->Create->Label...

The **Create->Create Label...** command creates a label.

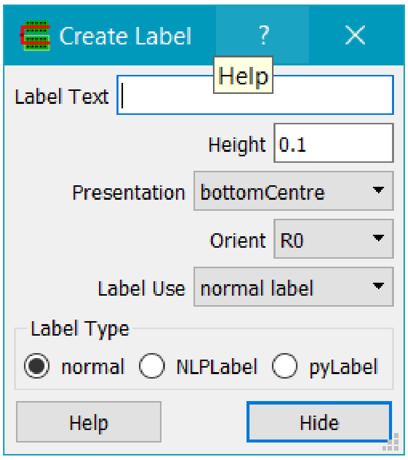


Figure 134 - Create Symbol Label

Label Text is the label text string. Label Use sets the use mode of the label:

- *normal label*: Can be used to represent general textual information, the current layer is used.
- *instance label* : These are labels typically of the form [@instName] using NLP parser syntax. They are created on the 'annotate' 'drawing7' layer/purpose. They are of type 'NLPLabel'.
- *pin label*: These are labels typically of the form [@pinName] using NLP parser syntax. They are created on the 'annotate' 'drawing8' layer/purpose. They are of type 'NLPLabel'.

- *device label*: These are labels e.g. [@l:l=%:l=0.13u] using NLP parser syntax. They are created on the 'annotate' 'drawing' layer/purpose. They are of type 'NLPLabel'.
- *device annotate*: These are labels typically of the form [@cellName] using NLP parser syntax. They are created on the 'annotate' 'drawing4' layer/purpose. They are of type 'NLPLabel'.

Label Type is the type of the label:

- *normal* is a simple text string.
- *NLPLabel* is a label that will be interpreted according to NLP expression syntax.
- *pyLabel* is a label whose text will be evaluated by the Python interpreter.

2.7.42 Create->Create Pin...

The Create->Create Pin... command is used to create symbol pins.

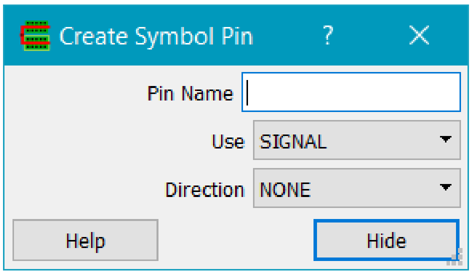


Figure 135 - Create Pin

Pin Name is the name of the pin. Use is the pin type. Direction is the pin direction and sets the pin shape used. It can be None (square pin), Input, Output (directional pin) or Inout (bidirectional pin). A pin is actually an instance, and pin instances use masters from the ‘basic’ library.

2.7.43 Check

2.7.44 Check->Check

The **Check->Check** command checks the symbol view. It cleans up the symbol connectivity by deleting any non-shape nets, deleting non-shape pins and setting any shapes with net info to be pins.

2.8 Floorplan Menus

2.8.1 View

See the Layout View menus

2.8.2 Edit

See the Layout Edit menus

2.8.3 Create

See the Layout Create menus

2.8.4 Verify

See the Layout Verify menus

2.8.5 Floorplan

2.8.6 Floorplan->Initialise Floorplan

The **Floorplan->Initialise Floorplan**… command (Re)Initialises the floorplan.

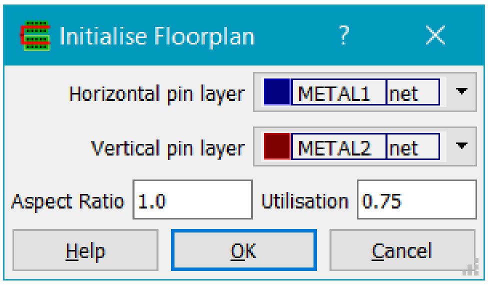


Figure 136 - Initialise Floorplan

The *Horizontal pin layer* and *Vertical pin layer* fields set the layer for pins created on the top/bottom and left/right edges of the design boundary, respectively. *Aspect ratio* is the block aspect ratio, with numbers greater than 1 representing tall blocks and numbers less than 1 giving wide blocks. *Utilisation* is the desired cell utilisation, i.e. the ratio of total cell area to design boundary area.

2.8.7 Floorplan->Create Rows...

The **Floorplan->Create Rows…** command creates rows for use with Place & Route.

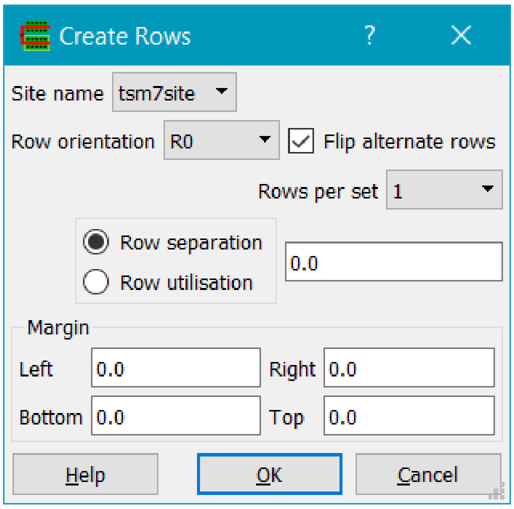


Figure 137 - Create Rows

The design must have standard cells and a valid design boundary (a rectangle on the boundary layer). A valid *Site name* must exist, which will normally be found automatically from the library is a cell exists with the boolean property 'site'. *Row orientation* is the desired row orientation - R0, R180, MX and MY create horizontal rows wheras the rest create vertical rows. *Flip alternate rows* if checked will flip the orientation of adjacent rows to provide power track sharing in libraries that support it. *Rows per set* can be either 1 or 2 and specifies the number of rows placed together before any row spacing is applied. *Row separation* is the separation between sets of rows, in microns. Alternatively *Row utilisation* can be give as a percentage. *Margin* specifies a margin around the rows, which are created inside the design boundary (boundary layer).

2.8.8 Floorplan->Create Groups...

The **Floorplan->Create Groups...** command creates and/or edits groups for use with Place & Route.

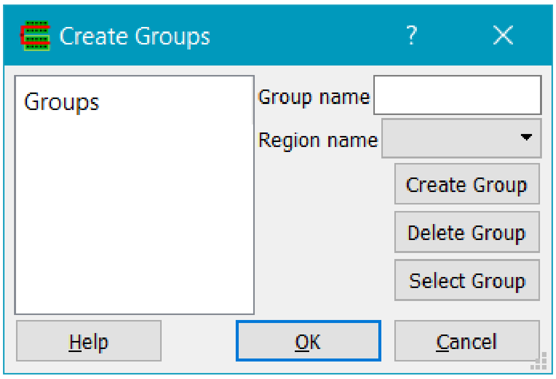


Figure 138 - Create Groups

Any existing groups are shown in the *Groups* list box on the left; clicking on a group in the list box updates the *Group name* field, and also the *Region name* field if there is a region associated with the group. Groups can be created using *Create Group*; instances must be selected first and a *Group name* must be specified (or be an existing group name). *Delete Group* will delete a group specified by the Group name. *Select Group* will select all instances of the group in the *Group name* field.

2.8.9 Floorplan->Create Region...

The **Floorplan->Create Region...** command creates a region for use with Place & Route.

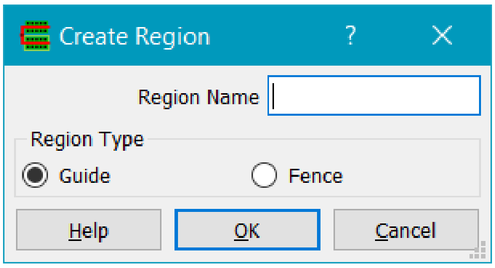


Figure 139 - Create Region

A region is a rectangle on the Region layer with string properties type and name. Regions can be of type *Guide* or *Fence*. *Fence* regions are hard constraints whereas *Guide* regions are soft constraints.

2.8.10 Floorplan->Placement->Place

The **Floorplan->Placement->Place** command places cells in rows using the UCLA Capo placer.

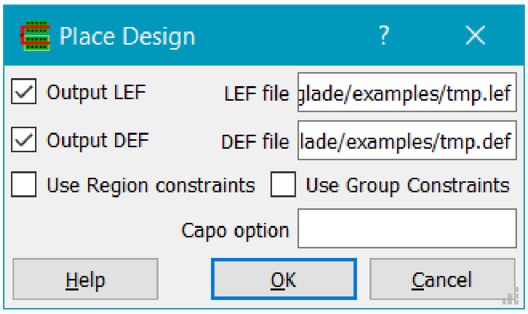


Figure 140 - Place Design

Glade exports LEF and DEF from the current design and invokes Capo; on completion, Glade reads in the placed DEF.

The design must have rows for placement of cells, as defined using the Row layer, and a valid design area as defined by the boundary layer. Placement regions may exist as defined by the Region layer.

Currently only horizontal cell rows are supported for placement - this is a limitation of the Capo code. Rows can be flipped and cell orientations will obey row orientations (this is a bugfix from the distribution Capo code which assumes all rows are N orientation)

2.8.11 Floorplan->Placement->Unplace

The **Floorplan->Placement->Unplace** command unplaces all standard cells. Cells are moved to the right of the design area and have their orientation set to R0 and placement status set to unplaced.

2.8.12 Floorplan->Global Route->Global Route

The **Floorplan->Global Route->Global Route** command runs global routing on the design.

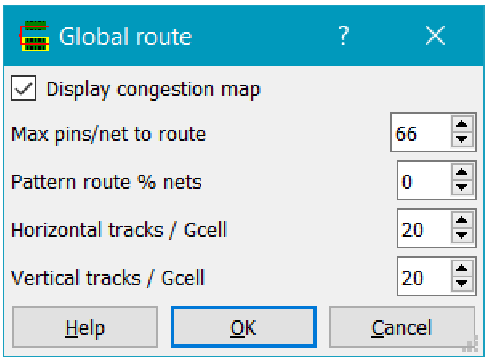


Figure 141 - Global Route

The design must be a placed standard cell or block design, imported using LEF/DEF or LEF/Verilog. Global routing partitions the design into bins also known as gcells. The size of the gcell has a direct impact on the speed and accuracy of the global routing: smaller gcells give a more accurate picture of the congestion but at the expense of speed. Max pins/net to route limits global routing to nets with less than the limit. Pattern route routes small two pin nets first using a L-shaped pattern. This is much faster than routing nets using the full maze router.

The congestion map displayed shows the gcell grid and the edge congestion in a colourmap. Edges that are blue have 0 more tracks available (supply) to route on than are required (demand). Cyan edges have a demand of 1, green edges have a demand of 2, yellow 3, red 4, purple 5, white greater than 5.

Currently the global router is single-layer i.e. all layers are compressed into one. This gives a good idea of congestion but no layer by layer congestion map. This is an area for future enhancement, as is accurately modelling obstructions.

2.8.13 Floorplan->Global Route->Show global routed net

The **Floorplan->Global Route->Show global routed net** command displays the path the global router took for the user-specified net.

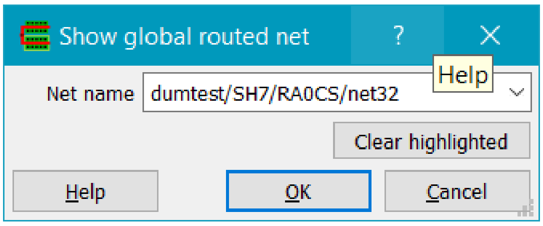


Figure 142 - Show global route

If a net is not displayed, either it has more pins than the max pins/net limit, or the net starts and ends within the gcell.

2.8.14 Floorplan->Global Route->Toggle congestion map display

The **Floorplan->Global Route->Toggle congestion map display** command toggles the display of the congestion map.

2.8.15 Floorplan->Placement->Check Overlaps

The **Floorplan->Placement->Check Overlaps** command checks for any overlapping standard cells in the design, reporting their names and locations if found

2.8.16 Floorplan->Fillers->Add...

The **Floorplan->Fillers->Add...** command adds filler cells.

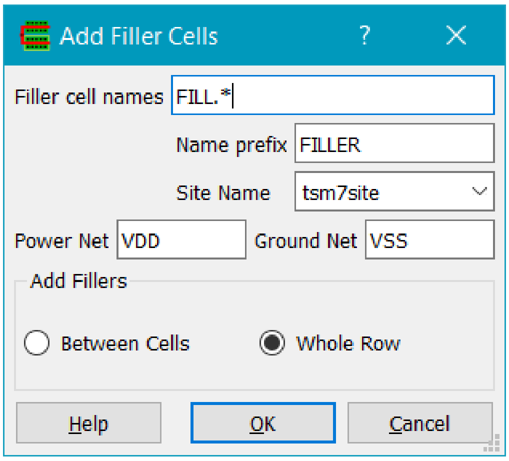


Figure 143 - Add Fillers

Rows must be present in order to add filler cells. The filler cells are specified by the filler name pattern given (e.g. FILL*). The instance names of fillers are prepended by the given name prefix and a count, e.g. FILLER1234. The site name needs to be specified from the list of site names. Power Net and Ground Net are the names of the power and ground nets that the filler cells should be connected to. Fillers can be added either for the whole row, or just between cells.

2.8.17 Floorplan->Fillers->Delete...

The **Floorplan->Fillers->Delete...** command deletes cells matching the pattern.

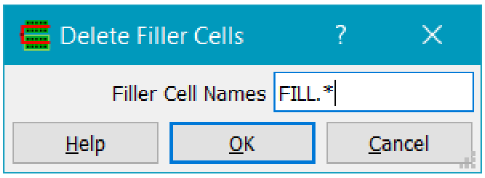


Figure 144 - Delete Fillers

Note that this can be used to delete any cells matching the pattern, not just filler cells.

2.8.18 Floorplan->Replace Views...

The **Floorplan->Replace Views...** command replaces instances with masters of one view type with masters of another view type.

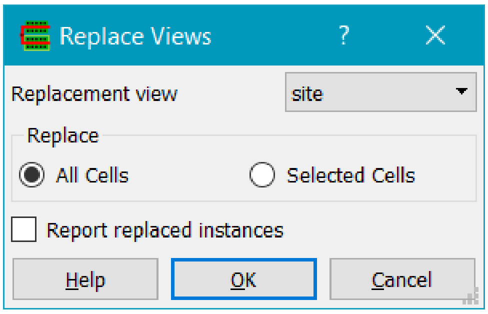


Figure 145 - Replace Views

Replacement view is the view to replace the existing master's view with. *All Cells* will replace all instances in the current cellView. *Selected Cells* will replace just the instances in the selected set. *Report replaced instances* will report to the logfile the instances changed.

2.8.19 Floorplan->highlightNetTypes...

The **Floorplan->Highlight Net Types...** command highlights DEF nets by type i.e. SIGNAL, ANALOG, CLOCK, POWER, GROUND, RESET, SCAN, TIEOFF.

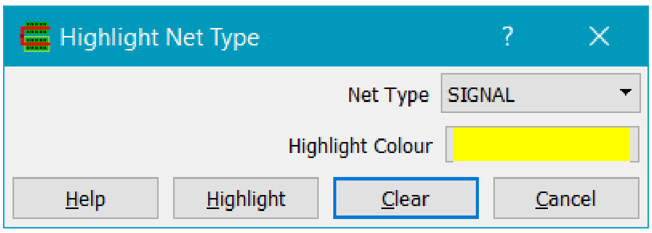


Figure 146 - Highlight Net Types

3 Verification

3.1 Layer Processing

Before using DRC or LPE (extraction) commands, you normally need to perform some layer processing using boolean operations or select operations. For example in order to extract a MOS transistor, we need to identify the gate area by using the geomAnd() of the poly and diffusion layers, and we need to use the geomAndNot() of the diffusion and poly layers to split the diffusion between the source and the drain of the device, else we end up with all devices S/D terminals shorted.

The following layer processing functions are supported. Note that these don't have to be used just for DRC or LPE - you can use them in any python script for layer processing. The 'layers' that the commands produce are in fact temporary binary edge files. These files are called file0000.dat, file0001.dat etc. and are automatically deleted during geomEnd(). By default the layer files are written in the directory that Glade is invoked in. However if the environment variable GLADE_DRC_WORK_DIR is set to a valid directory, then they will be written to that directory instead.

Optional arguments are shown as e.g. hier=True, indicating that the default value of True is used if the argument is not specified.

3.1.1 geomBegin(cellView *cv)

Initialise the DRC package. A valid cellView pointer must be passed to initialise the package. The cellView will be the one that subsequent processing operates on. Note the former equivalent function drcInit(cv) is still supported, but deprecated.

3.1.2 geomEnd()

Uninitialise the DRC package. Working memory is freed. Temporary layer files are deleted. Note the former equivalent function drcUnInit(cv) is still supported, but deprecated.

3.1.3 out_layer = geomGetShapes("layerName", purpose = "drawing", hier=True)

Initialises out_layer with all shapes on the layer layerName, with purpose purpose. purpose defaults to "drawing" if not given. The resulting derived out_layer contains merged shapes. The default is to get all shapes through the hierarchy; if the optional parameter hier is False, then only top level shapes are processed.

3.1.4 out_layer = geomAddShape(layer, shape *shape)

Adds a shape *shape* to the edge layer *layer*. The resulting *out_layer* is merged. For example:

```
y4 = geomEmpty()
cutshape = cv.dbCreateRect(cut, y4_lyr)
y4 = geomAddShape(y4, cutshape)
```

3.1.5 out_layer = geomAddShape(layer, shapes)

Adds a python list of shapes to the edge layer *layer*. The resulting *out_layer* is merged. For example:

```
y3 = geomEmpty()
shapes = []
for i in range(0,4) :
        shape = cv.dbCreateRect(box, y3_lyr)
```

```
        box.offset(2000, 0)
        shapes.append(shape)
    y3 = geomAddShapes(y3, shapes)
```

3.1.6 geomNumShapes(layer)

Returns the number of shapes in a layer. This can be used as a test, e.g.

```
if geomNumShapes(diff) != 0 :
    gate = geomAnd(poly, diff)
```

3.1.7 geomEmpty()

Returns a dummy empty out_layer .

3.1.8 geomBkgnd(size = 0.0)

Returns a layer with an extent the size of the cellView's bounding box, plus *size* (which defaults to 0.0um). This is useful for example to create a pwell layer when the original mask data just has nwell information:

```
nwell = geomGetShapes("nwell", "drawing")
bkgnd = geomBkgnd()
psub = geomAndNot(bkgnd, nwell)
```

3.1.9 geomErase(layerName, purpose="drawing")

Erases any design data on layer *layerName* in the current cellView. *purpose* defaults to "drawing" if not given. Beware: there is no way of undoing this operation.

3.1.10 geomMerge(layer)

Returns the merged shapes on layer. This is equivalent to a single layer OR. Note that geomGetShapes() always merges raw input data, so there is normally no need to separately merge layers.

3.1.11 geomOr(layer1, layer2)

Returns the OR (union) of the two layers.

3.1.12 geomAnd(layer1, layer2)

Returns the AND (intersection) of the two layers.

3.1.13 geomNot(layer)

Returns the inverse of the layer. Effectively it runs geomAndNot(), with the first 'layer' being a rectangle the size of the cellView's bounding box, and the second the specified layer.

3.1.14 geomAndNot(layer1, layer2)

Returns the AND NOT of layer1 with layer2. This is equivalent to subtracting all shapes on layer2 from layer1.

3.1.15 geomXor(layer1, layer2)

Returns the XOR of the two layers.

3.1.16 geomSize(layer, size, flag = 0)

Returns the layer sized by *size* microns. A positive size grows the layer, while a negative size shrinks the layer. If a shape should shrink so its width becomes zero, it will no longer be present in the sized_layer . The third argument, *flag*, if not specified sizes all edges by size. If flag is set to 'vertical' then sizing is only done in the vertical direction, if flag is set to 'horizontal' then sizing is only done in the horizontal direction.

3.1.17 geomTrapezoid(layer)

Returns the layer converted to trapezoids. If layer has connectivity established via geomConnect(), the connectivity will be maintained in the trapezoids generated.

3.1.18 geomTouching(layer1, layer2)

Select and return all shapes on layer2 that touch layer1. Touching is defined as any edge of layer2 polygons that touch an edge from a layer1 polygon.

3.1.19 geomOverlapping(layer1, layer2)

Select and return all shapes on layer2 that touch layer1. Overlapping is defined as any edge of layer2 polygons that intersects an edge from a layer1 polygon, i.e. the layer2 polygon is part inside, part outside layer1.

3.1.20 geomInside(layer1, layer2)

Select and return all shapes on layer2 that are inside (completely enclosed by) layer1. To make sure that shapes on layer2 are completely 'inside' layer1, the 'enclosing' shape should be layer1.

3.1.21 geomContains(layer1, layer2)

Select and return all shapes on layer2 that enclose shapes on layer1.

3.1.22 geomOutside(layer1, layer2)

Select and return all shapes on layer2 that are outside layer1. The shapes may touch or abut layer1. To make sure that shapes on layer2 are completely 'outside' layer1, and get considered, the 'enclosing' shape should be layer1.

3.1.23 geomAvoiding(layer1, layer2)

Select and return all shapes on layer2 that avoid layer1 and do not touch or overlap.

3.1.24 geomButting(layer1, layer2)

Select and return all shapes on layer2 that have outside edges that abut layer1 outside edges.

3.1.25 geomCoincident(layer1, layer2)

Select and return all shapes on layer2 that have edges coincident with inside edges of layer1.

3.1.26 geomGetTexted(layer, layerName, purpose = "drawing", name=NULL)

Returns all shapes on layer that have text labels on the layer/purpose pair given by layerName / purpose. purpose defaults to drawing if not given. Optionally a name can be supplied e.g. "vdd", and then only shapes with text labels with that name are output to out_layer .

3.1.27 geomHoles(layer)

Returns the holes in polygons on layer and outputs them to out_layer.

3.1.28 out_layer = geomNoHoles(layer)
Returns the polygons on layer and outputs them, minus any holes, to out_layer.

3.1.29 geomGetNon90(layer)
Returns the polygons on layer and outputs ones that have one or more edges that are non 90 degrees to out_layer.

3.1.30 geomGetNon45(layer)
Returns the polygons on layer and outputs ones that have one or more edges that are non 90 and non 45 degrees to out_layer

3.1.31 geomGetRectangles(layer)
Returns the shapes on layer and outputs ones that are rectangular, i.e. have 4 edges parallel to the X or Y axes.

3.1.32 geomGetPolygons(layer)
Returns the shapes on layer and outputs ones that are not rectangular, i.e. have at least one edge not parallel to the X or Y axes.

3.2 DRC
DRC functions, like boolean operations, are edge-based, using a Bentley-Ottman scanline algorithm. Edges are currently only considered in error if they project onto each other in the X or Y direction and the spacing between the edges is less than the specified rule. Perpendicular edges are not considered errors. The resulting error marker shapes are constructed from the four vertices of the error edges, and are returned as an output layer where they may be used for subsequent boolean operations.

3.2.1 Flags
Many commands take a flags parameter. Flags can be bitwise OR's together using the '|' operator, although some flags are mutually independent, e.g. equal/not_equal/greater, or samenet/diffnet, or horizontal/vertical/diagonal. Flags include:

- **none** - No flag bits set.
- **samenet** - Used for geomSpace checks, check only carried out if shapes are on the same net.
- **diffnet** - Used for geomSpace checks, check only carried out if shapes are on different nets.
- **vertical** - Only vertical edges are checked.
- **horizontal** - Only horizontal edges are checked.
- **diagonal** - Only diagonal (45 degree) edges are checked.
- **project** - The checked edges must project, i.e. be parallel and share a common parallel runlength.
- **parallel** - The checked edges must be parallel.
- **abut** - Used for geomSpace and geomEnclose checks. If set, abutting edges will flag the spacing check, else abutting is allowed.
- **equal** - If set, violations are created where the test result is equal to the rule, e.g. geomWidth(M1, 0.030, equal) will result in shapes with width equal to 30nm selected for error/output.

- **not_equal** - If set, violations are created where the test result is not equal to the rule, e.g. geomWidth(M1, 0.030, not_equal) will result in shapes with width not equal to 30nm selected for error/output.
- **greater** - If set, violations are created where the test result is greater than the rule, e.g. geomWidth(M1, 0.030, greater) will result in shapes with width greater than 30nm selected for error/output.
- **output_only** - Do not flag a violation on the marker, used when a DRC check generates edge data to be further processed.
- **opposite** - Report violations with opposite lengths of the edges only. Else the full length of the edges are reported.

3.2.2 geomWidth(layer, rule, message=NULL)

Checks layer for minimum width violations i.e. widths less than rule. Error polygons are created on the drcMarker layer in the current cellView. The rule dimension must be specified in microns.

An optional message will be written as a property 'drcWhy' on the marker shape if specified.

3.2.3 geomWidth(layer, rule, flags, message=NULL)

Checks layer for minimum width violations i.e. widths less than rule. Error polygons are created on the drcMarker layer in the current cellView. The rule dimension must be specified in microns. flags can be either horizontal, vertical or diagonal, in which case only horizontal, vertical edges or diagonal edges will be checked for width.

An optional message will be written as a property 'drcWhy' on the marker shape if specified.

3.2.4 geomAllowedWidths(layer, rules, flags, message=NULL)

Checks layer for width violations. The widths must be discrete values specified in rules, which is a python sequence. Error polygons are created on the drcMarker layer in the current cellView. The rules must be specified in microns. flags can be none, horizontal, vertical or diagonal. If none, then the minimum width of a shape is checked against the allowed widths. If horizontal, then horizontal edge separation(s) of shapes are checked; if vertical, then distances between vertical edges of shapes are checked.

An optional message will be written as a property 'drcWhy' on the marker shape if specified.

Example:

```
geomAllowedWidths(poly, [0.020, 0.022, 0.024], horizontal)
```

3.2.5 geomLength(layer, rule, message=NULL)

3.2.6 geomLength(layer, rule, flags, message=NULL)

Checks layer for minimum length violations i.e. lengths less than rule. Error polygons are created on the drcMarker layer in the current cellView. The rule dimension must be specified in microns. flags can be either horizontal, vertical or diagonal, in which case only horizontal, vertical edges or diagonal edges will be checked for width.

An optional message will be written as a property 'drcWhy' on the marker shape if specified.

3.2.7 geomSpace(layer, rule, message=NULL)

3.2.8 geomSpace(layer, rule, flags, message=NULL)

Checks layer for minimum spacing violations i.e. single layer spacings less than rule. Error polygons are created on the drcMarker layer in the current cellView. The rule dimension must be specified in microns. Note that spacing violations between edges of the same polygon are not reported; to detect these perform a geomNotch check. An optional message will be written to the error marker flag if specified. flags can be used to control the spacing check. A flag of samenet applies the check only if the two shapes are on the same physical net. A flag of diffnet applies the rule only if the two shapes are on different nets. In order to check samenet/diffnet spacings, a geomConnect() command must have been previously given to form connectivity.

Example:

```
geomSpace(active, 0.2, samenet, "active space < 0.2 for same net")
geomSpace(active, 0.3, diffnet, "active space < 0.3 for different nets")
```

3.2.9 geomSpace(layer, rule, width, length, flags, message=NULL)

Checks layer for minimum spacing violations i.e. single layer spacing less than rule, where one of the shapes has a width > width and a parallel run length > length. Error polygons are created on the drcMarker layer in the current cellView. The rule and width dimensions must be specified in microns. Note that spacing violations between edges of the same polygon are not reported; to detect these perform a geomNotch check. An optional message will be written to the error marker flag if specified. flags can be used to control the spacing check. A flag of samenet applies the check only if the shapes are on the same physical net. A flag of diffnet applies the rule only if the two shapes are on different nets. In order to check samenet / diffnet spacings, a geomConnect() command must have previously been given to form connectivity. flags can also be either horizontal, vertical or diagonal, in which case only horizontal, vertical edges or diagonal edges will be checked for spacing. flags can include abut, in which case abutting edges are considered a violation. Note that geomSpace with no flags will check all edges (horizontal, vertical and diagonal) so it is not necessary to specify e.g. a rule with no flags, and one with a 'vertical' flag unless they are different. Flags can be combined by python logical OR.

Example:

```
geomSpace(active, 0.2, 10.0, samenet, "active space < 0.2 for same net with width > 10.0")
geomSpace(active, 0.3, 10.0, diffnet, "active space < 0.3 for different nets with width > 10.0")
```

3.2.10 geomSpace2(layer, rule, width, length, flags=0, message = NULL)

Checks layer for minimum spacing violations i.e. single layer spacing less than rule, where both of the shapes has a width > width and a parallel run length > length. Error polygons are created on the drcMarker layer in the current cellView. The rule and width dimensions must be specified in microns. Note that spacing violations between edges of the same polygon are not reported; to detect these perform a geomNotch check. An optional message will be written to the error marker flag if specified. flags can be used to control the spacing check. A flag of samenet applies the check only if the shapes are on the same physical net. A flag of diffnet applies the rule only if the two shapes are on different nets. In order to check samenet / diffnet spacings, a geomConnect() command must have previously been given to form connectivity. flags can also be either horizontal, vertical or diagonal, in which case only horizontal, vertical edges or diagonal edges will be checked for spacing. flags can include abut, in which case abutting edges are considered a violation. Note that geomSpace with no flags will check all edges (horizontal, vertical and diagonal) so it is not necessary to specify e.g. a rule with no flags, and one with a 'vertical' flag unless they are different. Flags can be combined by python logical OR.

Example:

```
geomSpace2(active, 0.2, 10.0, samenet, "active space < 0.2 for same net with width >
10.0")
geomSpace2(active, 0.3, 10.0, diffnet, "active space < 0.3 for different nets with width >
10.0")
```

3.2.11 geomSpace(layer1, layer2, rule, message=NULL)

3.2.12 geomSpace(layer1, layer2, rule, flags, message=NULL)

Checks layer1 to layer2 for minimum spacing violations i.e. two layer spacings less than rule. Error polygons are created on the drcMarker layer in the current cellView. The rule dimension must be specified in microns. An optional message will be written to the error marker flag if specified. flags can be used to control the spacing check. A flag of samenet applies the check only if the two shapes are on the same physical net. A flag of diffnet applies the rule only if the two shapes are on different nets. In order to check samenet/diffnet spacings, a geomConnect() command must have been previously given to form connectivity. flags can also be either horizontal, vertical or diagonal, in which case only horizontal, vertical edges or diagonal edges will be checked for spacing. flag can include abut, in which case abutting edges are considered a violation. Note that geomSpace with no flags will check all edges (horizontal, vertical and diagonal) so it is not necessary to specify e.g. a rule with no flags, and one with a 'vertical' flag unless they are different. Flags can be combined by python logical OR.

Example:

```
geomSpace(nwell, ndiff, 0.2, samenet, "nwell to n+ diff space < 0.2 for same net")
geomSpace(nwell, ndiff, 0.3, diffnet, "nwell to n+ diff space < 0.3 for different nets")
```

3.2.13 geomAllowedSpaces(layer, rules, flags, message=NULL)

Checks layer for space violations. The spaces must be discrete values specified in rules, which is a python sequence. Spacing greater or equal to the last rule is allowed. Error polygons are created on the drcMarker layer in the current cellView. The rules must be specified in microns. flags can be none, horizontal, vertical or diagonal. If none, then the minimum width of a shape is checked against the allowed spaces. If horizontal, then horizontal edge separation(s) of shapes are checked; if vertical, then vertical edge separation(s) of shapes are checked.

An optional message will be written as a property 'drcWhy' on the marker shape if specified.

Example:

```
geomAllowedSpaces(active, [0.020, 0.022, 0.024], horizontal)
```

In the above, the layer 'active' must have spacing of either 20nm, 22nm or >= 24nm.

3.2.14 geom2DSpace(layer, rules, flags, message=NULL)

Checks layer for spacing that is both length and width dependent. The rules consist of a 2D python array, of which row 0 defines the widths of the rules, and column 0 defines the lengths of the rules. The other entries are the rule values. flags are as defined above. An optional message will be written as a property 'drcWhy' on the marker shape if specified.

Example:

```
geom2DSpace(m1,  [ [0.000, 0.028, 0.032, 0.040, 0.064, 0.120, 0.240, 0.320, 0.600],
                   [0.028, 0.036, 0.036, 0.036, 0.036, 0.036, 0.036, 0.036, 0.036],
                   [0.240, 0.036, 0.068, 0.076, 0.076, 0.076, 0.076, 0.076, 0.076],
                   [0.480, 0.036, 0.068, 0.076, 0.092, 0.092, 0.092, 0.092, 0.092],
                   [1.200, 0.036, 0.068, 0.076, 0.092, 0.120, 0.120, 0.120, 0.120],
                   [1.800, 0.036, 0.068, 0.076, 0.092, 0.120, 0.240, 0.240, 0.240],
                   [2.400, 0.036, 0.068, 0.076, 0.092, 0.120, 0.240, 0.320, 0.600]],
             0, "M1 Minimum spacing")
```

The above defines a minimum rule of 28nm for normal metal. For metal wider than 240nm then if the width is wider than 32nm the rule is 68nm, if the width is wider than 40nm the rule is 76nm etc.

3.2.15 geomNotch(layer1, rule, message=NULL)

3.2.16 geomNotch(layer1, rule, flags, message=NULL)

Checks layer1 polygons for notch violations. Error polygons are created on the drcMarker layer in the current cellView. The rule dimension must be specified in microns. Note that notches are effectively spacing violations between edges of the same polygon. An optional message will be written as a property 'drcWhy' on the marker shape if specified.

3.2.17 geomLineEnd(layer1, rule, num_ends, min_adj_edge_length=0.0, flags=0, message=NULL)

Checks layer1 for minimum line end spacings. The spacing is from the line end edge to another edge which is either a normal edge (if num_ends=1) or another line end edge (if num_ends = 2). A line end edge is a horizontal edge with two adjacent vertical edges, or a vertical edge with two adjacent horizontal edges. The adjacent edge length must be greater than the line end edge length, or min_adj_edge_length, whichever is the greater.

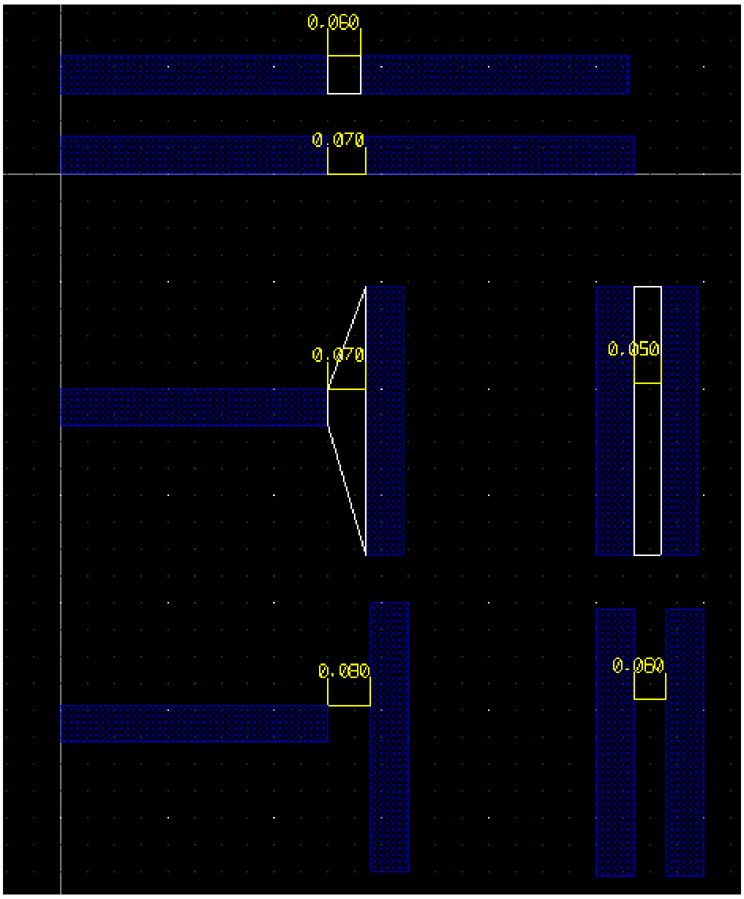


Figure 147 - geomLineEnd

In the above example, the rules are as follows:

geomSpace(metal1, 0.06)
geomLineEnd(metal1, 0.08, 1)
geomLineEnd(metal1, 0.07, 2)

3.2.18 geomLineEnd(layer1, layer2, rule, num_ends, min_adj_edge_length=0.0, flags = 0, message=NULL)

As above but for two layer checking.

3.2.19 geomPitch(layer1, rule, flags = 0, message=NULL)

Checks that layer1 is on a pitch greater than or equal to the rule specified. flags can be vertical or horizontal. An optional message will be written as a property 'drcWhy' on the marker shape if specified.

3.2.20 geomOverlap(layer1, layer2, rule, message=NULL)

3.2.21 geomOverlap(layer1, layer2, rule, flags, message=NULL)

Checks layer1 to layer2 for minimum overlap violations i.e. layer1 overlaps layer2 by less than rule. Error polygons are created on the drcMarker layer in the current cellView. The rule dimension must be specified in microns. An optional message will be written as a property 'drcWhy' on the marker shape if specified.

3.2.22 geomEnclose(layer1, layer2, rule, message=NULL)

3.2.23 geomEnclose(layer1, layer2, rule, flags, message=NULL)

Checks layer1 to layer2 for minimum enclosure violations i.e. layer1 encloses layer2 by less than rule. Error polygons are created on the drcMarker layer in the current cellView. The rule dimension must be specified in microns. The optional flags can have the 'abut' flag set which considers abutting edges an error; otherwise abutting edges are allowed. An optional message will be written as a property 'drcWhy' on the marker shape if specified.

3.2.24 geomEnclose2(layer1, layer2, rule1, rule2, rule3, edges, message=NULL)

Checks layer1 to layer2 for minimum enclosure violations. layer1 should enclose layer2 by rule1 normally. However if there 1 or more edges of layer2 with enclosure greater than or equal to rule2, but less than rule1, and edges (e.g. 2) perpendicular edges of layer1 enclose layer2 by greater than or equal to rule3, then no violation occurs. The rule* dimensions must be specified in microns. An optional message will be written to the error marker flag if specified. Any edge enclosure less than rule2 will give an error. The parameter edges must be 1 or 2.

Example:

```
geomEnclose2(nwell, active, 0.18, 0.08, 0.23, 2)
```

Enclosure of active by nwell should be >= 0.18um, however if 2 parallel edges of active have a nwell enclosure of 0.08um then the other two perpendicular edges should have a minimum enclosure of 0.23um.

```
layer = geomEnclose2(cont, metal, 0.15, 0.05, 0.30, 1)
```

Enclosure of cont by metal should be 0.15um, however an edge can be enclosed by 0.05um if one perpendicular edge is greater than or equal to 0.3um.

3.2.25 geomAllowedEncs(layer1, layer2, rules, message=NULL)

Checks layer1 to layer2 for minimum enclosure violations. layer1 must enclose layer2 according to rules, which is a list of triplets e.g. [[0.010, 0.010, 4], [0.0, 0.032, 2], [0.002, 0.028, 2]]. For each triplet, the first two numbers are allowed enclosures, and the third number is the number of sides that must obey the second rule. For example, in the first case [0.010, 0.010, 4] all 4 sides of layer2 can be enclosed by 10nm. In the second case [0.0, 0.032, 2] there can be 2 opposite sides with enclosure of 0nm, and the other two sides must have an enclosure of 32nm. Finally in the third case [0.002, 0.028, 2] there can be 2 opposite sides with enclosure of 2nm and the other two sides must have enclosure of 28nm. These are the only allowed enclosure values; anything else will give a violation. The rule dimensions must be specified in microns. An optional message will be written as a property 'drcWhy' on the marker shape if specified.

3.2.26 geomExtension(layer1, layer2, rule, message=NULL)

3.2.27 geomExtension(layer1, layer2, rule, flags, message=NULL)

Checks layer1 to layer2 for minimum extension violations i.e. layer1 extends beyond layer2 by less than rule. Error polygons are created on the drcMarker layer in the current cellView. The rule dimension must be specified in microns. An optional message will be written as a property 'drcWhy' on the marker shape if specified.

3.2.28 geomArea(layer1, minrule, maxrule=9e99, message=NULL)

Checks layer1 shapes for minimum area violations. Error polygons are created on the drcMarker layer in the current cellView. The minrule and optional maxrule dimensions must be specified in microns. An optional message will be written as a property 'drcWhy' on the marker shape if specified.

3.2.29 geomMargin(layer1, rule, message=NULL)

Checks layer1 shapes for minimum margin violations. A margin violation is the distance (typically greater than the normal minimum spacing), from the vertex common to two adjacent concave edges of a polygon, to edge(s) of a nearby polygon. Error polygons are created on the drcMarker layer in the current cellView. The rule dimension must be specified in microns. An optional message will be written as a property 'drcWhy' on the marker shape if specified.

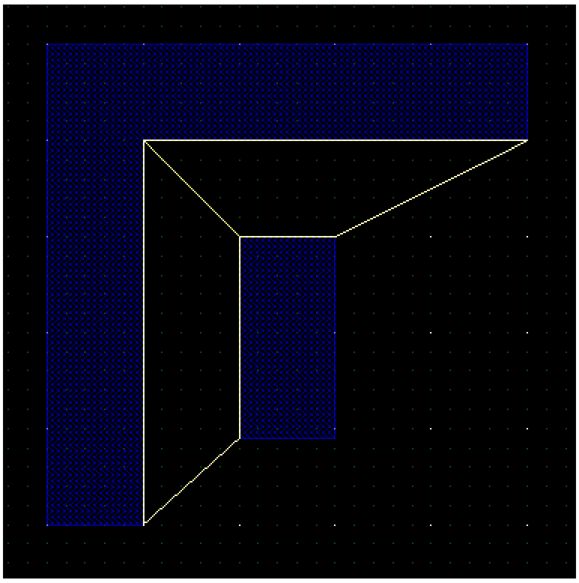


Figure 148 - geomMargin

In the above example, the distance of the inner (concave) edge of the L shaped polygon vertex is less than the specified rule to the nearest vertex of the rectangle. Two error flags are created because there are two edges of the rectangle containing the vertex in violation.

3.2.30 geomOffGrid(layer1, grid, marker_size=0.1, message=NULL)

Checks all layer1 vertices of edges to see if they are on a multiple of the grid specified in microns by grid. marker_size is the size of the marker in microns (shown as a '+' centered on the offgrid vertex) on the drcMarker layer. An optional message will be written to the error marker flag if specified. The return value is the number of off-grid vertices found.

3.2.31 geomAdjLength(layer1, rule, length, flags, message=NULL)

Checks vertices for adjacent edge length, If one edge has length less than length, the other edge must have length greater than rule.An optional message will be written as a property 'drcWhy' on the marker shape if specified.

3.2.32 geomAllowedSize(layer1, rule, message=NULL)

Checks the size of rectangular shapes e.g. contacts or vias. rule specifies the permissible edge lengths as length/width pairs.An optional message will be written as a property 'drcWhy' on the marker shape if specified.

For example:

```
geomAllowedSize(via, [[0.028, 0.028],[0.028,0.056]], "Via is rectangular 28x28 or 28x56nm")
```

Checks shapes on the layer via which must be rectangles with either sides of 28nm or 2 sides of 28nm and 2 sides of 56nm.

3.2.33 geomGetCount()

Returns the number of errors detected in the most recent DRC check.

3.2.34 geomGetTotalCount()

Returns the total number of DRC errors since the start of the run.

3.3 Extraction

Glade can trace connectivity in a layout and identify devices such as resitors, capacitors, diodes, mos and bipolar transistors plus parasitic capacitors. To extract a layout, a python script is used to form derived layers, run connectivity tracing and extract devices. The results are saved to a cellView with the viewName of 'extracted' but this can be changed using the setExtViewName command.

Connectivity tracing is performed using the geomConnect command. Device extraction is performed using the extract… commands. When devices are extracted, a PCell is used to form an instance in the extracted view which is connected via the layers traced in geomConnect. The PCell is passed the coordinates of the recognition region used to identify the device, along with any properties for the particular type of device. Parasitic extraction determines capacitance between connect layers either using a simple area/perimeter calculation, or a more accurate (but much slower) 3D field solver 'Fastcap'.

The extracted view can be netlisted to a CDL/Spice file; there are two methods by which the netlister will format the lines for each instance in the file. If a string property named 'NLPDeviceFormat' is present on the PCell master, this property allows user defined netlisting. See 'NLP expressions' for more details of the NLPDeviceFormat syntax. If this property is not present, the netlister will look for hardcoded property names on devices:

- MOS/TFT : 'w', 'l', 'm', 'as', 'ad', 'ps', 'pd'
- Resistors : 'r', 'w', 'l'
- Capacitors : 'c', 'w', 'l'
- Inductors : 'l'
- Diodes : 'area', 'pj' or 'perim'
- Bipolars : 'area', 'pj' or 'perim'

The hardcoded netlister expects specific pin names on the extraction PCell devices:

- MOS : D, G, S, B

- FET : D, G, S
- Resistors : A, B
- Capacitors : A, B
- Inductors : A, B
- Diodes : A, C
- Bipolars : C, B, E

3.3.1 setExtViewName(name)

Sets the name of the extracted view. The default is "extracted", but for e.g. abstract generation you can set this to "abstract". Note this command should be given before any saveDerived / saveInterconnect commands.

3.3.2 geomConnect([[viaLayer, bottomLayer, topLayer], [...]])

Trace connectivity through layers. This function takes a list of lists of layers, where the layers are a via or contact layer and the layers that are connected by it. For example:

```
geomConnect( [
   [cont, active, poly, metal1],
   [via1, metal1, metal2]
] )
```

The above will connect active and metal1 by the cont layer, poly and metal1 also by the cont layer, and metal1 and metal2 by the via1 layer. There is no limit to the number of lists of layers, or to the number of layers connected by a contact layer. However the list of connected layers must have only one contact/via layer, and that layer must be the first layer in the list. If shapes already have net information (e.g. through the use of the geomLabel() command) then these shapes are used as initial tracing points, and net names are propagated to connected shapes. Other shapes are assigned automatically generated names (n0, n1, n2 etc). Shorts between shapes with assigned or traced net names that are different are reported.

The geomConnect() command uses a scanline algorithm combined with graph labelling and is quite fast compared to e.g. the Net Tracer, and is multithreaded. The number of threads used can be controlled by setting the environment variable GLADE_THREADED_EXTRACTION. The env var can take an optional value, being the maximum number of threads to run. For example on a Core i7 cpu with 4 physical cores each capable of running 2 threads, you could set GLADE_THREADED_EXTRACTION=8

3.3.3 geomLabel(layer, labelLayer, labelPurpose = "drawing", createPin= True)

Label a layer with existing text labels. If a text label with layer labelLayer and purpose labelPurpose has its origin contained in a shape on layer, then the shape will have its net name set to the text label name. Note that labelling layers should be performed prior to connectivity extraction for net name propagation. Logical nets/pins will be created in the extracted view for all text labels that attach to shapes on layer. If not specified, labelPurpose defaults to "drawing" . Note that labels are only used for the top level of the design; in other words labels at lower levels of the layout hierarchy are ignored. For LVS purposes, labelling power, ground, clock and primary IOs is all that is usually necessary. createPin can be set to False (default is True) to disable creating a pin - only a net will be created.

3.3.4 geomSetText(layer, xcoord, ycoord, labelName, createPin = True)

Labels layer with text label labelName at the coordinates given by xcoord and ycoord (in microns). Returns True if a shape on the layer was found at the given xy coordinates; False if no shape was found (i.e. the command failed). This is useful if you cannot modify the original layout and want to try and resolve LVS errors by forcing a shape to be a specified net name. createPin can be set to False (default is True) to disable creating a pin - only a net will be created.

3.3.5 saveDerived(layer, why, outLayer = TECH_DRCMARKER_LAYER)

Outputs layer geometries as polygons to the current cellView. The layer they are output to can be set by outLayer, which defaults to the drcMarker layer. Each polygon has a string property drcWhy with value set to the string why.

3.3.6 saveDerived(layer, layerName, purpose, viewType="ext_view")

Outputs layer geometries as polygons to the current cellView. The layer they are output to can be set by layerName and purpose. If viewType is specified, the layer geometries are output to this view name rather than the default view name 'extracted'.

3.3.7 saveInterconnect([layer1, layer2, ...])

Creates a new cellview with the same cell name as the current cell, and a view name of 'extracted'. Shapes on layers specified are created in the extracted cellview. Shapes will have net information if they are on layers present in geomLabel and/or geomConnect() commands. Optionally instead of a layer name, a list of derived layer name, a techFile layer name and optionally a purpose can be specified, for example:

```
saveInterconnect( [
        poly,
        active,
        [ metal1, "M1" ],
        [ metal2, "M2", "pin"]
] )
```

Original layers that are generated from geomGetShapes() do not need the techFile layer name/purpose specified, but derived layers MUST specify the target layer name, else they will be assigned a fake layer number which the LSW will not show. It is often desirable to add dummy layers to the techFile and use these for saveInterconnect(). For example, when extracting a lateral PNP, derived layers for emitter, base and collector need to be generated for the extractBjt() terminals. In this case it's desirable to have dummy layers 'emitter', 'base' and 'collector' so that devices get extracted correctly.

If an existing layer name is specified without a purpose name, the purpose name defaults to 'drawing'. There is no limit on the number of layers in the list. Note that any terminal layer used in subsequent 'extract...' commands should be saved.

3.3.8 extractMOS(modelName, recLayer, gateLayer, diffLayer, bulkLayer)

Extracts MOS devices and creates instances of a cellView *'modelName* layout' in the extracted view. *recLayer* is the recognition layer of the gate region. *gateLayer* is the poly layer and *diffLayer* is the source/drain diffusion layer. *bulkLayer* is the optional well layer; if present the extracted instances have terminals D G S B, otherwise 3 terminal devices with terminals D G S are extracted. The layers *gateLayer*, *diffLayer* and *bulkLayer* must have previously been saved using the saveInterconnect command.

The extracted instances have the property 'w' set to the recLayer shape width (length of gate recognition shape edge coincident with diffLayer) and 'l' set to the distance between the coincident diffLayer/gateLayer edges. Both manhattan and any-angle gates are supported. The recLayer shapes should be a simple polygon without holes.

The cellview *'modelName* layout' will be created if it does not already exist. If it does exist, it is assumed to be a PCell, and its ptlist property is set to the point list of the recognition region. Example nmos/pmos cells with parameterised point lists are nmos_ex.py and pmos_ex.py.

3.3.9 extractDevice(modelName, recLayer, [[termLayer1, "term1", ...] [termLayer2, "term2", ...]]

Extracts a generic deviceand creates instances of a cellView *'modelName* layout' in the extracted view. The first letter of the *modelName* should correspond to the Spice device type e.g. R for a resistor, C for a capacitor (case insensitive) etc. *recLayer* is the device recognition layer. The *termLayer*(s) should be connection layers previously saved by the saveInterconnect command. Each terminal layer can have one of more terminal names. Shapes on the terminal layer(s) that touch or overlap the recognition layer will be created as terminals of the device. The *recLayer* shapes should be a simple polygon without holes.

The cellview *'modelName* layout' will be created if it does not already exist. If it does exist, it is assumed to be a PCell, and its ptlist property is set to the point list of the recognition region.

3.3.10 extractRes(modelName, recLayer, termLayer, bulkLayer)

Extracts a 2 terminal resistor and creates instances of a cellView *'modelName* layout' in the extracted view. *recLayer* is the recognition layer for the resistor. *termLayer* is the layer of the resistor terminals e.g. poly, and shapes on this layer should overlap or touch the recognition layer shape. The layer *termLayer* must have previously been saved using the saveInterconnect command. If the optional layer *bulkLayer* is specified, an additional bulk node is generated for the resistor model.

The extracted instances will have properties ‘w’ set the the *recLayer* width (length of recLayer edge coincident with termLayer edge) and ‘l’ set to the *recLayer* length (total length of all recLayer edges minus twice the width, then divided by two), ‘nsquares’ set to l/w, ‘nbends’ to the number of bends. These properties can be accessed via an extraction PCell - see the example 'pres_ex.py' in the distribution.

The cellview *'modelName* layout' will be created if it does not already exist. If it does exist, it is assumed to be a PCell, and its ptlist property is set to the point list of the recognition region. If a PCell is used, its terminals are expected to be "A" and "B".

3.3.11 extractMosCap(modelName, recLayer, gateLayer, diffLayer)

Extracts a 2 terminal capacitor and creates instances of a cellView *'modelName* layout' in the extracted view. *recLayer* is the recognition layer for the capacitor. *gateLayer* and *diffLayer* form the terminal layers of the capacitor, and these layers must have previously been saved using the saveInterconnect command.

The extracted instances will have the properties area set the the *recLayer* area and perim set to the recLayer perimeter.

The cellview *'modelName* layout' will be created if it does not already exist. If it does exist, it is assumed to be a PCell, and its ptlist property is set to the point list of the recognition region. If a PCell is used, its terminals are expected to be "G" for the gate layer and "S" for the diff layer.

3.3.12 extractDio(modelName, recLayer, anodeLayer, cathodeLayer)

Extracts a 2 terminal diode and creates instances of a cellView *'modelName'* layout in the extracted view. *recLayer* is the recognition layer for the capacitor. *anodeLayer* and *cathodeLayer* form the terminal layers of the diode, and shapes on these layers should overlap or touch the recognition layer shape, and these layers must have previously been saved using the saveInterconnect command. .

The extracted instances will have the properties 'area' set the the recLayer area and 'pj' set to the recLayer perimeter.

The cellview *'modelName* layout' will be created if it does not already exist. If it does exist, it is assumed to be a PCell, and its ptlist property is set to the point list of the recognition region. If a PCell is used, its terminals are expected to be "A" for the anode and "C" for the cathode.

3.3.13 extractBjt(modelName, recLayer, emitLayer, baseLayer, collLayer)

Extracts a 3 terminal bjt and creates instances of a cellView *'modelName* layout in the extracted view. *recLayer* is the recognition layer for the bjt. *emitLayer*, *baseLayer* and *collLayer* form the terminal layers of the bjt, and shapes on these layers should overlap or touch the recognition layer shape, and these layers must have previously been saved using the saveInterconnect command. .

The extracted instances will have the properties 'area' set the the emitter recLayer area and 'perim' set to the recLayer perimeter.

The cellview *'modelName* layout' will be created if it does not already exist. If it does exist, it is assumed to be a PCell, and its ptlist property is set to the point list of the recognition region. If a PCell is used, its terminals are expected to be "C" for the collector, "B" for the base and "E" for the emitter.

3.3.14 extractTFT(modelName, recLayer, gateLayer, diffLayer)

Extracts TFT (thin film) MOS devices and creates instances of a cellView *'modelName* layout' in the extracted view. *recLayer* is the recognition layer of the gate region. *gateLayer* is the poly layer and diffLayer is the source/drain diffusion layer. The layers *gateLayer* and *diffLayer* must have previously been saved using the saveInterconnect command. The *gateLayer* is normally the bottom metal1 plate and the *diffLayer* the top metal2 fingers.

The extracted instances have the property 'w' set to the *recLayer* shape width (length of gate recognition shape coincident with diffLayer) and 'l' set to the distance between the coincident edges. Both manhattan and any-angle gates are supported. The *recLayer* shapes should be simple polygons without holes.

The cellview '*modelName* layout' will be created if it does not already exist. If it does exist, it is assumed to be a PCell, and its ptlist property is set to the point list of the recognition region. If a PCell is used, its terminals are expected to be "D" for the drain, "S" for the source and "G" for the gate.

3.3.15 extractParasitic(metLayer, areaCap, perimCap, gndNetName)

Extracts the parasitic capacitance of net shapes on layer *metLayer*. *metLayer* can be any layer in the geomConnect() set of layers; for each (merged) shape on *metLayer* its area (in microns^2) and perimeter (in microns) are calculated and multiplied by the values of *areaCap* and *perimCap*. An instance of a capacitor is created (of size 100x100 database units so not normally visible) on one of the vertices of the shape, with property 'c' set to the area * areaCap + perimeter * perimCap. The capacitance will be connected to the shape's net and to the ground net specified by *gndNetName*.

3.3.16 extractParasitic2(metLayer1, met2Layer, areaCap, perimCap)

Extracts the parasitic capacitance of net shapes between layers *met1Layer* and *met2Layer*. The two layers can be any layer in the geomConnect() set of layers; for each intersection of met1Layer and met2Layer the area (in microns^2) and perimeter (in microns) are calculated and multiplied by the values of *areaCap* and *perimCap*. An instance of a capacitor is created (of size 100x100 database units so not normally visible) on one of the vertices of the shape, with property 'c' set to the area * areaCap + perimeter * perimCap. The capacitance will be connected between the nets of the shapes of each metal layer.

3.3.17 extractParasitic3(metLayer1, met2Layer, areaCap, perimCap, [layer1,...layerN])

Only extracts capacitance between *metal1Layer* and *metal2Layer* if shield layer(s) *layer1*... *layerN* are not present between them. Note no checking is done for valid layers (yet). The two layers can be any layer in the geomConnect() set of layers; for each intersection of *met1Layer* and *met2Layer* the area (in microns^2) and perimeter (in microns) are calculated and multiplied by the values of *areaCap* and *perimCap*. An instance of a capacitor is created (of size 100x100 database units so not normally visible) on one of the vertices of the shape, with property 'c' set to the area * areaCap + perimeter * perimCap. The capacitance will be connected between the nets of the shapes of each metal layer.

3.3.18 extractParasitic3D(subsNetName, refNetName, tol=0.01, order=-1, depth=-1)

Perform parasitic capacitance extraction using the Fastcap 3D field solver. *subsNetName* is the name of the silicon substrate net; capacitances from conductor layers to the substrate plane will have this net as one of their terminals. *refNetName* is the name of the reference net used by the field solver. *tol* is an optional tolerance (fastcap -t option) and defaults to 0.01 i.e. 1%. *order* is an optional parameter and corresponds to the fastcap option -o. By default (-1) fastcap automatically sets this. A higher number e.g. 3 may be used e.g. if accuracy of net-net capacitance is small and fastcap gives warning about non-negative values of the capacitance matrix. *depth* is the partitioning depth and corresponds to the fastcap option -d. It defaults to fastcap automatically setting the value.

In order to extract parasitics for layers, they must be defined in the techFile with non-zero thickness. An example:

```
METLYR metal1 drawing HEIGHT 0.890 THICKNESS 0.280 ;
VIALYR via1 drawing HEIGHT 1.170 THICKNES 0.450 ;
```

METLYR metal2 drawing HEIGHT 1.620 THICKNESS 0.370 ;

In the above, HEIGHT specifies the conductor height above the silicon surface and THICKNESS the layer thickness. Dielectric constants are assumed to be 3.9 currently; the ability to set dielectric layer thickness/permittivity may be added in future. Automatic meshing is performed to generate fastcap compatible format input files. Each layer shape has conductors with a net name resulting from a geomConnect() connectivity extraction. Capacitances are calculated by Fastcap as a matrix in which the diagonal elements are the total capacitance for the conductor, and off-diagonal elements are the capacitances between conductors. Capacitances are backannotated to the extracted view as instances of a parasitic cap 'pcap'; netlisting the extracted view using File->Export CDL will allow a spice compatible netlist to be generated.

In addition to the substrate net, a reference net *refNetName* is also created. Capacitances to this net represent field lines from a conductor to infinity. For most usage this can be lumped to the substrate net by a zero volt source connecting the two in your simulation testbench.

The temporary files produced by Glade are in Fastcap2 format, so another extractor could be used that can read this format. Temp files are created in the current working directory, or in the directory specified by the env var GLADE_FASTCAP_WORK_DIR. Temp files are normally deleted after extraction is completed; the env var GLADE_NO_DELETE_TMPFILES can be set to keep them. Glade expects is to find an executable called 'fastcap.exe' (windows) or 'fastcap' (Linux) in the same directory as the glade executable.

Note that Fastcap is a field solver - and as such is not designed to handle large problems. Typically cells with up to about 50 nets will extract in a resonable amount of time and memory .

3.3.19 LVS

4 PCells

4.1.1 PCell Flow

PCells allow Glade to reuse layout of cellViews that may have differences – ‘parameterised cells’. The layout for the cell is created by a script which takes the parameters as arguments. Note that Glade PCells are **NOT** compatible with Cadence Skill-based PCells, or Synopsys PyCells.

The PCell scripting language in Glade is Python, the same as used to access the database and GUI. When a PCell is compiled, it creates a master cellView known as a ‘supermaster’. The purpose of the supermaster is to provide a cellView that can be instanced in a design. When the cellView is instanced, a submaster cellView is created, which is unique by view of its parameter values. Submasters are named of the form <supermaster_name>$$nnnnnnnn where nnnnnnnn is an unique number created by hashing the parameter name/values etc. Submasters are normally hidden in the library browser, as they should never be manipulated directly by the user. However there is an environment variable, GLADE_DEBUG_SUBMASTERS, which can be set to make submasters visible in the library browser.

4.1.2 An example PCell

An example of a MOS transistor PCell is as follows.

```
#-----------------------------------------------------------------------------
# NMOS Pcell example
#
# Note: The first argument is always the cellView of the subMaster.
# All subsequent arguments should have default values and will
# be passed by name
#-----------------------------------------------------------------------------

# Import the db wrappers
from ui import *

# The entry point. The name should match the superMaster.
def nmos(cv, w=1.1e-06, l=0.18e-06)

# Some useful variables
lib = cv.lib()
dbu = lib.dbuPerUU()
width = int(w * 1.0e6 * dbu)
length = int(l * 1.0e6 * dbu)

# Some predefined rules
cut = int(0.18 * dbu)
poly_to_cut = int(0.1 * dbu)
active_ovlp_cut = int(0.1 * dbu)
poly_ovlp_active = int(0.12 * dbu)
nplus_ovlp_active = int(0.2 * dbu)
metal_ovlp_cut = int(0.05 * dbu)


# Create active
tech = lib.tech()
layer = tech.getLayerNum("active", "drawing")
r = Rect(-width/2,
        -(active_ovlp_cut + cut + poly_to_cut + length/2),
        width/2,
        (active_ovlp_cut + cut + poly_to_cut + length/2))
active = cv.dbCreateRect(r, layer);
bbox = Rect(active.bBox())

# Create nplus
layer = tech.getLayerNum("nplus", "drawing")
r = Rect(bbox.left() - nplus_ovlp_active,
        bbox.bottom() - nplus_ovlp_active,
        bbox.right() + nplus_ovlp_active,
        bbox.top() + nplus_ovlp_active)
cv.dbCreateRect(r, layer);

# Create poly
```

```
layer = tech.getLayerNum("poly", "drawing")
p = Rect(-width/2-poly_ovlp_active,
        -length/2,
        width/2+poly_ovlp_active,
        length/2)
gate = cv.dbCreateRect(p, layer)
net = cv.dbCreateNet("G")
pin = cv.dbCreatePin("G", net, DB_PIN_INPUT)

# Create contacts
layer = tech.getLayerNum("cont", "drawing")
numCuts = width / (2 * cut)
c = Rect(-width/2 + active_ovlp_cut - cut * 2,
        -(length/2 + poly_to_cut + cut),
        -width/2 + active_ovlp_cut - cut,
        -(length/2 + poly_to_cut))

for i in range(numCuts) :
        c.offset(cut * 2, 0)
        cv.dbCreateRect(c, layer)

c = Rect(-width/2 + active_ovlp_cut - cut * 2,
        (length/2 + poly_to_cut),
        -width/2 + active_ovlp_cut - cut,
        (length/2 + cut + poly_to_cut))

for i in range(numCuts) :
        c.offset(cut * 2, 0)
        cv.dbCreateRect(c, layer)

# Create metal
layer = tech.getLayerNum("metal", "drawing")
m = Rect(-width/2 + active_ovlp_cut - metal_ovlp_cut,
        -length/2 - poly_to_cut - cut - metal_ovlp_cut,
        width/2 - active_ovlp_cut + metal_ovlp_cut,
        -length/2 - poly_to_cut + metal_ovlp_cut)
source = cv.dbCreateRect(m, layer)
net = cv.dbCreateNet("S")
pin = cv.dbCreatePin("S", net, DB_PIN_INPUT)

m = Rect(-width/2 + active_ovlp_cut - metal_ovlp_cut,
        length/2 + poly_to_cut - metal_ovlp_cut,
        width/2 - active_ovlp_cut + metal_ovlp_cut,
        length/2 + poly_to_cut + cut + metal_ovlp_cut)
drain = cv.dbCreateRect(m, layer)
net = cv.dbCreateNet("D")
pin = cv.dbCreatePin("D", net, DB_PIN_INPUT)

# Update the subMaster's bounding box
cv.update()
```

In the above example, we declare a function called 'nmos' which takes 3 arguments. The first argument is always a cellView object. The remaining arguments can be any desired parameters; they must all have default parameters. This is so that if one of the parameters is missing, the default value can be used.

Note that you can pass a list of points to a Pcell. A list of points is defined in the standard Python syntax, and can be set as a string property in the Add Property dialog e.g.

```
[[0,0],[1000,0],[1000,1000],[0,1000]]
```

Also note that all dimensions must be converted to database units (dbu). The dimensional quantities (l, w in this case) should be passed as units of metres rather than microns if schematic driven layout is used. This is so device W, L etc can be entered as ‘1.0u’ in schematics i.e. using SPICE compatible multipliers.

4.1.3 Changing PCell arguments from within PCell code

If you change any of the PCell arguments within your code and want the instance properties updated, you should save your properties e.g.

```
def nmos(cv, w=1.1e-06, l=0.18e-06)
        a = w * l
        cv.dbAddProp("a", a)
```

Why might you want to do this? If you create an extraction PCell (see e.g. rppoly_ex.py in the distribution directory), you can include PCell arguments that you can to calculate and use for netlisting. For example for the resistor, the PCell code computes 'r' from the extracted width, length, number of bends and the resistor's sheet resistance in ohms/sq which can be hard coded into the extraction PCell.

It is important to follow the above syntax carefully - do not add any whitespace to the PCell argument list. All points must have an X and Y coordinate.

4.1.4 Using Python PCells

With the PCell code created, it should be saved to a file e.g. nmos.py - the .py extension is required, and the name of the file, like the name of the function, must match the intended cellView name for the PCell. Currently the python PCell files can reside in any directory, provided that directory is included in the PYTHONPATH environment variable. Refer to Python documentation for more details.

Next, in Glade use the New Cell command to create the PCell supermaster.

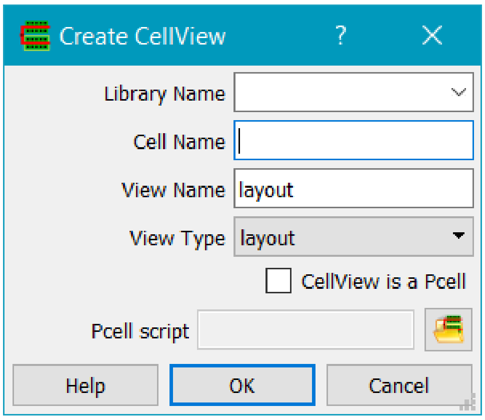


Figure 149 - Create PCell supermaster

Click on *the CellView is a Pcell* button to enable the *Pcell script* field. The file chooser can be used to select the name of the script file. This will create a new cellView for the PCell. Do not edit this cell - it is solely for visual display of the results of the script, using default values for the arguments.

Alternatively, PCells can be loaded into Glade using the ui().loadPcell() command.

To place an instance of a PCell, use the Create Instance command to place the PCell instance.

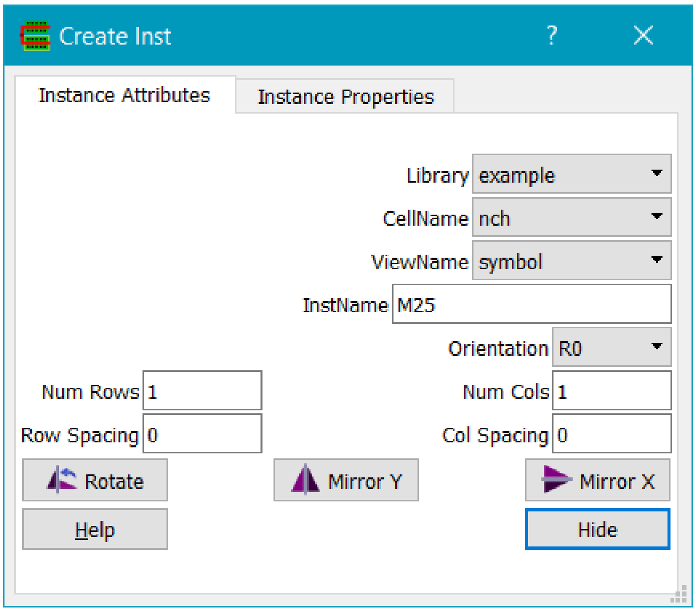


Figure 150 - Create PCell Instance

First set the required PCell parameters. These are stored as properties on the instance, typically as floats.

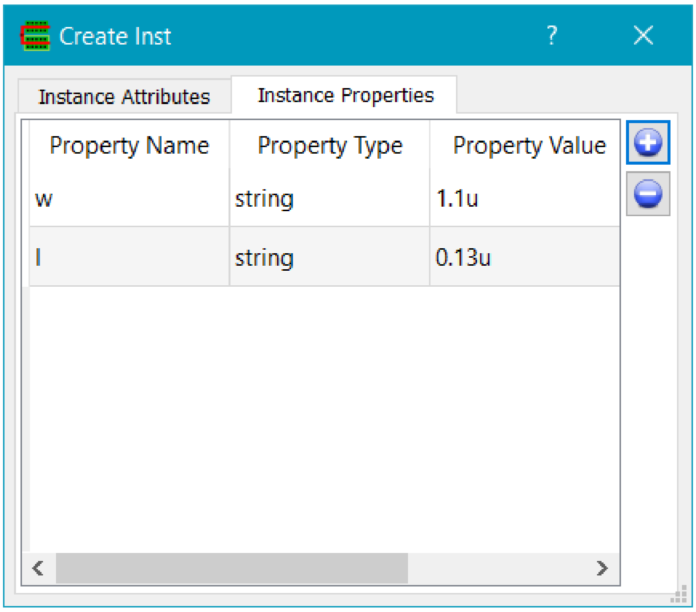


Figure 151 - Setting PCell Instance parameters

You use the Instance Properties tab on the Create Inst dialog to add properties.

To change the parameters of a PCell instance, for example to change its width, select the instance and use the Query Properties dialog to modify the instance's properties. The Pcell will be updated accordingly. Each instance of a PCell will create a superMaster cell - this cell is names according to the PCell name, concatenated with the PCell parameters e.g. nmos$w:1.000000$$l:0.180000$

4.1.5 Loading PCells using Python

PCells can be loaded in Python code, and instances of PCells can be created and their properties changed. For example:

```
gui = cvar.uiptr
gui.importTech("default", "example.tch")
gui.loadPCell("default", "nch”)
lib = getLibByName("default")
cv = lib.dbOpenCellView("test", "layout", 'w')
origin = Point(0,0)
i = cv.dbCreatePCellInst("default", "nch", "layout", origin)
i.dbReplaceProp("w", 2.50e-6)
cv.dbUpdatePCell(i);
cv.update()
```

In the above, we create a library 'default' and load a PCell called ‘nch’ into the library. We then create a cellView called 'test' and create a PCell instance in that cellView. Next we change the value of the property 'w' to 2.5e-6. After changing any property or properties, we need to call dbUpdatePCell(), giving it the PCell instance as the argument. Lastly the cellView is updated in the database.

4.1.6 PCell Python API

See the cellView python bindings.

5 Symbol Creation

5.1.1 Selection Box

Symbols should have a selection box – a rectangle on the 'boundary' 'drawing' layer. This rectangle defines the selection area when the symbol is placed in a schematic, rather than the bounding box of the symbol (which may be large, for example if a lengthy text label is present). The selection box is also used for dynamic highlighting in schematics.

5.1.2 Symbol Properties

Symbols need additional properties for netlisting.

"**type**" (string property), which can be one of the following:

- "mos" : a MOS device (NMOS, PMOS etc) corresponding to a Spice M element.
- "res" : a resistor, corresponding to a Spice R element.
- "cap" : a capacitor, corresponding to a Spice C element.
- "pcap" : a parasitic capacitor, corresponding to a Spice C element.
- "ind" : an inductor, corresponding to a Spice L element.
- "dio" : a diode, corresponding to a Spice D element.
- "bjt" : a bipolar device (NPN, PNP) corresponding to a Spice Q element.
- "fet" : a jfet, corresponding to a Spice J element.
- "pin" : a pin. The device is a pin instance.

If no "type" property is present, then the device is assumed to be a hierarchical element corresponding to a Spice X subcircuit call.

"**NLPDeviceFormat**" (string property). See below for NLP parser syntax.

"**modelName**" (string property) : a device model name associated with this device.

5.1.3 Pins

Symbols require pins, created using the **Create->Pin…** command.

5.1.4 Labels and NLP expressions

5.1.4.1 NLP syntax

NLP (Net List Property) syntax is used for labels with type NLPLabel, and for the hierarchical netlister.

An NLP expression is enclosed in square brackets. An NLP label can consist of multiple expressions and other text, which is copied literally. Expressions must be delimited by whitespace. To add special characters into an NLP expression, use backquoting. Currently \[(left square bracket), \] (right square bracket), \s (space) and \n (newline) are supported.

- [@instName] evaluates to the name of the instance.
- [@libName] evaluates to the name of the instance library.
- [@cellName] evaluates to the name of the instance cell master.
- [@viewName] evaluates to the name of the instance master view.

- [@modelName] evaluates to the value of the instance master property 'modelName'.
- [@elementNum] evaluates to the number of the instance, if the instance name is of the form <char><digits> (which is the default for instance creation)
- [@someName] evaluates to the value of the property 'someName' on the instance. If the property is not found on the instance, then the instance master is checked for the property.

An expression can have formatting information about the property. The syntax is [@<propName>:<prefix>%<suffix>:<defaultValue>].

For example [@w:w=%u] with an instance property w of value 1.0 will evaluate to 'w=1.0u'. [@w:w=%u:w=2.2u] with no property w will evaluate to 'w=2.2u'. If a defaultValue is not given then the property will evaluate to a null string.

A linefeed character can be inserted into a NLP label expression using the sequence \n. For example:

```
[@w:w=%u\n:] [@l:l=%u\n:]
```

If the instance has properties w, l e.g. w=6u l=1u then the resulting display will be:

```
w=6u
l=6u
```

5.1.4.2 NLPDeviceFormat properties

A property with the name NLPDeviceFormat is used to control the schematic netlister. A NLPDeviceFormat property on a symbol is a whitespace delimited sequence:

- <string><expression> <string> <expression>...
- <string> is an arbitrary string of zero or more characters. Backquoted characters \n, \[, \] are treated as a newline character and literal [or].
- <expression> is a NLP expression enclosed in square brackets and can be of the form:
- [|<pinName>:%:<default>] where <pinName> is replaced by the name of the net connecting to the instance pin of an instance of the symbol. If the instance does not have an instance pin, then the expression evaluates to <default>
- [@<propName>:<string>%:<default>] as for NLP labels.

For example an nmos device may have a NLPDeviceFormat property of:

```
M[@elementNum] [|D:%] [|G:%] [|S:%] [|B:%:gnd!] [@modelName] [@w:w=%u:w=2.0u]
[@l:l=%u:l=0.13u] [@m:m=%]
```

An extraction PCell for a mos device may have a NLPDeviceFormat property of:

```
[@instName] [|D:%] [|G:%] [|S:%] [|B:%] [@modelName] [@w:w=%] [@l:l=%]
```

Note that in the above, default values for the w and l properties are not specified, as the extraction PCell will always have a value for these properties.

6 Schematic Creation

A library of symbols must exist in order to place and wire devices in the schematic. A library of simple pins and power/ground symbols is provided in the 'basic' library. This is automatically loaded when Glade starts. The 'basic' library is required by the Create Pin command in schematics.

Schematic entry and editing does not require any specific technology file information - schematics use predefined system layers. For portability, it is recommended that the user does not use non-system layers in schematics or symbols.

The typical steps involved in creating a schematic are as follows:

- Enter devices using the Create->Instance command.
- Add pins for external connections using the Create->Pin command.
- Add wires using the Create->Wire command.
- Add solder dots, if required and not already added when creating wires.
- Add wire labels if required.
- Run the Check command to extract the circuit connectivity and check the schematic for connectivity errors.
- Output a (hierarchical) netlist for simulation or LVS.

6.1.1 Wiring

Schematics are wired using the Create->Wire… command. Note that it is not necessary to enter net names during wiring; connectivity is created when running the Check command.

6.1.2 Checking

Schematics need to be checked and saved before netlisting. The netlisters will compare the lastExtracted property with the last modified property, and give an error if the schematic has not been checked more recently than the last modification.

6.1.3 Netlisting, Switch and Stop Lists

7 Programming in Python

The entire Glade database and much of the UI is wrapped in Python using SWIG. This means you can write Python scripts to automate tasks - PCells (parameterised cells) are a good example.

You can enter python commands directly at the command line. Some useful ones:

```
getSelectedSet()
```

Returns a python list of the selected objects. You can print information about an object using the print command:

```
objs = getSelectedSet()
for obj in objs :
```

```
print obj
```

To get the current cellView displayed in the gui, use:

```
cv = getEditCellView()
```

To access an open library, use:

```
lib = getLibByName("myLib")
```

To open a cellView, use:

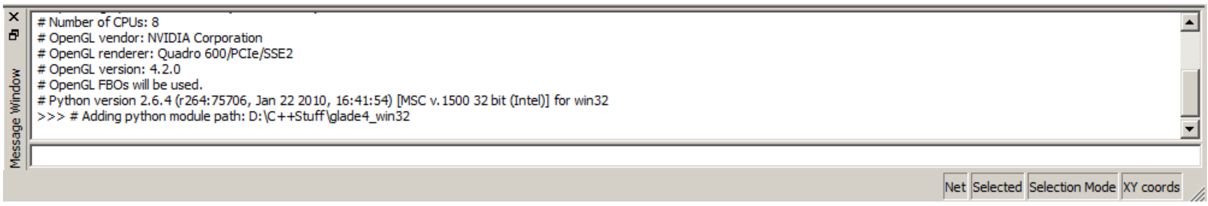
```
# 'r' opens an existing cell for read, 'a' opens an existing cell for edit, 'w' creates a new cell.
cv = lib.dbOpenCellView("myCell", "layout", 'a')
```

Some python bindings require arrays of coordinates. You can use the python intarray(number_of_elements) function to create an array with a specified size. Or you can pass a python list, with each list element being a list of x and Y coordinates:

```
[ [0, 0], [1000, 0], [1000, 1000], [0, 1000] ]
```

7.1 The command line interpreter

The message window at the bottom of the Glade main window is split into two parts: the message pane, which shows messages and output from the Python interpreter. You can use the Right Mouse Button to copy text from the message pane. Below the message pane is the command line. You can type Python commands into the command line.



The Python command line supports various control characters to assist in typing in Python commands:

- Left Arrow - move the cursor one character left.
- Right Arrow - move the cursor one character right.
- Up arrow - retrieve previous command (or clear line if no previous command)
- Down arrow - retrieve next command (or clear line if no next command)
- Home - move the cursor to the start of the line
- End - move the cursor to the end of the line
- Ctrl-A - select all text on the line
- Ctrl-C - copy the selected text to the clipboard
- Ctrl-V - paste the clipboard to the line
- Ctrl-X - delete the selected text
- Ctrl-Z - undo the last editing operation

- Ctrl-Y - redo the last editing operation
-

7.2 Writing Python scripts

An example of a Python script follows.. Don't forget that Python relies on indentation for e.g. for and while loops!

```
# Example python script
print 'Starting script...'
#
# Create a new library, called 'fred'
lib = library("fred")
#
# Create a new cellView in this library
cv = lib.dbOpenCellView("test", "layout", 'w')
#
# A rectangle. By default database units are 0.001 micron
width = 10000
pitch = width * 2
r = Rect(0, 0, 0, 0)
#
# Create four rectangles on layer 1
layer = 1
for i in range(2) :
    for j in range(2) :
        r.setLeft(j * pitch)
        r.setRight(j * pitch + width)
        r.setBottom(i * pitch)
        r.setTop(i * pitch + width)
        cv.dbCreateRect(r, layer);
#
# Update the cellView after creating any objects
cv.update()
#
# Open the cellView for display
ui().openCellView("fred", "test", "layout")
#
# Do a region query
q = cv.bBox()
objs = cv.dbGetOverlaps(q, layer)
obj = objs.first()
while obj :
    print 'found object ', obj.objName(), ' with origin = (', obj.left(), obj.bottom(), ')'
    obj = objs.next()
#
print 'Finished script...'
```

7.3 Python API

Python does not have type declarators like C/C++, so in this documentation the type of an argument is give in its C++ notation, such as int, float, char* etc.

7.3.1 arc class

An arc is a portion of an ellipse and is derived from an ellipse. It is normally created by the cellView::dbCreateArc() function.

a.setStartAngle(double angle)

Sets the arc start *angle* in degrees. Zero degrees is at the 3 o'clock position with respect to the centre of the arc..

double angle = a.startAngle()

Gets the arc start angle.

a.setSpanAngle(double angle)

Sets the arc span *angle* in degrees. The span is the angle from the start to end point of the arc.

double angle = a.spanAngle()

Gets the arc span angle.

dbtype_t_t type = a.objType()

Gets the arc object type, ARC.

char *name = a.objName()

Gets the arc object name as "ARC".

Bool ok = a.offGrid(int grid)

Returns True if the radius of the arc is offgrid.

a.transform(transform trans)

Transforms this arc by *trans*.

a.Move(cellView dest, Point delta, bool opt=True)

Move the arc origin by *delta* in the cellView given by *dest*. If *opt* is true then the database is re-optimised for the new array position. If there are a lot of objects to move it makes sense to turn this off and instead use the cellView update() function after moving them all.

dbObj obj = a.Copy(cellView dest, Point delta, int layer=-1)

Copy the arc. *dest* is the destination cellView, *delta* is the offset from the current origin. If *layer* is non-negative, the arc is copied to the layer number.

dbObjList<dbObj> objs = a.Flatten(cellView dest, transform trans, bool hier=True)

Flatten the arc into the cellView *dest*, with the given transform *trans*, and return a list of the flattened objects.

7.3.2 array class

An array is a reference to an array of cellViews, in another cellView. Arrays correspond to GDS2 AREFs. Arrays are created using the dbCreateArray cellView function. An array is derived from the inst class.

a.numRows(int rows)

Set the number of rows *rows* of this array.

int rows = a.rows()

Get the number of rows for this array.

a.numCols(int cols)

Set the number of columns *cols* for this array.

int cols = a.cols()

Get the number of columns for this array.

a.rowSpacing(int spacing)

Set the row *spacing*. This can be positive or negative.

int spacing = a.rowSpacing()

Get the row spacing.

a.colSpacing(int spacing)

set the column *spacing* for this array. This can be positive or negative.

int spacing = a.colSpacing()

Get the column spacing.

int coord = a.left()

Get the left edge of the array's bounding box.

int coord = a.bottom()

Get the bottom edge of the array's bounding box.

int coord = a.right()

Get the right edge of the array's bounding box.

int coord = a.top()

Get the top edge of the array's bounding box.

bool ok = a.offGrid(int grid)

Checks if an array origin is on the grid grid, which is in database units.

a.orient(orient_t orient)

Set the array orientation. *orient* can be one of: R0, R90, R180, R270, MX, MXR90, MY, MYR90.

orient_t orient = a.orient ()

Get the array orientation.

a.bound(bool b)

Set the array binding. This should probably not be set by the user.

bool status = a.bound()

Get the instance binding status. An array is bound if it references a valid master.

double m = a.mag()

Get the array magnification. Magnifications other than 1.0 are supported, but their use is deprecated.

char *s = a.libName()

Get the array's lib name.

library lib = a.lib()

Get the array's library pointer.

char *s = a.cellName()

Get the array's master cell name.

a.cellName(char *s)

Set the array master's cellName.

char *s = a.viewName()

Get the array's view name.

a.instName(cellView cv, char *instName)

Set the array's *instName*. *cv* is the cellView containing the instance.

char * a.instName()

Get the array's instName.

cellView cv = a.getMaster()

Get the cellview of the array's master.

a.setMaster(cellView cv)

Set the array's master cellView.

Point p = a.origin()

Get the origin of the array. Note that an array's origin does not have to be e.g. the lower left of its bounding box - it can be anywhere.

a.origin(Point p)

a.origin(int x, int y)

Set the origin of the array.

Rect box = a.bBox()

Get the array's bounding box.

dbtype_t_t type = a.objType()

Returns the objects type as ARRAY

char *name = a.objName()

Returns the object name i.e. "ARRAY"

int a.getNearestEdge(Point &p, segment &edge)

Returns the distance to the nearest edge of this object to a Point *p*. *edge* is the nearest segment.

a.transform(transform trans)

Transform the array by the given transform *trans*.

a.scale(double scalefactor, double grid)

Scale the array origin coordinates by *scalefactor*, snapping to *grid*.

a.Move(cellView dest, Point delta, bool opt=True)

Move the array origin in cellView *dest* by *delta*. If *opt* is true then the database is re-optimised for the new array position. If there are a lot of objects to move it makes sense to turn this off and instead use the cellView update() function after moving them all.

dbObj obj = a.Copy(cellView dest, Point delta)

Copy the array. *dest* is the destination cellView, *delta* is the offset from the current origin.

dbObjList<dbObj> objs = a.Flatten(cellView dest, transform trans)

Flatten the array into the cellView *dest*, with the given transform *trans*, and return a list of the flattened objects.

instPin ip = a.dbCreateInstPin(net n, char *name)

Create an instance pin on this array for the net *n* and pin name *name*.

a.dbDeleteInstPin(instPin ip)

Delete the instPin *ip* from this array.

instPin ip = a.dbFindInstPinByName(char *name)

Find the inst pin with name *name* on this array. Returns null if not found.

dbObjList<instPin> instPins = a.getInstPins()

Get a list of all instPins for this array.

int num = a.getNumInstPins()

Get the number of instPins for this array.

---

cell class

The cell class represents a cell, which can have multiple representations (views). A library contains a list of cells and a list of views. A combination of a unique cell and view is a cellView.

cell c = cell

Creates an cell object.

dbObjList<cellView> list = c.cellViews()

Get a list of the cellViews for this cell.

list = c.getCellViews()

Gets a Python list of the cellViews for this cell.

c.name(char *s)

Sets the cell name.

char *s = c.name()

Gets the cell's name.

c.addCellView(cellView cv)

Adds *cv* to the cell's cellView list.

cellView cv = c.dbFindCellViewByView(char *viewName)

Finds the cellView for this cell with view name *viewName*. If it does not exist, a null pointer is returned.

dbtype_t type = c.objType()

Returns the object's type (CELL).

char *name = c.objName()

Returns the object's print name ("CELL").

---

cellView class

A cellView stores design data. It is a unique combination of a cell and a view. CellViews correspond to GDS2 STRUCTs, LEF MACROs or a DEF DESIGN. CellViews are stored in a library. CellView access functions are as follows. Note that all coordinate values are expected in database units. To find the number of database units per micron, use the library::dbuPerUU() function.

7.3.2.1 Creating or opening cellViews

cellView cv = lib.dbOpenCellView(char *cellName, char *viewName, 'mode')

Create a cellView in an existing library lib with cell *cellName* and view *viewName*. The function returns a cellview. *mode* is a single character denoting the access mode; 'r' signifies readonly access, 'w' signifies write access (the cellview should not already exist and will be created), and 'a' signifies append access (the cellview already exists and is opened for modification). Note that after creating a new cellView and any objects in it, update() must be called to build the data structures before editing/viewing/querying.

7.3.2.2 Creating objects in a cellView

A cellView contains shape and instance/array objects. Shape objects are created on a specified layer number, which is internally represented by a signed 16 bit integer value. To get a layer number given a layer name and purpose, you can use the dbTechFile class functions to get and manipulate layers:

```
layer = tech.getLayerNum( layerName, purposeName)
```

rectangle r = cv.dbCreateRect(Rect box, int layer, bool use_rect = False)

Creates a rectangle object in the cellView with bounding box *box* and on layer *layer* and returns the created rectangle. If *use_rect* is false (the default), a square will be created instead of a rectangle if the box width equals the box height.

For example to create a rectangle on layer 3:

```
box = Rect(0, 0, 1000, 2000)
layer = 3
cv.dbCreateRect(box, layer
```

polygon p = cv.dbCreatePolygon(int * xpts, int *ypts, int numPoints, int layer, bool use_poly = False)

Create a polygon object in the cellView and returns the polygon created. The array *xpts* and *ypts* are the X and Y coordinates of the polygon and should be created in python using the intarray() function. *numPoints* specifies the number of points and *layer* the layer the polygon is created on. If *use_poly* is False (the default), a rectangle will be created instead of a polygon if possible.

For example to create a triangle on layer 3:

```
numPoints = 3
x = intarray(numPoints)
y = intarray(numPoints)
x[0] = 0
y[0] = 0
x[1] = 2000
y[1] = 0
x[2] = 0
y[2] = 2000
layer = 3
poly = cv.dbCreatePolygon(x, y, numPoints, layer)
```

polygon p = cv.dbCreatePolygon(ptlist, int numPoints, int layer, bool use_poly = False)

Similar to the above, but uses a python list of points, each of which is a list of x and y coordinates of the point.

```
poly = cv.dbCreatePolygon([[0,0],[1000,0],[1000,1000],[0,1000]], 4, 3)
```

polygon p = cv.dbCreateHole(int *xpts, int *ypts, int nPoints, int lyr, shape *obj=NULL)

Creates a hole in a polygon . The hole to be 'cut' is represented by the poygon defined by *xpts*, *ypts*, *nPoints*. The polygon and hole are on layer *lyr*. If *obj* is non-null, it is assumed to be the polygon to cut the hole in; if non-null, the largest polygon that overlaps the hole will be cut. In general, it is better to use the boolean operations to cut holes in polygons as they are more robust.

label l = cv.dbCreateLabel(Point origin, char *name, int orient, double height, int presentation, int layer)

Creates a label in the cellView at location *origin* with text *name* and returns the label created. The orientation of the label is given by *orient* and the label height by *height*. *presentation* is the alignment of the text label and *layer* is the label's layer.

path p = cv.dbCreatePath(int *xpts, int *ypts, int numPoints, int layer, int width, int style, int beginExtent, int endExtent)

Create a path object in the cellView and returns the path created. The array *xpts* and *ypts* are the X and Y coordinates of the path. *numPoints* specifies the number of points and *layer* the layer the polygon is created on. *width* is the width of the path and *style* the path style (0 = TRUNCATE, 1 = ROUND, 2 = EXTEND, 4 = VAREXTEND, 8 = OCTAGONAL). If the path style is type 4, varExtend, then *beginExtent* and *endExtent* specify the path extension beyond the beginning and ending points.

path p = cv.dbCreatePath(ptlist, int numPoints, int layer, int width, int style, int beginExtent, int endExtent)

Create a path object in the cellView and returns the path created. The python ptlist is a list of points, each of which is a list of x and y coordinates of the point. *numPoints* specifies the number of points and layer the layer the polygon is created on. *width* is the width of the path and *style* the path style (0 = TRUNCATE, 1 = ROUND, 2 = EXTEND, 4 = VAREXTEND, 8 = OCTAGONAL). If the path style is type 4, varExtend, then *beginExtent* and *endExtent* specify the path extension beyond the beginning and ending points.

mpp m = cv. dbCreateMPP(inst *xpts, int *ypts, int nPoints)

Creates a mpp (MultiPartPath) in the cellView and returns the mpp created. The array *xpts* and *ypts* are the X and Y coordinates of the path. *numPoints* specifies the number of points.

mpp m = cv. dbCreateMPP(ptlist, int nPoints)

Creates a mpp (MultiPartPath) in the cellView and returns the mpp created. The python list pts is the coordinates of the path. *nPoints* specifies the number of points.

```
poly = cv.dbCreateMPP([[0,0],[1000,0],[1000,1000],[0,1000]], 4)
```

mpp m = cv. dbCreateMPP(char *ruleName, ptlist, int nPoints)

Creates a mpp (MultiPartPath) in the cellView using the specified rule *ruleName* and returns the mpp created. The python list *ptlist* is the coordinates of the path. *nPoints* specifies the number of points. *ruleName* is the name of the (existing) mpp rule, as defined in the techFile.

inst i = cv.dbCreateInst(char *libName, char *cellName, char *viewName, Point origin, orient_t orient, double mag, char *instName=null)

Create an inst in the cellView and returns the instance created. The instance master cellView is specified by *libName*/cellName/*viewName*. The instance's origin is given by *origin* and its orientation by *orient*. The enumerations R0, R90, R180, R270, MX, MXR90, MY, MYR90 can be used to specify the orientation. Orientations other than variants of 90 degrees are not currently supported. The magnification is specified by *mag*. If specified, *instName* is used to name the instance; else the instance name is autogenerated with the first being I0, then I1, I2 etc.

inst i = cv.dbCreateInst(library lib, char *cellName, char *viewName, Point origin, orient_t orient, double mag, char *instName=null)

Create an inst in the cellView and returns the instance created. This is identical to the above but takes a library, rather than a library name, as argument. The instance master cellView is specified by *cellName*/*viewName*. The instance's origin is given by *origin* and its orientation by *orient*. The enumerations R0, R90, R180, R270, MX, MXR90, MY, MYR90 can be used to specify the orientation. Orientations other than variants of 90 degrees are not currently supported. The magnification is specified by *mag*. If specified, *instName* is used to name the instance; else the instance name is autogenerated with the first being I0, then I1, I2 etc.

inst i = cv.dbFindInstByName(char *name)

Finds the instance with name *name* in the cellView and returns it, or null if not found.

array a = cv.dbCreateArray(char *libName, char *cellName, char *viewName, Point origin, orient_t orient, double mag, int numRows, int numCols, int rowSpacing, int colSpacing, char *instName=null)

Create an array in the cellView and returns the array created. The array master cellView is specified by *libName*, *cellName* and *viewName*. The array's origin is given by *origin* and its orientation by *orient*. The enumerations R0, R90, R180, R270, MX, MXR90, MY, MYR90 can be used tp specify the orientation. Orientations other than variants of 90 degrees are not supported. The magnification is specified by *mag*. If specified, *instName* is used to name the instance; else the instance name is autogenerated with the first being I0, then I1, I2 etc. *numRows* specifies the number of rows and must be greater than 0. *numCols* specifies the number of columns and must be greater than 0. *rowSpacing* is the spacing between rows and can be negative or positive, as can *colSpacing*.

array a = cv.dbCreateArray(librarylib, char *cellName, char *viewName, Point origin, orient_t orient, double mag, int numRows, int numCols, int rowSpacing, int colSpacing, char *instName=null)

Create an array in the cellView and returns the array created. This is identical to the above but takes a library *, rather than a library name, as argument. The array master cellView is specified by *libName*, *cellName* and *viewName*. The array's origin is given by *origin* and its orientation by orient. The enumerations R0, R90, R180, R270, MX, MXR90, MY, MYR90 can be used tp specify the orientation. Orientations other than variants of 90 degrees are not supported. The magnification is specified by *mag*. If specified, *instName* is used to name the instance; else the instance name is autogenerated with the first being I0, then I1, I2 etc. *numRows* specifies the number of rows and must be greater than 0. *numCols* specifies the number of columns and must be greater than 0. *rowSpacing* is the spacing between rows and can be negative or positive, as can *colSpacing*.

viaInst v = cv.dbCreateViaInst(char *name, Point origin, orient_t orient = R0)

Creates a viaInst of a via with master *name*, origin *origin* and oriention *orient* and returns the viaInst created.

HSeg h = cv.dbCreateHSeg(int x1, int y1, int x2, int y2, int layer, net n, int width=0, int style=DB_TRUNCATED)

A HSeg is a horizontal track segment. HSegs are a memory efficient way of representing a two point path with a given layer that has a fixed width and style, and as such are used in representing DEF regular net routing. This function creates a HSeg object in the cellView and returns the HSeg created. (*x1, y1*) is the first point of the HSeg, (*x2, y2*) is the second point. *layer* is the layer the HSeg is created on. *width* is the HSeg width (defaults to 0) and *style* is the HSeg's path style (defaults to truncated). If the cellView's library does not contain a segparam index for the HSeg with matching layer and width/style, one is created.

VSeg v = cv.dbCreateVSeg(int x1, int y1, int x2, int y2, int layer, net *n, int width=0, int style=DB_TRUNCATED)

A VSeg is a vertical track segment. VSegs are a memory efficient way of representing a two point path with a given layer that has a fixed width and style, and as such are used in representing DEF regular net routing. This function creates a VSeg object in the cellView and returns the VSeg created. (*x1, y1*) is the first point of the VSeg, (*x2, y2*) is the second point. *layer* is the layer the VSeg is created on. *width* is the VSeg width (defaults to 0) and *style* is the VSeg's path style (defaults to truncated). If the cellView's library does not contain a segparam entry for the VSeg, one will be created with matching layer and width/style, one is created.

ellipse **e = cv.dbCreateEllipse(Point origin, int xRadius, int yRadius, int layer)**

Create an ellipse with given *origin* (the centre of the ellipse), *xRadius*, *yRadius* and *layer* number.

ellipse **e = cv.dbCreateCircle(Point origin, int radius, int layer)**

Create a circular ellipse, i.e. one with the same X and Y radius.

arc **a = cv.dbCreateArc(Point origin, int xRadius, int yRadius, double startAngle, double spanAngle, int layer)**

Create an arc with the specified *origin*, *Xradius* and *Yradius* on *layer*. The arc is part of an ellipse with the specified *startAngle* and *spanAngle*. *startAngle* is the angle the arc starts on. Zero degrees corresponds to 3 o'clock. *spanAngle* is the angle increment from the *startAngle*.

line l = cv.dbCreateLine(Point p1, Point p2, int layer)

Creates a line in the cellView with vertices defined by points *p1* and *p2* on layer *layer* and returns the line created.

line l = cv.dbCreateLine(int *x, int *y, int numPoints, int layer)

Creates a line in the cellView with vertices defined by integer arrays *x* and *y* with size *numPoints* on layer *layer* and returns the line created.

line l = cv.dbCreateLine(ptlist, int numPoints, int layer)

Creates a line in the cellView with vertices defined by the python list *ptlist*, which is a list of points. Each point is a list of x and y coordinates. *numPoints* is the number of points. The line is created on layer *layer*.

7.3.2.3 Creating connectivity in a cellView

A cellView can also contain connectivity, such as nets, pins and ports (physical pin shapes).

net = cv.dbCreateNet(char *name)

Creates a net in the cellView with name *name* and returns the net created. If the net already exists in the cellView, the net is not created.

net n = cv.dbFindNetByName(char *name)

Finds the net with name *name* in the cellView and returns it, or null if not found.

pin p = cv.dbCreatePin(char *name, db_PinDirection dir, net n)

Creates a logical pin in the cellView with name *name* and direction *dir* for the net *n* and returns the pin created. The net *n* must exist in the cellView.

pin p = cv.dbFindPinByName(char *name)

Finds the pin with name *name* in the cellView and returns it, or null if not found.

cv.dbCreatePort(pin p, shape shp)

Creates a port for a pin *p*. A port is a physical representation of a pin so a valid shape *shp* must be specified.

7.3.2.4 Creating and updating PCell instances in a cellView

PCell (programmable cell) instances can be created in a cellView. See also loadPCell.

inst i = cv.dbCreatePCellInst(char *libName, char *cellName, char *viewName, Point origin, int orient=R0, int numRows=1, int numCols=1, int rowSpacing=0, int colSpacing=0)

Create an instance of a PCell in the cellView and returns the instance created. The PCell master must have been previously created e.g. by a call to ui::loadPCell(). *libName* is the library name containing the pcell, *cellName* is the cellView name of the PCell and *viewName* is the view name of the PCell. *origin* is the instance's origin. If specified, *orient* is the instance's orientation, otherwise defaulting to R0. If *numRows* or *numCols* are not 1, an array is created of PCells.

inst newInst = cv.dbUpdatePCell(inst originalInst)

Updates a PCell instance after any of its properties have been changed. This is equivalent to querying the PCell instance properties in the GUI and changing them. Note that the *originalInst* is destroyed, and *newInst* is created.

7.3.2.5 Searching for objects in a cellView

dbObjList<dbObj> list = cv.dbGetOverlaps(Rect box, int layer, bool allLayers=False, bool instsToo=False, bool viaInstsToo=False)

Searches the area given by *box* for any objects whose bounding boxes overlap the area. If *allLayers* is 0, then shapes on only the specified *layer* are returned. If *allLayers* is 1, shapes on all layers are searched. If *instsToo* is 1, any instances whose bounding box overlaps the area are returned in addition to any valid shapes, similarly is *viaInstsToo* is 1 then any via insts that overlap are also checked.

cv.dbGetOverlaps(dbObjList<dbObj> &list, Rect box, int layer, bool allLayers=False, bool instsToo=False, bool viaInstsToo=False)

As dbGetOverlaps, but appends objects found to list.

list = cv.getOverlaps(Rect box, int layer, bool allLayers=False, bool instsToo=False, bool viaInstsToo=False)

As above, but returns a Python list of dbObjs.

cv.dbGetHierOverlaps(dbObjList<dbHierObj> &list , Rect box, int layer, bool allLayers = False, int level = 99)

Searches the area given by *box* for any objects whose bounding boxes overlap the area. If *allLayers* is 0, only the shapes on the specified *layer* are returned. If *allLayers* is 1, shapes on all layers are searched. The search is carried out hierarchically up to level levels deep.

A dbHierObj is a simple class containing the object itself, the cellView containing the object and the transform of the object relative to the top level.

list = cv.getHierOverlaps(Rect box, int layer, bool allLayers = False, int level = 99)

As above, but returns a Python list of dbHierObjs .

7.3.2.6 cellView utility functions

UserUnits units = cv.userUnits()

Returns the user units as *inches* or *microns*.

cv.userUnits(units)

Sets the user units. *units* can be either *inches* or *microns*.

int dbu = cv.dbuPerUU()

Returns the number of database units per user unit. The default number of dbu is 1000 if user units are *microns*, and 160 if user units are *inches*.

cv.dbuPerUU(dbu)

Sets the database units per user unit.

cv.updateBbox()

Updates the cellView's bounding box to enclose all objects it contains. This function is deprecated and update() should be used.

cv.optimiseTrees()

Build the internal data structures for the cellView, or updates them. This must be called after creating any objects in a new cellView, but before viewing / editing / querying the cellView. This function is deprecated and update() should be used.

cv.update()

Calls updateBbox(), optimiseTrees(), sets the cellView as edited and sets the modification date. This should be called after a modification, or a set of modifications, to the cellView. For perfomance reasons it is better to call this after a set of operations rather than for each operation.

Rect box = cv.bBox()

Get the bounding box of the cellView as a Rect.

cv.clearBbox()

Resets the cellView bounding box to (0,0) (0,0).

cv.bBox(Rect box)

The existing cellView bounding box becomes the union of the current bounding box and *box*.

Rect box = cv.getBoundary()

Gets the cellView boundary rectangle, if such a shape exists on the boundary drawing layer.

cv.dbDeleteObj(dbObj object, bool reallyDelete=True, bool opt=True)

Delete the database object *object*. If *reallyDelete* is true, the object is deleted, else it is just removed from the object trees (and hence undoing the delete is possible). If *opt* is true, the tree is (re)optimised after the delete.

int num = cv.getNumShapes()

Get the number of shapes in the cellView.

int num = cv.getNumInsts()

Get the number of instances in the cellView.

int num = cv.getNumViaInsts()

Get the number of viaInsts in the cellView.

int num = cv.getNumNets()

Get the number of nets in the cellView.

int num = cv.getNumPins()

Get the number of pins in the cellView.

library lib = cv.lib()

Get the cellView ‘s library.

bool val = cv.isPCell()

returns true if the cellView is a PCell superMaster.

bool val = cv.isSubMaster()

Returns true if the cellView is a PCell subMaster.

char *s = cv.cellName()

Get the cellView's name.

char *s = cv.viewName()

Get the cellView's viewname.

dbObj obj = cv.getNearestObj(Point p, int dist)

Get the nearest object to a point *p* in the cellView , up to a maximum distance *dist*.

lpp l = cv.getLpp(int layer)

Get the layer-purpose pair with layer number *layer* in this cellView.

bool b=cv.deleteLpp(lpp l)

Delete the layer-purpose pair *l* in this cellView. All objects (shapes, insts and viaInsts) on that lpp will be deleted.

list = cv.getLpps()

Returns a Python list of all layer-purpose pairs in the cellView. This is a python wrapper created using the SWIG %extend function.

list = cv.getInsts()

Returns a Python list of all instances in the cellView. This is a python wrapper created using the SWIG %extend function.

list = cv.getNets()

Returns a Python list of all nets in the cellView. This is a python wrapper created using the SWIG %extend function.

list = cv.getPins()

Returns a Python list of all pins in the cellView. This is a python wrapper created using the SWIG %extend function.

shape new_shp = cv.roundCorners(shape shp, int inner_radius, int outer_radius, int segs, double grid)

Rounds the shape *shp* with the radius given in dbu, using a minimum number of segments segs, and snaps the vertices of the curve to grid in microns. *inner_radius* is the radius of inner (concave) corners; *outer_radius* is the radius of outer (convex) corners.

7.3.2.7 Iterators

Instead of using getInsts/getNets/getPins/getLpps it is possible to use iterators in Python:

iter = instIterator(cellView cv)

Initialises the inst iterator for the cellView. For example:

```
cv = getEditCellView()
iter = instIterator(cv)
while not iter.end() :
    inst = iter.value()
    name = inst.instName()
    print "inst name = ", name
    iter.next()
```

iter.next()

Advances the iterator to the next instance.

bool iter.end()

Returns false if there are more instances, else returns true if there are no more.

inst = iter.value()

Returns the current instance.

iter = netIterator(cellView cv)

Initialises the net iterator for the cellView. The iterator has similar next(), end() and value() functions as above.

iter = pinIterator(cellView cv)

Initialises the pin iterator for the cellView. The iterator has similar next(), end() and value() functions as above.

iter = lppIterator(cellView cv)

Initialises the lpp iterator for the cellView. The iterator has similar next(), end() and value() functions as above.

7.3.3 dbObj class

The dbObj class is the base class of Glade database objects (it is derived from a lower level memory allocation class which caches objects, but the user need not be concerned about that). A dbObj is never created directly. Most access to dbObjs is at the derived class level.

dbtype_t_t type = obj.objType()

Returns the type of an object. This may be one of the following.

```
ARC
ARRAY
CELL
CELLVIEW
ELLIPSE
HSEG
INST
LINE
MPP
PATH
POLYGON
NET
PIN
RECTANGLE
SEGMENT
TEXT
VERTEX
VSEG
VIAINST
VIEW
```

char *name = obj.objName()

Returns the print name of an object.

bool val = obj.isInst()

Returns true if the object is an inst or array.

bool val = obj.isShape()

Returns true if the object is a shape.

bool val = obj.isViaInst()

Returns true if the object is a viaInst.

bool val – obj.isSeg()

Returns true if the object is a segment.

bool val = obj.isVertex()

Returns true if the object is a vertex.

bool val = obj.dbFindProp(char *name, bool caseSensitive=True)

Returns true if the object has a property *name*.

propType type = obj.dbFindPropType(char *name)

Returns the type of a property as one of *stringType*, *integerType*, *floatType*, *booleanType*.

obj.dbSetPropVisible(char *name, bool visible)

Sets whether the property is visible in schematics when displayed using a NLPLabel.

Bool val = obj.dbIsPropVisible(char *name)

Returns true if the property is set as visible.

obj.dbAddProp(char *name, char *value)

Adds a property *name* to the object with value *value*. If the property already exists, it is replaced.

obj.dbAddProp(char *name, int value)

Adds a property *name* to the object with value *value*. If the property already exists, it is replaced.

obj.dbAddProp(char *name, float value)

Adds a property *name* to the object with value *value*. If the property already exists, it is replaced.

obj.dbAddProp(char *name, bool value)

Adds a property *name* to the object with value *value*. If the property already exists, it is replaced.

bool val = obj.dbReplaceProp(char *name, char *value)

Returns true if the property *name* exists and is replaced. Else returns false.

bool val = obj.dbReplaceProp(char *name, int value)

Returns true if the property *name* exists and is replaced. Else returns false.

bool val = obj.dbReplaceProp(char *name, float value)

Returns true if the property *name* exists and is replaced. Else returns false.

bool val = obj.dbReplaceProp(char *name, bool value)

Returns true if the property *name* exists and is replaced. Else returns false.

char *val = obj.dbGetStringProp(name, bool caseSensitive=True)

Returns the value of the stringType property *name*.

int val = obj.dbGetIntProp(name, bool caseSensitive=True)

Returns the value of the integerType property *name*.

double val = obj.dbGetFloatProp(name, bool caseSensitive=True)

Returns the value of the floatType property *name*.

bool val = obj.dbGetIntProp(name, bool caseSensitive=True)

Returns the value of the booleanType property *name*.

bool val = obj.dbDeleteProp(char *name)

Returns true if the property *name* is found, if so the property is deleted.

list = obj.getPropList()

Gets the object's property list as a Python list of prop objects. A prop object is a helper class with the following accessor methods:

> **char *name = prop.name()**
>
> Returns the name of the property.
>
> **prop.setName(char *name)**
>
> Sets the name of the property.
>
> **propType type = prop.type()**
>
> Returns the type of the property.
>
> **prop.setType(propType type)**
>
> Sets the type of the property.
>
> **union value = prop.data()**
>
> Returns the property value. value.s is the string data, value.i the integer data, value,f the float data, value.b the boolean data.
>
> **prop.setData(value)**
>
> Sets the property value. The function is overloaded for the common propType types.

Casting a dbObj to a derived class

In Python, there is no means of casting a base class to a derived class. So for example if you use the cellView::dbGetOverlaps() function to get a list of objects, these are returned as dbObj class. So to facilitate conversion, there are a set of functions that convert a dbObj to a derived class e.g. rectangle.

arc a = obj.toArc()

Casts a dbObj to an arc. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

array a = obj.toArray()

Casts a dbObj to an array. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

cell c = obj.toCell()

Casts a dbObj to a cell. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

cellView cv = obj.toCellView()

Casts a dbObj to a cellView. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

ellipse e = obj.toEllipse()

Casts a dbObj to a ellipse. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

HSeg h = obj.toHSeg()

Casts a dbObj to a HSeg. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

inst I = obj.toInst()

Casts a dbObj to a inst. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

label l = obj.toLabel()

Casts a dbObj to a label. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

line l = obj.toLine()

Casts a dbObj to a line. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

path p = obj.toPath()

Casts a dbObj to a path. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

polygon p = obj.toPolygon()

Casts a dbObj to a polygon. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

rectangle r = obj.toRectangle()

Casts a dbObj to a rectangle. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

segment s = obj.toSegment()

Casts a dbObj to a segment. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

viaInst v = obj.toViaInst()

Casts a dbObj to a viaInst. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

vertex v = obj.toVertex()

Casts a dbObj to a vertex. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

VSeg v = obj.toVSeg()

Casts a dbObj to a VSeg. For use with Python e.g to cast the return objects from dbGetOverlaps() which are returned as dbObj types.

7.3.4 dbHierObj class

A dbHierObj is a helper class created in hierarchical searches using dbGetHierOverlaps(). It contains the object, the cellView that contains the object and the transformation of the object from the top level.

hierObj = dbHierObj(cellView cv, dbObj obj, transform trans)

Construct a dbHierObj with the cellView *cv* containing object *obj* and the transformation *trans* as seen from the top level.

bool operator == (dbHierObj other)

True if the two dbHierObjs are equal, i.e. represent the same hierarchical object.

bool operator != (dbHierObj other)

True if the two dbHierObjs are not equal, i.e. are different hierarchical objects.

bool operator < (dbHierObj other)

True if the dbHierObj is 'less than' the *other*. 'Less than' is a rather arbitrary comparison used for sorting. Objects are compared by type, layer, transformation and their pointlist.

dbObj obj = hierObj.object()

Returns the dbObj associated with the dbHierObj.

transform trans = hierObj.transform()

Returns the transform of the dbHierObj.

int layer = hierObj.layer()

Returns the layer of the object.

cellView cv = hierObj.cv()

Returns the cellView that contains the object.

7.3.5 dbObjList class

A dbObjList is a list class containing dbObj objects. It is returned e.g by cellView::dbGetOverlaps()

list.clear()

Clears a dbObjList

int list.size()

Returns the number of objects in a dbObjList

bool list.isEmpty()

Returns True if the dbObjList is empty, i.e. the size is zero.

bool list.member(dbObj obj)

Returns True if the object obj is a member of the list, else False.

list.prepend(dbObj obj)

Inserts the object at the beginning of the list. No list traversal is required.

list.append(dbObj obj)

Inserts the object at the end of the list. No list traversal is required.

list.concat(dbObjList otherlist)

Concatenate the two lists. otherlist is appended to the list, note this is a soft copy and otherlist remains unchanged.

bool list.remove(dbObj obj)

Removes the object from the list. The list size is decremented. Returns True if the dbObj was found in the list, False if not.

dbObj list.pop()

Pops an object from the front of the list; the size of the list is decremented by one.

dbObj obj = list.first()

Returns the first object in the list.

dbObj obj = list.next()

Returns the next object in the list, or null if the end of the list is reached. The iterator is incremented.

dbObj obj = list.peek()

Returns the next object in the list, or null if the end of the list is reached. The iterator is NOT incremented.

dbObj list.last()

Returns the last object in the list, or null if the list is empty.

7.3.5.1 Casting to other types

Because many operations e.g. dbGetOverlaps() as mentioned above return a dbObjList with object as the base class, dbObj , there are swig wrapped C functions to cast to the derived type (you cannot cast in python).

See the dbObj class for a list of all cast functions.

7.3.5.2 Iterator

An iterator to allow traversing the objects in the dbObjList using Python.

iter = objIterator(dbObjList list)

Initialises the dbObj iterator for the dbObjList . For example:

```
objs = cv.dbGetOverlaps(box, layer)
iter = dbObjIterator(objs)
while not iter.end() :
obj = iter.value()
type = obj.objType()
print "object type = ", type
iter.next()
```

dbObj obj = iter.value()

Returns the current object.

iter.next()

Advances the iterator to the next dbObj .

bool iter.end()

Returns False if there are more objects, else returns True if there are no more.

7.3.6 Edge class

The Edge class represents an edge, i.e a connected pair of vertices.

Edge e = Edge

Creates and edge object. The endpoints are undefined.

Edge e = Edge (Point p0, Point p1)

Creates and Edge object and initialises the endpoints.

Edge e = Edge (x0, y0, x1, y1)

Creates and Edge object and initialises the endpoints.

Point p = e.getP0()

Gets one endpoint P0.

Point p = e.getP1()

Gets the other endpoint P1.

e.setP0(Point p)

Sets endpoint P0 to p.

e.setP1(Point p)

Sets endpoint P1 to p.

e.offset(int dx, int dy)

Transposes the edge by the distance specified by *dx, dy*.

bool operator ==

Returns True if the edges are the same i.e. endpoints P0 and P1 are identical.

bool operator !=

Returns True if of the edges are not the same i.e. endpoints P0 and P1 are not identical.

int dist = e.pointToEdge(Point p)

Returns the shortest distance from a point *p* to the edge.

int i = e.length()

Returns the Euclidian length of the edge e.

bool b = e.isHorizontal()

Returns true if the edge is horizontal.

bool b = e.isVertical()

Returns true if the edge is vertical.

bool b = e.isDiagonal()

Returns true if the edge is diagonal.

bool b = e.isOrthogonal()

Returns true if the edge is either horizontal or vertical.

int e.deltax()

Returns the horizontal distance between the edges endpoints i.e. P1-P0.

int e.deltay()

Returns the vertical distance between the edges endpoints i.e. P1-P0.

bool b = e.contains(Point p, bool includeEnds=True)

Returns True if the point *p* lies on the edge e. If *includeEnds* is True, the point p can lie on the endpoints of the edge and be considered 'contained'.

bool b = e.crosses(Rect r, bool touch = True)

Returns True if the edge crosses the Rect *r*, i.e. if the edge intersects one of the Rect 's edges. If *touch* is True, this includes the endpoint of the edge touching an edge of the Rect .

bool b = e.crosses(Point *pts, int numPoints, bool touch = True)

Returns True if the edge crosses the polygon given by pts, i.e. if the edge intersects one of the polygon's edges. If *touch* is True, this includes the endpoint of the edge touching an edge of the polygon.

int dist = e.pointToEdge(Point p)

Returns the distance of a point *p* to the edge.

bool b = e.intersects(Edge other, bool includeEnds=True)

Returns true if the edges intersect at some point. If *includeEnds* is true, returns true if the edges intersect at endpoint(s).

Point p = e.interSectsAt(Edge other)

Returns the point of intersection of two edges. The result is only valid if the edges intersect.

bool b = e.isColinear(Edge other)

Returns true if the edges are colinear, i.e. the edges are parallel and a point of one edge is on the other edge.

bool b = projects(Edge e1, Edge e2, Edge &e3, Edge &e4)

Returns true if the edges are parallel and project onto each other. Edges *e3* and *e4* are the projecting edges.

Point p = nearestPoint(Point pt)

Returns the point on the edge that is nearest the Point *pt*. The point p is either on a line perpendicular to the edge, or if no such line exists, is the nearest endpoint of the edge.

bool b = e.left(Point p)

Returns true if point *p* is to the left of edge e, i.e. 'inside'. Note this assumes the direction of the edge is from endpoint P0 to endpoint P1.

7.3.7 ellipse class

An ellipse is represented by a centre point and an X and Y radius. If X and Y are equal, you have a circle. An ellipse is normally created by the cellView::dbCreateEllipse() or cellView::dbCreateCircle() function.

int x = e.left()

Returns the least X value of the ellipse's bounding box.

int x = e.right()

Returns the greatest X value of the ellipse's bounding box.

int e.bottom()

Returns the lowest Y value of the ellipse's bounding box.

int e.top()

Returns the highest Y value of the ellipse's bounding box.

e.setOrigin(Point origin)

e.setOrigin(int x, int y)

Sets the ellipse's centre point.

int origin = e.origin()

Returns the ellipse's centre point.

int h = e.height()

Returns the height of the ellipse.

int w = e.width()

Returns the width of the ellipse.

Rect e.bBox()

Returns the ellipse's bounding box.

e.setXRadius(int r)

Set the X radius of the ellipse

e.setYRadius(int r)

Set the Y radius of the ellipse.

int r = e.xRadius()

Returns the X radius of the ellipse.

int r = e. yRadius()

Returns the Y radius of the ellipse.

e.setNumChords(int n)

Sets the number of edges that the ellipse will be fractured into when converting to a polygon.

int n = e.numChords()

Get the number of chords for the ellipse.

dbtype_t type = e.objType()

Returns the object's type as ELLIPSE

char *name = e.objName()

Returns the object's name as "ELLIPSE"

double a = e.area()

Returns the ellipse's area.

int p = e.perimeter()

Returns the ellipse's perimeter.

bool ok = e.offGrid(int grid)

Returns true if the ellipse's xRadius or yRadius is offgrid.

e.transform(transform trans)

Transforms the ellipse by *trans*.

e.Move(cellView dest, Point delta, bool opt=True)

Move this ellipse by distance *delta*. If *opt* is True then the database is re-optimised for the new rectangle position. If there are a lot of objects to move it makes sense to turn this off and instead use the cellView update() function after moving them all.

dbObj obj = e.Copy(cellView dest, Point delta, layer=-1)

Copy this ellipse to cellView *dest*, with offset *delta*. If *layer* is non negative the rectangle will be copied to the new layer number.

dbObjList<dbObj> objs = e.Flatten(cellView dest, transform trans, bool hier=True)

Flatten this ellipse into cellView *dest* with transformation *trans*.

e.bias(int bias, int xgrid, int ygrid, int layer=-1)

Bias this ellipse by bias, snapping to the grid *xgrid* and *ygrid*.

e.scale(double scale, int grid)

Scale this ellipse by *scale*, snapping to the grid *grid*.

7.3.8 HSeg class

A HSeg represents a wiring segment for place&route data. It is a 2 vertex horizontal path. A HSeg is normally created by the cellView::dbCreateHSeg() function.

v.setPoints(int x1, int y1, int x2, int y2)

Sets the vertices of the HSeg

v.left()

Gets the leftmost X coordinate of a HSeg.

v.right()

Gets the rightmost X coordinate of a HSeg.

v.bottom()

Gets the lowest Y coordinate of a HSeg.

v.top()

Gets the highest Y coordinate of a HSeg.

int w = v.width()

Gets the HSeg width.

v.setStyle(int s)

Sets the HSeg style, i.e. the type of the path end. The style can be one of: 0 - truncate, 1 - round, 2 - extend, 4 - varextend, 8 - octagonal. Python global variables TRUNCATE, ROUND, EXTEND, VAREXTEND, OCTAGONAL are defined to these values.

int s = v.getStyle()

Gets the HSeg style.

v.setSpecial(bool val)

Sets the HSeg specialNet status

v.isSpecial()

Returns true is the HSeg is a specialNet.

v.setNet(net n)

Sets the HSeg net

v.getNet()

Returns the HSeg net.

Rect **b = v.bBox()**

Get the bounding box of this HSeg.

dbtype_t_t t = v.objType()

Returns the object type of this HSeg as HSEG.

char *name = p.objName()

Returns the object name of this HSeg as "HSEG".

int n = v.nPoints()

Returns the number of points of the HSeg (2).

Point * pts = v.ptlist()

Returns the point list of this HSeg as a C array of Points.

double a = v.area()

Get the area of this HSeg.

int p = v.perimeter()

Get the perimeter of this HSeg.

Point v.origin()

Returns the origin point of a HSeg

v.setOrigin(Point p)

Sets the origin of a HSeg.

int e = v.extent()

Returns the extent, i.e. the length of the HSeg.

v.setExtent(int e)

Sets the extent of the HSeg.

Point p = v.getFirstVertex()

Gets the first vertex of this HSeg

Point p = v.getLastVertex()

Gets the last vertex of this HSeg.

bool v.ptInPoly(Point p)

Returns true if the Point p is contained in the HSeg or on its edges.

v.Move(cellView dest, Point delta, bool opt = True)

Move this HSeg by distance *delta*. If *opt* is True then the database is re-optimised for the new HSeg position. If there are a lot of objects to move it makes sense to turn this off and instead use the cellView update() function after moving them all.

dbObj obj = v.Copy(cellView dest, Point delta, int layerNum = -1)

Copy this HSeg to cellView *dest*, with offset *delta*. If *layer* is a positive integer the HSeg will be copied to the new layer number.

dbObj obj = v.Flatten(cellView dest, transform trans)

Flatten this HSeg into cellView *dest* with transformation *trans*.

int dist = v.getNearestEdge(Point p, segment &edge, bool centreLine=True, bool outLine=True)

Gets the nearest segment *edge* to the HSeg from the Point *p* and returns the distance. If *centreline* is True, the centre line of the HSeg is considered. If *outLine* is True, the outline edges of the HSeg are considered.

int dist = v.getNearestVertex(Point p, vertex &vert)

Gets the nearest vertex *vert* to the HSeg from the Point p and returns the distance.

char *name = v.getNetName()

Returns the HSeg net name as a string.

bool ok = v.offGrid(int grid)

Returns true if the HSeg is offgrid.

7.3.9 inst class

An instance is a reference to a cellView, in another cellView. Instances correspond to GDS2 SREFs or DEF components. Instances are created using the dbCreateInst cellView function.

int coord = i.left()

Get the left edge of the instance's bounding box.

int coord = i.bottom()

Get the bottom edge of the instance's bounding box.

int coord = i.right()

Get the right edge of the instance's bounding box.

int coord = i.top()

Get the top edge of the instance's bounding box.

bool i.offGrid(int grid)

Checks if an instance origin is on the grid grid, which is in database units.

i.status(db_PlaceStatus s)

Set the placement status of the instance. db_PlaceStatus can be one of: DB_UNPLACED, DB_PLACED, DB_FIXED, DB_COVER, DB_UNKNOWN.

db_PlaceStatus i.status()

Get the placement status of the instance.

char * i.getPlacementStatusStr()

Get the placement status of the instance as a string.

i.orient(orient_t orient)

Set the instance orientation. orient can be one of: R0, R90, R180, R270, MX, MXR90, MY, MYR90.

orient_t i.orient ()

Get the instance orientation.

i.source(db_SourceType s)

Set the instance source type. db_SourceType can be one of: DB_SRC_NONE, DB_SRC_NETLIST, DB_SRC_DIST, DB_SRC_USER, DB_SRC_TIMING.

db_SourceType i.source()

Get the instance source type.

char * i.getPlacementSourceStr()

Get the instance source type as a string.

i.bound(bool b)

Set the instance binding. This should probably not be set by the user.

bool i.bound()

Get the instance binding status. An instance is bound if it references a valid master.

double i.mag()

Get the instance's magnification. Magnifications other than 1.0 are supported, but their use is strongly discouraged.

i.mag(double m)

Sets the instance's magnification.

char *s = i.libName()

Get the instance's lib name.

[library](#) lib = i.lib()

Get the instance's [library](#) .

char *s = i.cellName()

Get the instances's cell name.

i.cellName(char *s)

Set the instance master's cellName.

char *s = i.viewName()

Get the instance's view name.

i.instName(cellView cv, char *instName)

Set the instance's *instName*. *cv* is the cellView containing the instance.

char * i.instName()

Get the instance's instName.

[cellView](#) cv = i.getMaster()

Get the cellView of the instance's master. If the instance is unbound, returns a null cellView .

i.setMaster(cellView cv)

Set the instance's master.

Point p = i.origin()

Get the origin of the instance. Note that an instance's origin does not have to be e.g. the lower left of its bounding box - it can be anywhere.

i.origin(Point p)

i.origin(int x, int y)

Set the origin of the instance.

Rect box = i.bBox()

Get the instance's bounding box.

Rect box = i.getBoundary()

Gets the instance's boundary rectangle. If the instance is e.g. a LEF macro then it will contain a shape on the TECH_PRBOUNDARY_LAYER, and the Rect representing this boundary shape will be returned. The shape is transformed according to the inst's origin, orientation and magnification.

dbtype_t_t i.objType()

Returns the objects type as INST

char * i.objName()

Returns the print name i.e. "INST"

int dist = i.getNearestEdge(Point p, segment &edge)

Get the nearest edge of this object.

i.transform(transform trans)

Transform the instance by the given transform.

i.scale(double scalefactor, double grid)

Scale the instance origin coordinates by *scalefactor*, snapping to *grid*.

i.Move(cellView dest, Point delta, bool opt = True)

Move the instance origin by *delta*. If *opt* is True then the database is re-optimised for the new inst position. If there are a lot of objects to move it makes sense to turn this off and instead use the cellView update() function after moving them all.

dbObj obj = i.Copy(cellView dest, Point delta)

Copy the instance. *dest* is the destination cellView , *delta* is the offset from the current origin.

dbObjList <dbObj > objs = i.Flatten(cellView dest, transform trans)

Flatten the instance into the cellView *dest*, with the given transform *trans* .

i.dbCreateInstPin(net n, char *name)

Create an instance pin on this instance for the net *n* and pin name *name* .

i.dbDeleteInstPin(instPin ip)

Delete the instPin *ip* from this instance.

instPin ip = i.dbFindInstPinByName(char *name)

Find the inst pin with name *name* on this instance. Returns null if not found.

list = i.getInstPins()

Get a list of all instPins for this instance.

int num = i.getNumInstPins()

Get the number of instPins for this instance.

7.3.10 instPin class

An instPin is usually created by its constructor. An instPin represents the hierarchical crossing of a net at one level of hierarchy to a pin on the instance of a cell (the lower level of hierarchy). Thus an instPin needs a valid net and instance whose master must have a pin of the given name.

instPin ip = instPin (inst i, net n, char *name)

Create an instPin for inst *i* and net *n* with name *name*.

ip.setInst(inst i)

Set the instPin's instance to *i*.

inst i = ip.getInst()

Get the instPin's inst .

ip.setName(char *name)

Set the instPin's *name*

char *name = ip.getName()

Get instPin's name

ip.setNet(net n)

Set the instPin's net

net n = ip.getGet()

Get the instPin's net

ip.setPin(pin p)

Set the master's pin

pin p = ip.getPin()

Get the master's pin

ip.setSpecial(bool s)

Set this instPin as special. Used for LEF/DEF.

bool special = ip.isSpecial()

Get the instPin 's special status. Used for LEF/DEF.

Point ip.getPortLoc()

Get the centre of the bounding box of the inst pin 's port(s)

bool ip.isSupplyPin()

Returns true if this instPin is a supply pin .

7.3.11 label class

The label class is derived from a shape. This class is normally created in a cellView using the dbCreateLabel function.

char *name = l.theLabel()

Gets the label name

l.theLabel(char *name)

Sets the label *name*.

Rect box = l.bBox

Gets the bounding box of the label . Note that as a label does not have a 'real' bounding box - the box is approximately the size of the displayed text of the label .

double h = l.height()

Gets the label 's height attribute

l.height(double h)

Sets the label 's height

double w = l.width()

Gets the label 's width

l.width(double w)

Sets the label 's width. This is not used.

orient_t o = l.orient()

Gets the label orientation.

l.orient(orient_t o)

Sets the label orientation.

Point p = l.origin()

Gets the label 's origin.

l.origin(int x, int y)

Sets the label 's origin.

l.origin(Point p)

Sets the label 's origin.

db_TextAlign a = l.align()

Gets the label 's alignment.

l.align(db_TextAlign a)

Sets the label 's alignment.

l.objType()

Get the object type (TEXT)

char *name = l.objName()

Gets the object name ("LABEL")

int dist = l.getNearestEdge(Point p, segment &edge)

Gets the nearest edge of the label 's bounding box to a Point *p*. The function returns the distance to the *edge*.

l.transform(transform trans)

Transform a label by some transform *trans*.

l.Move(cellView dest, Point delta, bool opt = True)

Moves a label by distance *delta*. If *opt* is True then the database is re-optimised for the new label position. If there are a lot of objects to move it makes sense to turn this off and instead use the cellView update() function after moving them all.

dbObj obj = l.Copy(cellView dest, Point delta, int layer=-1)

Copy the label to cellView *dest*, with offset *delta*. If *layer* is non-negative the label will be copied to the new layer.

dbObj obj = l.Flatten(cellView dest, transform trans)

Flatten the label into cellView *dest* with some transform *trans*.

l.bias(int bias, int xgrid, int ygrid,)

Bias the label . As the label is really just a point, this does nothing useful.

l.scale(double scale, double grid)

Scale the label . The label 's origin is scaled by the value *scale*.

7.3.12 library class

All design data is stored in libraries. Libraries contain cells and views; the combination of a cell and a view is a cellView , which contains the actual design data. For example a library may contain a cell 'NAND2'. This cell may contain a cellView 'NAND2' 'layout', where 'layout is the view of the cell.

A library has a techfile associated with it.

lib = library ("fred")

Construct a new library called "fred", returning the library object.

bool result = lib.dbOpenLib("libPath")

Opens and reads a previously saved library . libPath is the full path to the library , including the library name. Returns True if the library can be opened successfully, otherwise False.

bool result = lib.dbSaveLib("libPath")

Saves a library to disk. libPath is the full path to the library , including the library name. Returns True if the library can be saved successfully, otherwise False.

lib.dbClose("cellName", "viewName")

Closes a cellView . Currently this does not purge the cellView from virtual memory.

lib = getLibByName("fred")

Searches for a library called "fred" in the list of currently open libraries and returns the library if found, or None if not.

list = getLibList()

Returns a Python list of the currently open libraries.

list = lib.cellNames()

Returns a Python list of all the cell names in the library .

list = lib.getCells()

Returns a Python list of the cells in the library.

list = lib.viewNames()

Returns a Python list of all the view names in the library .

list = lib.getViews()

Returns a Python list of the views in the library.

int size = lib.dbu()

Return the size of a database unit in meters. This is deprecated; use the cellView userUnits() function to determine the user units, and the cellView dbuPerUU() function to return the number of database units per user unit.

int num = lib.dbuPerUU()

Return the number of database units per micron (defaults to 1000). This is deprecated; use the cellView dbuPerUU() function to return the number of database units per user unit.

lib.dbBindInstMasters()

Rebinds the instance masters for this library. All cellViews in the library are checked, and if their master cellView is unbound, then a search is performed in the currently open libraries in an attempt to rebind it. For example, to rebind all open libraries you can use the following:

```
libs = getLibList()
for lib in libs :
        lib.dbBindInstMasters()
# end for
```

cellView cv = lib.dbOpenCellView("cellName", "viewName", 'mode')

Returns a cellView object. "mode" can be 'r', 'w' or 'a'. 'w' mode is used to create a new cellView ; the cellView must not exist. 'a' mode is used to append (edit) an existing cellView ; the cellView must exist. 'r' mode is used to read an existing cellView ; the cellView must exist. An exception is thrown on failure.

bool result = lib.dbCopyCellView("newCellName", "newViewName", "oldLibName", "oldCellName", "oldViewName")

Copies a cellView into this library. Returns True if successful.

bool result = lib.dbRenameCellView("newCellName", "newViewName", "oldCellName", "oldViewName")

Renames a cellView in this library. Returns True if successful.

cellView cv = lib.dbFindCellViewByName("cellName", "viewName")

Find a cellView in this library. Returns a cellView object corresponding to the given cellName and viewName, or None if it does not exist in the library .

cell c = lib.dbFindCellByName("cellName")

Returns a cell object corresponding to the given cellName, or None if it does not exist.

view v = lib.dbFindViewByName("viewName")

Returns a view object corresponding to the given viewName, or None if it does not exist.

bool result = lib.dbDeleteCellView("cellName", "viewName")

Deletes the cellview specified by cellName and viewName and returns True if sucessful, False if not.

bool result = lib.dbRenameCell("newCellName", "oldCellName")

Renames a cell. Returns True if sucessful, False if not.

bool result = lib.dbDeleteCell("cellName")

Deletes the cell specified by cellName and returns True if sucessful, False if not.

char *name = lib.libName()

Returns the name of the library .

char *path = lib.libPath()

Returns the library path if the library has been read or saved on disk, otherwise None.

dbTechFile tech = lib.tech()

Returns the library 's techFile.

library lib = getLibByName("libName")

Get a library by name.

int index = lib.addVia(via v, bool check)

Adds a via v to the library via table and returns the via's index in that table. If *check* is true (the default is false), the via name is checked and the new via will NOT be added; the index returned is that of the existing via.

via v = lib.getVia(int index)

Gets a via by *index* from the library . No bounds checking is performed.

via v = lib.getViaByName(char *name)

Gets a via by *name*.

int index = lib.getViaIndexByName(char *name)

Gets a via's index by the via *name*.

char *name = lib.getViaNameByindex(int index)

Gets a via's name from its *index*.

int num = lib.getNumVias()

Gets the number of vias in the library 's via table. Note the table size is currently limited to 8192 vias.

7.3.13 line class

The line class is derived from a shape. A line can be considered a zero width path. This class is normally created in a cellView using the dbCreateLine() function.

l = line

Creates a new line object.

Rect b = l.bBox()

Get the bounding box of this line. This is the convex hull of the points in the line.

dbtype_t_t t = l.objType()

Returns the object type of this line as LINE.

char *name = l.objName()

Returns the object name of this line as "LINE".

int num = l.nPoints()

Returns the number of points of the line.

Point * pts = l.ptlist()

Returns the point list of this line as a C array of Points.

Point p = l.at(index)

Returns the Point p at the *index* into the list of points.

bool l.addPoint(Point *p)

Adds a Point to the end of this line.

bool l.deletePoint(int index)

Deletes the vertex at *index* of this line.

l.setPoint(int index, Point p)

Sets the vertex *index* of the line to Point *p*.

l.setPoint(int index, int x, int y)

Sets the vertex *index* of the line to the point given by *x, y*.

int dist = l.getNearestEdge(Point p, segment &edge)

Get the distance of the nearest segment *edge* of this line to the point *p*

int dist = l.getNearestVertex(Point p, vertex &vert)

Get the distance of the nearest vertex *vert* of this line to the point *p*

double area = l.area()

Get the area of this line.

int perim = l.perimeter()

Get the perimeter of this line.

int length = l.length()

Get the length of this line.

l.transform(transform trans)

Transform this line using trans.

bool b = l.ptInPoly(Point p, bool includeEnds = True)

Returns True if the point *p* is on the line. If *includeEnds* is true, this includes the line start and end point.

bool b = l.ptInRect(Rect r)

Returns true if the line crosses (intersects) a rect r.

l.Move(cellView dest, Point delta, bool opt = True)

Move this line by distance *delta*. If *opt* is True then the database is re-optimised for the new line position. If there are a lot of objects to move it makes sense to turn this off and instead use the cellView update() function after moving them all.

dbObj obj = l.Copy(cellView dest, Point delta, int layer = -1)

Copy this line to cellView *dest*, with offset *delta*. If *layer* is non negative the line will be copied to the new layer number.

dbObj obj = l.Flatten(cellView dest, transform trans)

Flatten this line into cellView *dest* with transformation *trans*.

l.Stretch(Point delta, segment seg)

Stretch a segment *seg* of this line by *delta*.

7.3.14 lpp class

A **lpp** object forms a layer-purpose pair. It manages objects in a tree structure for fast spatial searching.

lp = lpp(cellView cv)

Constructs a lpp object with master cellView *cv*.

lp.layerName(const char *name)

Sets the layer *name* of the lpp.

char * lp.layerName()

Gets the lpp's layer name.

lp.purpose(char *name)

Sets the purpose *name* of the lpp.

char * lp.purpose()

Gets the lpp's purpose name.

lp.layerNum(int layerNum)

Sets the layer number of the lpp.

int lp.layerNum()

Gets the lpp's layer number.

int lp.numShapes()

Gets the number of shapes in this lpp.

cellView cv = lp.cv()

Get the cellView for this lpp.

Rect box = lp.bBox()

Get the bounding box of all shapes in this lpp.

lp.bBox(Rect box)

Set the bounding box of the lpp.

lp.optimiseTree()

Optimise the lpp. Must be carried out after adding objects.

lp.updateTree(dbObj obj)

Update the lpp for an object.

dbObjList <dbObj > list = lp.dbGetOverlaps (Rect searchRect, int filterSize=0)

Search the lpp for shapes overlapping the search rectangle *searchRect*. If *filterSize* is non-zero, only shapes with a width and height greater than filterSize are reported.

lp.dbGetOverlaps (dbObjList <dbObj > list, Rect searchRect, int filterSize=0)

As above, but shapes are appended to the existing list.

list = lp.dbGetOverlaps(Rect searchRect, int filterSize=0)

As above, but objects are returned as a Python list.

7.3.14.1 Iterator

An iterator to allow traversing the objects in the lpp using Python.

iter = objIterator(lpp lp)

Initialises the dbObj iterator for the lpp. For example:

```
iter = objIterator(lpp)
while not iter.end() :
    obj = iter.value()
    type = obj.objType()
    print "object type = ", type
    iter.next()
```

dbObj obj = iter.value()

Returns the current object.

iter.next()

Advances the iterator to the next dbObj .

bool iter.end()

Returns false if there are more objects, else returns true if there are no more.

7.3.15 mpp class

The mpp class is derived from a shape. This class is normally created in a cellView using the dbCreateMPP() function.

mpp m = mpp;

Creates a mpp.

m.addLayer(mppLayer lyr)

Adds a layer to the mpp.

mppLayer lyr = m.getLayer(int idx)

Gets the mpp layer by index *idx*.

mppLayer *lyrs = m.getLayers()

Gets the mpp layer as an array for the mpp.

m.setLayers(mppLayer lyr, int numLayers)

Sets the mpp layers.

int num = m.numLayers()

Get the number of mpp layers.

m.setNumLayers(int num)

Set the number of **mpp** layers.

m.setMppRule(mppRule rule)

Set the mpp rule.

mppRule rule = m.getMppRule()

Get the mpp rule.

Rect b = m.bBox();

Get the bounding box of this mpp.

dbtype_t_t t = m.objType()

Returns the object type of this path as MPP.

char *name = m.objName()

Returns the object name of this path as "MPP".

int n = p.nPoints()

Returns the number of points of the path.

Point * pts = m.ptlist()

Returns the point list of this mpp as an array or Points.

m.bias(int bias, int xgrid, int ygrid,)

Bias this mpp by bias, snapping to the grid *xgrid* and *ygrid*.

m.scale(double scale, double grid)

Scale this mpp by *scale*, snapping to the grid *grid*.

int d = m.getNearestEdge(Point p, &segment edge)

Get the distance of the nearest segment *edge* of this mpp to the point *p*

int d = m.getNearestVertex(Point p, &vertex vert)

Get the distance of the nearest vertex *vert* of this mpp to the point *p*

m.transform(transform trans)

Transform this mpp using *trans*.

bool m.ptInPoly(Point p)

Returns true if the Point *p* is contained in the mpp or on its edges.

m.Move(cellView dest, Point delta, bool opt = True)

Move this mpp by distance *delta*. If *opt* is True then the database is re-optimised for the new mpp position. If there are a lot of objects to move it makes sense to turn this off and instead use the cellView update() function after moving them all.

dbObj obj = m.Copy(cellView dest, Point delta)

Copy this mpp to cellView *dest*, with offset *delta*.

dbObj obj = m.Flatten(cellView dest, transform trans)

Flatten this mpp into cellView *dest* with transformation *trans*.

m.Stretch(Point delta, segment seg)

Stretch segment *seg* of this mpp by *delta*.

m.compressPoints();

Removes colinear points.

polygon q = m.shapeToPoly ()

Converts this mpp to polygons.

7.3.16 net class

The net class is normally created in a cellView using the dbCreateNet() function. A net is derived from a dbObj . Nets have pins (which represent connections at this level of hierarchy with upper levels of hierarchy) and instPins (which represent connections with instances, i.e. lower levels of hierarchy). These provide a means for hierarchical connectivity from the pins on an instance of the cellView to the instPins on instances in the cellView .

n.name(char *name)

Sets the *name* of the net .

char *name = n.name()

Gets the net name.

dbtype_t_t t = n.objType()

Gets the net object type as NET

char *name = n.objName()

Gets the net object name as "NET".

instPin ip = n.dbCreateInstPin(char *instname, char *pinname)

Creates an instPin for this net with instname as the instance name and pinname as the name of the pin .

n.dbDeleteInstPin(instPin ip)

Deletes an instPin *ip* of this net .

n.dbDeleteinstPin(char *instName, char *pinName)

Deletes an instPin of this net, the *instName* and *pinName* are the names of the inst and pin.

int num = n.getNumInstPins()

Gets the number of inst pins for this net .

n.addInstPin(instPin ip)

Add an instPin for this net .

list = n.getInstPins()

Get the net 's instPin list.

double l = n.getHPWL(double &x, double &y)

Gets the half perimeter wirelength of this net .

n.setSpecial(bool val)

Sets the net as a specialnet.

bool n.isSpecial()

Gets the net 's specialnet status.

n.setPins(dbObjList <pin > pins)

Set the net 's pin list.

list = n.getPins()

Gets the net 's pin list.

n.addPin(pin p)

Add a pin to this net .

n.setShapes(dbObjList <shape> shapes)

Sets the net 's shape list.

list = n.getShapes()

Gets the net 's shape list.

int n.getNumShapes()

Get the number of shapes associated with this net .

n.addShape(dbObj shp)

Add a shape to the net 's shape list.

n.deleteShape(dbObj shp)

Delete a shape from the net 's shape list.

7.3.17 path class

The path class is derived from a shape. A path is represented by a list of vertices, plus a width, style, beginExtent and endExtent. This class is normally created in a cellView using the dbCreatePath() function.

p.width(int w)

Sets the path width to w.

int w = p.width()

Gets the path width.

p.style(int s)

Sets the path style, i.e. the type of the path end. The style can be one of: 0 - truncate, 1 - round, 2 - extend, 4 - varextend, 8 - octagonal. Python global variables TRUNCATE, ROUND, EXTEND, VAREXTEND, OCTAGONAL are defined to these values.

int s = p.style()

Gets the path style.

p.beginExt(int e)

Set the path begin extent. For a path style 2 (extend) or 4 (varextend) , this is the begin extent of the path.

int e = p.beginExt()

Get the path begin extent. For a path style of 2 (extend) this is half the path's width. For a path style 4 (varextend), this is the begin extent of the path.

p.endExt(int e)

Set the path end extent. For a path style 2 (extend) or 4 (varextend), this is the end extent of the path.

int e = p.endExt()

Get the path end extent. For a path style of 2 (extend) this is half the path's width. For a path style 4 (varextend), this is the end extent of the path.

Rect **b = p.bBox()**

Get the bounding box of this path.

p.bBox(Rect b)

Set the bounding box of this path. Not useful and will throw an exception if called.

dbtype_t_t t = p.objType()

Returns the object type of this path as PATH.

char *name = p.objName()

Returns the object name of this path as "PATH".

int n = p.nPoints()

Returns the number of points of the path.

Point * pts = p.ptlist()

Returns the point list of this path as a C array of Points.

Point p = p[index]

Returns the Point p at the index into the list of points.

p.bias(int bias, int xgrid, int ygrid,)

Bias this path by bias, snapping to the grid *xgrid* and *ygrid*.

p.scale(double scale, double grid)

Scale this path by scale, snapping to the grid grid.

int d = p.getNearestEdge(Point p, segment &edge)

Get the distance of the nearest segment *edge* of this path to the point *p*

int d = p.getNearestVertex(Point p, vertex &vert)

Get the distance of the nearest vertex *vert* of this path to the point *p*

double a = p.area()

Get the area of this path.

int p = p.perimeter()

Get the perimeter of this path.

p.transform(transform trans)

Transform this path using *trans*.

bool p.ptInPoly(Point p)

Returns true if the Point *p* is contained in the path or on its edges.

p.Move(cellView dest, Point delta, bool opt = True)

Move this path by distance *delta*. If *opt* is True then the database is re-optimised for the new path position. If there are a lot of objects to move it makes sense to turn this off and instead use the cellView update() function after moving them all.

dbObj obj = p.Copy(cellView dest, Point delta, int layer = -1)

Copy this path to cellView *dest*, with offset *delta*. If *layer* is a positive integer the path will be copied to the new layer number.

dbObj obj = p.Flatten(cellView dest, transform trans)

Flatten this path into cellView *dest* with transformation *trans*.

p.Stretch(Point delta, segment seg)

Stretch segment *seg* of this path by *delta*.

p.compressPoints()

Removes colinear points from the path.

polygon poly = p.shapeToPoly ()

Converts this path to a polygon.

7.3.18 pin class

The pin class is normally created in a cellView using the dbCreatePin() function. A pin is derived from a dbObj . You can create a logical pin using cellView ::dbCreatePin(); to create a physical pin you need to first create a logical pin , then use cellView ::dbCreatePort() to assign a physical shape to the pin .

p.name(char *name)

Sets the pin 's *name.*

char *name = p.name()

Gets the pin name.

p.setDir(db_PinDirection dir)

Sets the pin direction. db_PinDirection can be one of DB_PIN_INPUT, DB_PIN_OUTPUT, DB_PIN_INOUT, DB_PIN_FEEDTHRU, DB_PIN_TRISTATE.

db_PinDirection dir = p.dir()

Gets the pin direction.

p.setShape(db_PinShape s)

Sets the pin shape. db_PinShape can be one of DB_PIN_ABUTMENT, DB_PIN_RING, DB_PIN_FEED.

db_PinShape s = p.getShape()

Gets the pin shape.

p.setUse(db_PinUse use)

Sets the pin use. db_PinUse can be one of DB_PIN_SIGNAL, DB_PIN_ANALOG, DB_PIN_CLOCK, DB_PIN_GROUND, DB_PIN_POWER, DB_PIN_RESET, DB_PIN_SCAN, DB_PIN_TIEOFF.

db_PinUse use = p.use()

Gets the pin use.

p.setNet(net n)

Sets the pin 's net .

net n = p.getNet()

Gets the pin 's net .

char *name = p.getNetName()

Gets the pin 's net name as a string.

dbtype_t_t t = p.objType()

Gets the pin object type as PIN.

p.setPorts(dbObjList <shape> *ports)

Sets the pin 's port (physical shape) list.

list = p.getPorts()

Gets the pin 's port (physical shape) list.

int p.getNumPorts()

Gets the number of port shapes for the pin .

p.addPort(shape *shp)

Adds a port shape to the pin .

7.3.19 Point class

A Point class represents a coordinate or xy pair.

p = Point

Creates a Point object p. The Point is initialised to (0, 0) by default.

p = Point (int x, int y)

Creates a Point object and initialises its coordinates.

int x = p.getX()

int y = p.getY()

Get the specified Point coordinate.

p.setX(int x)

p.setY(int y)

p.set(int x, int y)

Set the specified Point coordinate.

operator ==

Returns true if the two Points are equal.

operator !=

Returns true if the two Points are not equal.

operator <

Returns true if the first point is 'less than' the second. First the X coordinate is compared; if equal then the Y coordinate is compared.

operator >

Returns true if the first point is 'greater than' the second. First the X coordinate is compared; if equal then the Y coordinate is compared.

operator +

A Point plus a Vector returns a Point offset by the Vector.

A Point plus a Point is a Point , offset by the Point .

operator –

A Point minus a Vector returns a Point .

A Point minus a Point returns a Vector.

operator -

A Point minus a Vector returns a Point transposed by the Vector.

operator +=

A Point plus a scalar (i.e. an integer) is offset, or moved, by the value of the scalar in both X and Y.

A Point plus a Point returns a Point with the sum of the two Points X and Y values.

operator -=

A Point minus a Point returns a Point with the differrence of the two Points X and Y values.

operator *=

A Point times a scalar is scaled (multiplied) by the scalar.

7.3.20 pointList class

A pointList class represents a list (actually an array) of points.

pointList pl = pointList

Creates a pointList.

pointList pl = **pointList** (**Point** *pts, int num, bool compress = True)

Creates a pointList from the points specified by the array *pts* with size *num*. If *compress* is true, the points will be sorted counterclockwise and colinear points removed.

pointList pl = Point (int *xpts, int *ypts, int num, bool compress = True)

Creates a pointList from the points specified by the arrays *xpts* and *ypts* with size *num*. If *compress* is true, the points will be sorted counterclockwise and colinear points removed.

operator ==

Returns true if the two pointLists are equal.

operator !=

Returns true if the two pointLists are not equal.

operator <

Returns true if one pointList is less than another. 'Less' is the case is any vertex X or Y coordinate is less than the other corresponding vertex.

pl.setPtlist(Point *pts, int num, bool compress = True)

Sets a pointList from the points specified by the array *pts* with size *num*. If *compress* is true, the points will be sorted counterclockwise and colinear points removed.

pl.setPtlist(int *xpts, int *ypts, int num, bool compress = True)

Sets a pointList from the points specified by the arrays *xpts* and *ypts* with size *num*. If compress is true, the points will be sorted counterclockwise and colinear points removed.

pl.setPtlist(Rect box)

Sets a pointList with the 4 vertices of a rectangle (LL, LR, UR, UL).

Point *pts = pl.points()

Get the raw pointList as an array of Points.

pl.append(Point p)

Append the pointList with Point p.

pl.append(pointList pl)

Append the pointList with pointList pl.

Point p = pl.at(int idx)

Get the Point p given by the index idx.

int num = pl.numPts()

Get the number of points in the pointList.

Rect b = pl.bBox()

Gets the bounding box of the pointList.

double a = pl.area()

Gets the area of the pointList. This assumes the pointList is closed, i.e. there is an edge between the last and first vertex.

int p = pl.perimeter()

Gets the perimeter of the pointList. This assumes the pointList is closed, i.e. there is an edge between the last and first vertex.

pl.transform(transform trans)

Transform all points in the pointList by *trans*.

pl.scale(double factor, int grid)

Scales all points in a pointList by factor, snapping them to a grid grid (in database units)

pl.compressPoints(bool ortho, bool xfirst)

Compresses all points in a pointList by removing all colionear points and ordering them counterclockwise. If *ortho* is true, points are assumed to be manhattan and are stored in a more compressed format.

bool b = pl.isSelfIntersecting(bool isClosed = true)

Returns true if the pointList is self intersecting.

bool pl.overlaps(pointList other, touching = false)

Returns true if one pointList overlaps another. If *touching* is true, returns true if the pointLists touch.

bool pl.contains(Point p, bool touching = true)

Returns true if the pointList contains Point p. If *touching* is true, returns true if Point p touches an edge of the pointList.

bool pl.contains(Rect r, bool touching = true)

Returns true if the pointList contains Rect r. If *touching* is true, returns true if a vertex of Rect r touches an edge of the pointList.

Point p = pl.intersectsAt(Edge e)

Gets the first intersection of the Edge with the pointList.

bool pl.isOrthogonal(bool isClosed = true)

Returns true if the pointList is orthogonal i.e. manhattan.

7.3.21 polygon class

The polygon class is derived from a shape. This class is normally created in a cellView using the dbCreatePoly() function. Note that dbCreatePoly() will create a square or a rectangle if the polygon has 4 points. A polygon is represented by a series of points, which represent the vertices of the polygon. There is an implicit edge between the first and last point.

Rect b = p.bBox()

Get the bounding box of this polygon.

dbtype_t_t t = p.objType()

Returns the object type of this polygon as POLYGON.

char *name = p.objName()

Returns the object name of this polygon as "POLYGON".

int n = p.nPoints()

Returns the number of points of the polygon's boundary. Note that polygons are not closed as they are in GDS2.

Point * pts = p.ptlist()

Returns the point list of this polygon as a C array of Points.

Point p = p[index]

Returns the Point p at the index into the list of points.

p.bias(int bias, int xgrid, int ygrid,)

Bias this polygon by bias, snapping to the grid *xgrid* and *ygrid*.

p.scale(double scale, double grid)

Scale this polygon by scale, snapping to the grid grid.

int d = p.getNearestEdge(Point p, segment &edge)

Get the distance of the nearest segment *edge* of this polygon to the point *p*

int d = p.getNearestVertex(Point p, vertex &vert)

Get the distance of the nearest vertex *vert* of this polygon to the point *p*

bool ptInPoly(Point p)

Returns true if the point in inside or on the edge of the polygon.

double a = p.area()

Get the area of this polygon.

int p = p.perimeter()

Get the perimeter of this polygon.

p.transform(transform trans)

Transform this polygon using *trans*.

p.Move(cellView dest, Point delta, bool opt = True)

Move this polygon by distance *delta*. If *opt* is True then the database is re-optimised for the new polygon position. If there are a lot of objects to move it makes sense to turn this off and instead use the cellView update() function after moving them all.

dbObj obj = p.Copy(cellView dest, Point delta, int layer = -1)

Copy this polygon to cellView *dest*, with offset *delta*. If *layer* is non-negative the polygon will be copied to the new layer number.

dbObj obj = p.Flatten(cellView dest, transform trans)

Flatten this polygon into cellView *dest* with transformation *trans*.

p.Stretch(Point delta, segment seg)

Stretch segment *seg* of this polygon by *delta*.

p.Stretch(Point delta, vertex v)

Stretch vertex *v* of this polygon by *delta*.

p.compressPoints()

Removes colinear points, sets the point order to be counterclockwise and sets the first point to be the smallest in X and Y.

bool p.selfIntersecting()

Returns true if the polygon is self-intersecting.

7.3.22 property class

The property class is used to represent a property list. It is not recommended to use it directly, use the property functions for the dbObj and its derived classes.

7.3.23 Rect class

A Rect class is used to represent a rectangle comprising two coordinate pairs. Note that this is NOT the same as a rectangle object which is a database object instead.

r = Rect

Creates a Rect object r. The rectangle coordinates are set to invalid i.e. llx = +infinity, urx = -infinity etc.

r = Rect (Point ll, Point ur)

Creates a Rect object r and initialises it with Point types ll, ur.

r = Rect (int llx, int lly, int urx, int ury)

Creates a Rect object and initialises its coordinates.

int x = r.left()

int y = r.bottom()

int x = r.right()

int y = r.top()

Get the specified Rect coordinate.

r.setLeft(int x)

r.setBottom(int y)

r.setRight(int x)

r.setTop(int y)

Set the specified Rect coordinate.

Point ll = r.getLL()

Point ur = r.getUR()

Get the lower left or upper right Rect coordinates as Points.

r.invalidate()

Set the Rect to invalid, i.e. llx = +infinity, urx = -infinity etc.

r.scale(double s)

r.scale(int s)

Scale a Rect coordinates by dividing them by s.

Rect s = r.offset(int x, int y)

Offset (transpose) a Rect by the specified x and y coordinates. The Rect r is modified.

r.width(int w)

Set a Rect 's width. The lower left remains the same.

int w = r.width()

Get the width of a Rect .

r.height(int h)

Set the height of a Rect . The lower left remains the same.

int h = r.height()

Get the height of a Rect .

Point p = r.centre()

Get the centre point of a Rect .

bool b = r.isSquare()

Returns True is the rectangle is square, False if it is not.

transform(orient_t orient, Point p)

Transforms a Rect using Point *p* and orientation *orient*.

r.swapxy()

Swaps the X and Y coordinates of a Rect .

r.unionWith(Rect p)

Rect r is set to the union of the Rects *r* and *p*, i.e. the bounding box of both.

r.unionWith(Point p)

Rect r is set to the union of itself and Point *p*, i.e. the bounding box of both.

bool b = r.touchOrOverlaps(int x, int y)

Returns True if the Rect touches or overlaps the point *x, y*; returns False otherwise.

bool b = r.touchOrOverlaps(int xlo, int ylo, int xhi, int yhi)

Returns True if the Rect touches or overlaps the rectangle formed by *xlo, ylo, xhi, yhi*; returns False otherwise.

bool b = r.touchOrOverlaps(Rect p)

Returns True if the Rect touches or overlaps the Rect *p*; returns False otherwise.

bool b = r.touch(int x, int y)

Returns True if the Rect touches the point *x, y*; returns False otherwise.

bool b = r.touch(int xlo, int ylo, int xhi, int yhi)

Returns True if the Rect touches the rectangle formed by *xlo, ylo, xhi, yhi*; returns False otherwise.

bool b = r.touch(Rect p)

Returns True if the Rect touches the Rect *p*; returns False otherwise.

bool b = r.overlaps(int x, int y)

Returns True if the Rect overlaps the point *x, y*; returns False otherwise.

bool b = r.overlaps(int xlo, int ylo, int xhi, int yhi)

Returns True if the Rect overlaps the rectangle formed by *xlo, ylo, xhi, yhi*; returns False otherwise.

bool b = r.overlaps(Rect p)

Returns True if the Rect overlaps the Rect *p*; returns False otherwise.

bool b = r.contains(int x, int y)

Returns True if the Rect contains the point *x, y*; returns False otherwise.

bool b = r.contains(int xlo, int ylo, int xhi, int yhi)

Returns True if the Rect contains the rectangle formed by *xlo, ylo, xhi, yhi*; returns False otherwise.

bool b = r.contains(Rect p)

Returns True if the Rect contains the Rect *p*; returns False otherwise.

r.intersectsWith(Rect p)

Modifies Rect r to the intersection of itself and Rect *p*.

Rect **s = r.intersectsWith(Rect p)**

Returns a rectangle which is the intersection of *r* and *p*.

7.3.24 rectangle class

The rectangle class is derived from a shape. This class is normally created in a cellView using the dbCreateRect() function.

int left = r.left()

int bottom = r.bottom()

int right = r.right()

int top = r.top()

Get the coordinates of the rectangle.

r.setLeft(int x)

r.setBottom(int y)

r.setRight(int x)

r.setTop(int y)

Set the coordinates of the rectangle.

Point p = r.origin()

Get the origin (lower left) of this rectangle.

int w = r.width()

Get the width of this rectangle.

r.width(int w)

Set the width of this rectangle. The origin is maintained.

int h = r.height()

Get the height of this rectangle.

r.height(int h)

Set the height of this rectangle. The origin is maintained.

Point p = r.centre()

Get the centre of a rectangle.

Rect b = r.bBox()

Get the bounding box of this rectangle.

r.bBox(Rect b)

Set the bounding box of this rectangle. This will change the size of the rectangle.

dbtype_t_t t = r.objType()

Returns the object type of this rectangle as RECTANGLE.

char *name = r.objName()

Returns the object name of this rectangle as "RECTANGLE".

int n = r.nPoints()

Returns the number of points of the rectangle's boundary as 4.

Point * pts = r.ptlist()

Returns the point list of this rectangle as a C array of 4 points.

polygon p = r.shapeToPoly()

Returns a polygon with a pointlist identical to this rectangle.

r.bias(int bias, int xgrid, int ygrid,)

Bias this rectangle by bias, snapping to the grid *xgrid* and *ygrid*.

r.scale(double scale, double grid)

Scale this rectangle by scale, snapping to the grid grid.

int d = r.getNearestEdge(Point p, segment &edge)

Get the distance of the nearest segment *edge* of this rectangle to the point *p*;

int d = r.getNearestVertex(Point p, vertex &vert)

Get the distance of the nearest vertex *vert* of this rectangle to the point *p*;

double a = r.area()

Get the area of this rectangle.

int p = r.perimeter()

Get the perimeter of this rectangle.

r.transform(transform trans)

Transform this rectangle using *trans*.

bool ptInPoly(Point p)

Returns True if the point is contained in or on the edge of the rectangle.

r.Move(cellView dest, Point delta, bool opt = True)

Move this rectangle by distance *delta*. If *opt* is True then the database is re-optimised for the new rectangle position. If there are a lot of objects to move it makes sense to turn this off and instead use the cellView update() function after moving them all.

dbObj obj = r.Copy(cellView dest, Point delta, int layer = -1)

Copy this rectangle to cellView *dest*, with offset *delta*. If *layer* is non negative the rectangle will be copied to the new layer number.

dbObj obj = r.Flatten(cellView dest, transform trans)

Flatten this rectangle into cellView *dest* with transformation *trans*.

r.Stretch(Point delta, segment seg)

Stretch segment *seg* of this rectangle by *delta*.

r.Stretch(Point delta, vertex v)

Stretch vertex *v* of this rectangle by *delta*.

7.3.25 segment class

A segment is an edge of a dbObj with two points. It is derived from a dbObj so it can be selectable; it also references its parent dbObj . Segments are used when selecting an edge of e.g. a rectangle or polygon.

seg = segment(Point p0, Point p1)

Creates a segment with coordinates *p0* and *p1*.

seg = segment(int x1, int y1, int x2, int y2)

Creates a segment with the specified xy coordinates.

double seg.DistanceToPoint(Point *p*)

Get the distance from a point p to this segment.

Point seg.NearestPoint(Point *p*)

Get the nearest point on a segment to another point.

dbtype_t_t seg.objType()

Returns the objects type - SEGMENT.

seg.SetObj(dbObj obj)

Sets the dbObj associated with this segment.

dbObj seg.GetObj()

Gets the dbObj associated with this segment.

bool seg.isXSeg()

Returns True if this segment is horizontal, else False.

bool seg.isYSeg()

Returns True if this segment is vertical, else False.

bool seg.isManhattan()

Returns True if this segment is manhattan, else False.

Rect seg.bBox()

Returns a fake bounding box 10 dbu larger than the segment itself.

bool seg.segInRect(Rect r)

Returns True if the segment is contained in Rect r.

seg.transform(transform trans)

Transforms this segment according to *trans*.

seg.Move(cellView dest, Point delta, bool opt = True)

Moves this segment by *delta*. If *opt* is True then the database is re-optimised for the new segment position. If there are a lot of objects to move it makes sense to turn this off and instead use the cellView update() function after moving them all.

Point **seg.p0**

The first point of the segment.

Point **seg.p1**

The last point of the segment.

7.3.26 shape class

The shape class is derived from a dbObj . Shapes have layer and net information; a shape is not normally used directly but one of its derived classes is instead.

db.layer(int l)

Set the layer of this shape. Provided for backward compatibility only.

int l = db.layer()

Get the layer of this shape. Provided for backward compatibility only.

db.setNet(net n)

Set the net associated with this shape.

net n = db.getNet()

Get the net associated with this shape.

char name = db.getNetName()

Get the net name of the net associated with this shape.

7.3.27 techFile class

The techFile class contains technology related parameters, in particular the layers used in a design. A techFile object does not normally need to be created as creating a library will initialise a techFile associated with that library . For example:

```
lib = library ("myLib")
tech = lib.tech()
```

int tech.techLoad(char *fileName)

Loads a techFile specified by *fileName*.

int tech.techSave(char *fileName)

Saves a techFile to *fileName*.

7.3.27.1 Layer related operations

Layers are signed 16 bit integers and map to layer-purpose pairs. A layer number of -1 signifies an invalid layer.

bool tech.isSelectable(int layer)

Returns 1 if *layer* is selectable, else returns 0.

tech.selectable(int layer, bool sel)

Sets a *layer* selectable is *sel* is nonzero.

bool tech.isVisible(int layer)

Returns 1 if *layer* is visible, else returns 0.

tech.visible(int layer, bool vis)

Sets a *layer* visible is *vis* is nonzero.

bool tech.isUsed(int layer)

Returns 1 if *layer* is used, else returns 0.

tech.setUsed(int layer, bool used)

Sets a *layer* used if *used* is nonzero.

int tech.color(int layer)

Returns a 32 bit int of the *layer* color in rgba format.

tech.color(int layer, int color)

Sets a *layer* color. *color* is a 32 bit integer in rgba format.

tech.setLayerName(int layer, char *name)

Sets a layer's *name*.

char * tech.getLayerName(int layer)

Gets the layer's name.

tech.setLayerPurpose(int layer, char *name)

Sets a layer's purpose.

char * tech.getLayerPurpose(int layer)

Gets the layer's purpose.

char * tech.getLayerPurposePair(int layer)

Gets the layer's layer-purpose pair name e.g. "metal1 drawing".

tech.setLayerGdsLayer(int layer, int gdsNum)

Sets a layer's GDS number to *gdsNum*.

int tech.getLayerGdsLayer(int layer)

Gets a layer's GDS number.

tech.setLayerDataType(int layer, int *gdsNum*)

Sets a layer's GDS datatype to gdsNum.

int tech.getLayerDataType(int layer)

Gets a layer's GDS datatype.

int tech.getLayerNum(char *name, char *purpose, int warn=1)

Gets a layer number if one exists with the specified *name* and *purpose*. If it does not exist a warning is given unless *warn* is set to 0.

int tech.createLayer(char *name, char *purpose)

Creates a layer in the techFile with the specified *name* and *purpose*. Returns the layer number or -1 if the layer cannot be created.

tech.setLayerType(int layer, int type)

Sets the *layer* type. *type* can be one of T_CUT, T_ROUTING, T_BLOCKAGE, T_PIN, T_OVERLAP, T_MASTERSLICE or T_NONE.

int tech.getLayerType(int layer)

Gets the *layer* type.

tech.setLayerWidth(int layer, int width)

Sets the *layer* minWidth.

int tech.getLayerWidth(int layer)

Gets the *layer* minWidth.

tech.setLayerSpacing(int layer, int spacing)

Sets the *layer* minSpace.

int tech.getLayerSpacing(int layer)

Gets the *layer* minSpacing.

tech.setLayerPitch(int layer, int pitch)

Sets the *layer* pitch.

int tech.getLayerPitch(int layer)

Gets the *layer* pitch.

tech.setLayerDir(int layer, int dir)

Sets the *layer* routing direction. The routing direction can be LAYER_HORIZONTAL or LAYER_VERTICAL.

int tech.getLayerDir(int layer)

Gets the *layer* routing direction.

char *tech.getLayerDirAsStr(int layer)

Gets the *layer* routing direction as a string.

tech.setLayerResistance(int layer, double r)

Sets the *layer* resistance.

double tech.getLayerResistance(int layer)

Gets the *layer* resistance.

tech.setLayerAreaCap(int layer, double c)

Sets the *layer* area capacitance.

double tech.getLayerAreaCap(int layer)

Gets the *layer* area capacitance.

tech.setLayerEdgeCap(int layer, double c)

Sets the *layer* edge capacitance.

double tech.getLayerEdgeCap(int layer)

Gets the *layer* edge capacitance.

tech.setLayerOrder(int layer, int order)

Sets the *layer* order.

int tech.getLayerOrder(int layer)

Gets the *layer* order.

tech.setLineStyle(int layer, int style)

Sets the *layer* linestyle

int tech.getLineStyle(int layer)

Gets the *layer* linestyle.

tech.setLineWidth(int layer, int width)

Sets the *layer* linewidth

int tech.getLineWidth(int layer)

Gets the *layer* linewidth.

tech.setFillPattern(int layer, bits[128], int exists=0, char *name = NULL, int full32bit=0)

Sets the *layer* fill pattern. The bit array is 128 bytes when using OpenGL, 32 bytes when not using OpenGL.

bits[128] tech.getFillPattern(int layer)

Gets the *layer* fill pattern as an array of bytes; 128 bytes when using OpenGL else 32 bytes.

char * tech.getFillName(int layer)

Gets a fill pattern name for the *layer*.

tech.setFillType(int layer)

Gets the *layer* fill type. Can be one of F_HOLLOW, F_SOLID, F_CROSSED, F_STIPPLE.

int tech.getFillType(int layer)

Gets the *layer* fill type. Can be one of F_HOLLOW, F_SOLID, F_CROSSED, F_STIPPLE.

int tech.getCurrentLayer()

Gets the current layer as set by the LSW.

7.3.28 transform class

The transform class contains functions to transform coordinates in subcells placed with offset, rotation and magnification.

A a point with coordinates x, y can be transformed by a transformation matrix T by:

[x', y', 1] = [x, y, 1]T

The transformation matrix for an offset (a,b) with no rotation or magnification can be described as

```
T  = [ 1 0 0 ]
     [ 0 1 0 ]
     [ a b 1 ]
```

Rotations are e.g.

```
T90 = [  0 1 0 ]
      [ -1 0 0 ]
      [  0 0 1 ]
```

trans = transform (orient_t orient, Point p, double scale)

Construct a transform with orientation *orient*, origin *p* and magnification *scale*. The orientation can be specified by the constants R0, R90, R180, R270, MX, MXR90, My, MYR90.

trans = transform (orient_t orient, Point p)

trans = transform (orient_t orient, int x, int y)

Construct a transform with orientation *orient*, origin *p* or *x/y*.

trans = transform (orient_t orient)

Construct a transform with orientation *orient*.

trans = transform ()

Construct a transform with orientation R0.

trans.invert()

Invert a transformation matrix

trans.inverseTransformRect(Rect box)

Transform a Rect by the inverse of the transformation matrix. Useful if you want to take a Rect and transform it into the coordinate space of an instance with a transform. For example if you want to find if any shapes in an instance overlap a search box for a top level cell, use the inverse transform of the search box on all the instance's shapes. This means doing just one transform of the search box rather than one transform for each shape in the instance.

trans.inverseTransformPoint(Point p)

trans.inverseTransformPoint(int x, int y)

Transform a Point by the inverse of the transformation matrix

trans.transformRect(Rect box)

Transform a Rect by the transformation matrix

trans.transformRect(Rect box, Point origin)

Transform a Rect by the transformation matrix about a point *origin*.

trans.transformPoint(Point p)

Transform a Point by the transformation matrix

trans.transformPoint(Point p, Point origin)

Transform a point by the transformation matrix about a point *origin*.

trans.transformPointList(Point *ptlist, int size)

Transform an array of points of size *size* by the transformation matrix.

trans.transformPointList(Point *ptlist, int size, Point origin)

Transform an array of points of size *size* by the transformation matrix about a point *origin*.

trans.setOrient(orient_t orient)

Set the transformation matrix orientation

orient_t orient = trans.getOrient()

Get the transformation matrix orientation.

trans.setOrigin(Point p)

trans.setOrigin(int x, int y)

Set the transformation origin.

Point origin = trans.getOrigin()

Get the transformation origin.

trans.setMag(double scale)

Set the transformation matrix scale.

bool swapped = trans.isXYSwapped()

Returns True if the transformation is such that the objects XY coordinates would be swapped, e,g, if the object is R90.

7.3.29 ui class

All of the following functions are part of the ui class. There is a global pointer to the gui called cvar.uiptr . Therefore to use them, define you own variable e.g. gui=cvar.uiptr, then call as gui.OpenCellView(...)

int errors = ui.schCheck (char *libName, char *cellName, char *viewName, bool snapLabels, float snapDist)

Check a schematic cellView . *errors* is the number of errors found, or -1 if the cell could not be checked. If *snapLabels* is True, labels closer than *snapDist* to a wire will be snapped onto the wire.

int errors = ui.symCheck (char *libName, char *cellName, char *viewName)

Check a symbol cellView . *errors* is the number of errors found, or -1 if the cell could not be checked.

bool ok = ui.createCellView (char *libName, char *cellName, char *viewName)

Create a symbol cellView from the given cellView (normally a schematic). Returns True if successful.

bool ok = ui.loadPCell(char *libName, char *pcellName)

Loads the PCell with name *pcellName* into the library *libName*. If the PCell already exists in the library , the action is ignored and returns true. If a cellView with the same name exists, it is deleted and is replaced by the PCell supermaster. Note that once a PCell is loaded into a library and that library is saved, it will remain a PCell, so there is no need to load it again (although it is harmless and you will just get warnings about the load being ignored). If the PCell cannot be created, it returns false.

ui.addMarker(int x, int y, int size=20, int lineWidth=0, color=Qt::yellow)

Adds a marker at the specified *x* and *y* values (given in database units). The *size* of the marker defaults to 20 dbu and the *linewith* to 0 (i.e. one pixel wide).

ui.clearMarkers()

Clears all markers.

ui.getSelectedSet()

Returns a dbObjList of the selected set. There is also a top level python binding of the same name that returns a python list of selected objects.

ui.selectObj(dbObj *obj)

Selects an object. The existing selection list is cleared.

ui.deselectObj(dbObj *obj)

Deselects an object.

ui.addSelected(dbObj *obj)

Adds the object to the selected set.

ui.selectAll ()

Select all objects in the current canvas cellView.

ui.deselectAll ()

Deselect all objects in the current canvas cellView.

ui.selectArea (int x1, int y1, int x2, int y2, bool add = 0)

Select objects in the area given by *x1 y1 x2 y2*. If *add* is true, then the objects are added to the selected set.

ui.deselectArea (int x1, int y1, int x2, int y2)

Deselect objects in the area given *by x1 y1 x2 y2*.

ui.selectPoint (int x1, int y1, bool add = 0)

Select an object at the coordinate *x1 y1*. If *add* is true, then the object is added to the selected set.

ui.deselectPoint (int x1, int y1)

Deselect an object at the coordinate *x1 y1*.

ui.moveSelected(Point delta, orient_t orient)

Moves the selected set by *delta*, optionally rotating it by *orient*.

ui.copySelected(Point delta, orient_t orient)

Copies the selected set, moving the copy by *delta*, optionally rotating it by *orient*.

ui.getEditCellView()

Returns the current cellview being edited. If multiple cellviews are open, it returns the cellview of the current active window. There is also a top level python binding to this function, getEditCellView().

ui.getLibByName(char *name)

Returns the library given by *name*.

ui.getLibList()

Returns a dbObjList of all open libraries. There is also a top level python binding of the same name that returns a python list of open libraries.

ui.getCellList()

Returns a dbObjList of all open cellViews. There is also a top level python binding of the same name that returns a python list of open cellViews.

ui.openCellView (char * libName, char * cellName, char * viewName)

Opens the cellview specified by *libName*, *cellName* and *viewName* in a new window.

ui.closeLib(char *libName)

Closes the library specified by libName. If cellViews from that library are displayed, their windows will be closed. The library is removed from the list of open libraries. No checking is performed for edited cells.

bool ok = ui.importTech(char *libName, char *techFileName, unsigned int dbu=1000)

Imports the techFile *techFileName* into the library *libName*. The library is created if it does not already exist. Returns true if no error occurred.

bool ok = ui.exportTech(char *libName, char *techFileName, bool systemLayers)

Exports the techFile *techFileName* from the library *libName*. The library must exist. If *systemLayers* is 1, Glade system layers e.g. cursor, backgnd etc will be written to the techFile. This is only necessary if you have modified the system layers in the LSW - for example changed the backgnd color from black to white. Returns true if no error occurred.

bool ok = ui.importGds2 (char * libName, char * gdsFileName, char * dumpFile = NULL, int csen = 0, bool do_dump = False, double gdsScaleFactor = 1.0, double gdsXOffset = 0.0, double gdsYOffset = 0.0, int gdsNetAttr = 0, int gdsInstAttr = 0, bool compressed=False, bool dubiousData=True, bool setDBUfromGDS=True, bool reportCells=False, int pathConv=2, int convLayers=0, int layer=0, int datatype=0, bool openTopCell=False, bool setLibName=False, bool convertVias=False, int duplicates=0, char *viewName="layout")

Import the GDS2 file *gdsFileName* into library *libName*. The library is created if it does not already exist. If do_*dump* is 1 and *dumpFile* is a valid file name, the GDS2 will be written in an ascii format suitable for debugging purposes. *gdsScaleFactor* can be used to scale all coordinates in the GDS2 file. *gdsXOffset* and *gdsYOffset* can be used to apply a fixed offset to all GDS2 coordinates. *gdsNetAttr* specifies the GDS2 attribute number used for net names, if present, and gdsInstAttr specifies the GDS2 attribute number for instance names, if present. If *compressed* is true, a gzip compressed format file is expected and will be uncompressed during stream in. If *dubiousData* is true, dubious data constructs in the GDS2 file are reported. If *setDBUfromGDS* is true, the library DBUperUU is set from the GDS DBU. If *reportCells* is true, cells are reported in the message window as they are read. *pathconv* is used to control 2 point path conversion. If set to 0, 2 point manhattan paths are converted to rectangles. If set to 1, 2 point manhattan paths are set to H/VSegs. If set to 2 (the default), paths remain as paths. *convLayers* determines which layers are imported. If set to 0 (the default), all layers found in the GDS2 file are converted. If set to 1, only layers that are defined in the techFile with gds layer number/datatypes are imported. If set to 2, only a single layer will be imported, defined by *layer* and *datatype*. If *openTopCell* is true, all potential top cell candidates are opened in the gui. A top cell candidate is any cell that is not referenced by another cell, and is not empty. If *setLibName* is true, the library name is set to that of the GDS2 library name. If *convertVias* is true, Glade will convert via cells to Glade vias, and instances of these via cells to viaInsts. *duplicates* controls handling of duplicate cell definitions. If 0, duplicate cells replace any existing cell definitions. If 1, duplicate cells definitions are ignored. If 2, duplicate cell data is merged into existing cells. *viewName* sets the view name of cellViews created during import GDS2.

bool ok = ui.exportGds2 (char * libName, char * viewNames, char * gdsFileName, bool outputInstAttrs=0, bool outputNetAttrs=false, bool outputAllCells=true, char * topCellName =NULL, int instAttr=0, int netAttr=0, bool compressed=False, bool reportCells=False, double grid=0.005, bool writeViaCells=True)

Export a GDS2 file *gdsFileName* from the library *libName*. *viewNames* is a space seperated list of view names to export. If *outputAllCells* is true then all cells in the library are output to the GDS2 file and *topCellName* is ignored. If *outputAllCells* is false then *topCellName* is a space or comma delimited list of cells to output. If *outputInstAttrs* is true then instance names are output as GDS2 attributes with attribute number given by instAttr. If *outputNetAttrs* is true then net names are output as GDS2 attributes with attribute number given by netAttr. If *compressed* is true the GDS2 file is written in gzip compressed format according to RFC1951. If *reportCells* is true, cells are reported in the message window as they are written. *grid* specifies the manufacturing grid, used to snap vertices of circles/ellipses as they are converted to polygons on export. *writeViaCells* if true will write vias as cells and viaInsts as instances, else vias will get flattened.

bool ok = ui.importLef (char * libName, char * lefFileName, bool compressed= 0, bool generateLabels= 1, float size=0.25. bool allPinShapes=false)

Import a LEF file *lefFileName* into the library *libName*. The library will be created if it does not already exist. If *compressed* is true a gzip compressed format file is expected and will be uncompressed during LEF in. If *generateLabels* is true, text labels will be generated on the Text layer for each pin in the LEF macro. *size* sets the size of the generated labels. If *allPinShapes* is true (1), then text labels are generated for all pin shapes.

bool ok = ui.exportLef (char * libName, char * lefFileName, bool technology=1, bool allCells=1, char * powerNets=NULL, char * groundNets=NULL)

Export a LEF file *lefFileName* from library *libName*. If *technology* is true, the LEF technology section will be included in the LEF file. If *allCells* is true, all library cells will be output, else only the current open cellView will be output. The string *powerNets* is a space delimited list of net names. Any pins with a name in this list will have their +USE attribute set to POWER. Similarly, the string *groundNets* is a space delimited list of net names; any pins with a name in this list will have their +USE attribute set to GROUND.

bool ok = ui.importDef (char * libName, char * viewName, char * defFileName, bool ecoMode=0, bool compressed=0, bool reportMissingPins=1, bool importSpecial=1, bool importRegular=1, bool reportUnplacedComps=0)

Import a DEF file *defFileName* into the library *libName*, which must exist. The cellName is determined from the DEF DESIGN keyword and the view name from *viewName*. If *ecoMode* is true then the COMPONENTS and PINS sections only are read, and existing components and pins will have their origin and orientation updated from the DEF file. If *compressed* is true a gzip compressed format file is expected and will be uncompressed during DEF in. If *reportMissingPins* is true, missing net connections to pins will be reported. If *importSpecial* is true then the SPECIALNETS section is imported; if *importRegular* is true then the NETS section is imported. If *reportUnplacedComps* is true then any components with a placement status of UNPLACED will be reported.

bool ok = ui.exportDef (char * libName, char * cellName, char * viewName, char * defFileName, bool comps=1, bool pins=1, bool regular=1, bool special=1, bool regularRouting=1, bool specialRouting=1)

Export a DEF file *defFileName* from the library *libName*, cell *cellName* and view *viewName*. If *comps* is true the COMPONENTS section will be output; if *pins* is true the PINS section will be output; if *regular* is true the NETS section will be output; if *special* is true the SPECIALNETS section will be output. If *regularRouting* is true then routing from the NETS section is output, else just the connectivity. If *specialRouting* is true then routing from the SPECIALNETS section is output, else just the connectivity.

bool ok = ui.importVerilog (char * libName, char * verilogFileName, char * powerNet, char * groundNet, char * flatViewName, bool flatten, char * topCellName, int hPinLayer, int vPinLayer, double aspect, double utilisation)

Import a Verilog file *verilogFileName* into library *libName*. Cells with names matching the verilog module names are created with a view type of netlist. *powerNet* and *groundNet* specify the supply and ground nets used to resolve 1'b1 and 1'b0 references respectively. If *flatten* is true the Verilog netlist will be flattened into view *flatViewName*; *topCellName* is used as the top cell of the design to flatten. *hPinLayer* and *vPinLayer* are the layer numbers that are used for pins created in the flattened view. *aspect* is the aspect ratio of the resulting boundary layer created in the flattened view and *utilisation* sets the area out the boundary layer such that the total cell area divided by the boundary area equals the utilisation.

bool ok = ui.exportVerilog (char * libName, char * cellName, char * viewName, char * verilogFileName, bool flatMode=True)

Export a Verilog file *verilogFileName* from library *libName*, cell name *cellName*, and view name *viewName*. If *flatMode* is true, the verilog will be flattened else a hierarchical netlist will be written.

bool ok = ui.importECO (const char * ecoFileName)

Import an ECO file from file *ecoFileName* into the current open cellView .

bool ok = ui.importOasis (char * libName, char * oasisFileName, bool dubiousData=True, bool allowNonPrintingChars=False, bool reportCells=False, bool openTopCell=False, double scale=1.0, double xoffset=0.0, double yoffset=0.0, int csen=0, int duplicates=0, char *viewName="layout")

Import an OASIS file *oasisFileName* into the library *libName*. The library is created if it does not already exist. If *dubiousData* is true, dubious constructs in the Oasis data are reported. is *allowNonPrintingChars* is true, non-printing characters will be allowed to be read; normally Oasis only permits printable characters in a-string or n-string types. If *reportCells* is true, cells are reported in the message window as they are read. If *openTopCell* is true, all potential top cell candidates are opened in the gui. A top cell candidate is any cell that is not referenced by another cell, and is not empty. *scale* allows scaling of all input data by the factor specified. *xoffset* will add the specified offset to all x coordinate data, *yoffset* will add the specified offset to all y coordinate data. *csen* controls case sensitivity, 0 means preserve case, 1 converts to uppercase, 2 to lowercase. *duplicates* controls handling of duplicate cell definitions. If 0, duplicate cells replace any existing cell definitions. If 1, duplicate cells definitions are ignored. If 2, duplicate cell data is merged into existing cells. viewName sets the view name of cellViews created during import. *viewName* sets the view name of imported cellviews.

bool ok = ui.exportOasis (char * libName, char * viewNames, char * oasisFileName, bool outputAllCells = True, bool outputChildCells = True, char * topCellName = NULL, bool strict = False, bool cblock = False, bool cellOfsets = False, bool reportCells = False, double grid=0.005)

Export an OASIS file *oasisFileName* from library *libName*. All views specified in the space or comma delimited list *viewNames* are output. If *outputAllCells* is true then all cells in the library are output to the GDS2 file and *topCellName* is ignored. If *outputAllCells* is false then *topCellName* is a space or comma delimited list of cells to output. If *strict* is true, the OASIS file is written in STRICT mode. If *cblock* is true, CBLOCK compression is used which can substantially reduce the output file size. If *cellOfsetts* is checked in STRICT mode, the property S_CELL_OFFSET is written for each cell in the cellname table so that random access to cells are possible allowing e.g. multithreaded reading of the OASIS file. If *reportCells* is checked, cells are reported in the message window as they are written.

bool ok = ui.importDxf (char * libName, char * cellName, char * dxfFileName, int dbu=1000)

Import a DXF file *dxfFileName* into the library *libName*. The library is created if it does not already exist. The DXF file is imported into a cell with name *cellName* and viewName layout.

bool ok = ui.exportDxf (char * libName, char * cellName, char * dxfFileName, bool outputText=1, bool allLayers=1)

Export a DXF file *dxfFileName* from the library *libName*, cell *cellName* and view name layout. If *outputText* is true then text labels are output. If *allLayers* is true then all layers are output to the DXF file, else only the currently visible layers are output.

bool ok = ui.importCDL (const char * libName, const char * cdlFileName)

Import a CDL file *cdlFileName* into the library *libName*.

bool ok = ui.exportCDL (char * libName, char *cellName, char *viewName, char * cdlFileName, char *globals, bool annotateXY=0, bool microns= 0, bool rmodel=0, char * rpropname="r", bool cmodel=0, char *cpropname= "c", double filterCapLimit=-1.0, bool mergeCaps= 0)

Export a flat CDL file *cdlFileName* from the library *libName* with cell *cellName*, view *viewName*. *globals* is a space delimited list of global net names e.g. VDD and VSS. If *annotateXY* is true, XY coordinates of instances are written in the CDL file as $X= / $Y= values. If *rmodel* is true (1) then the resistor model name is reported, else the resistor value (R=...) is reported. *rpropname* is the property that is used to report the resistor value and should be a property of the resistor extraction pcell. *cmodel* and *cpropname* act similarly for capacitors (but not for parasitic capacitors which are always reported by value). If a positive *filterCapLimit* is specified, any parasitic capacitances below this limit (in Farads) will not be written in the CDL file. If *mergeCaps* is true (1) then parasitic caps between net pairs are lumped all togther and reported only once per net pair.

ui.schHNLOut (char *libName, char *cellName, char *viewName, char *cdlFileName, char *switchList, char *stopList, char *globals)

Export a hierarchical CDL file *cdlFileName* from the library *libName* with cell *cellName* and view *viewName*. *switchList* is a space delimited list of view names the netlister can switch into e.g. "schematic symbol". *stopList* is a space delimited list of views the netlist can stop on, which should have a NLPDeviceFormat string property to describe the netlist format for the cellView . *globals* is a space delimited list of global net names.

bool ok = ui.checkExtracted(cellView *cv)

Checks if a schematic cellView has been extracted since last modified.

int ui.check(char *libName, char *cellName, char *viewName)

Checks a schematic or symbol cellView for errors, and returns the number of errors.

ui.zoomIn ()

Zoom in according to the current zoomin factor.

ui.zoomIn (int x1, int y1, int x2, int y2)

Zoom in to the area given by x1 y1 x2 y2.

ui.zoomOut ()

Zoom out according to the current zoomout factor.

ui.zoomOut (int x1, int y1, int x2, int y2)

Zoom out by the area given by *x1 y1 x2 y2*.

ui.execPythonFile (char *fileName)

Execute the python script given by *fileName*.

ui.deleteCell(char *libName, char *cellName, char *viewName)

Delete the cell specified by *libName*, *cellName* and *viewName* .

ui.renameCell(char *libName, char *cellName, char *viewName)

Rename the cell specified by *libName*, *cellName* and *viewName* . A dialog will be displayed prompting for the new cell name.

ui.copyCell(char *libName, char *cellName, char *viewName)

Copy the cell specified by *libName*, cellName and *viewName* . A dialog will be displayed prompting for the new cell name.

ui.properties(char *libName, char *cellName, char *viewName)

Display the properties of the cell specified by *libName*, *cellName* and *viewName* .

ui.biasCells(cellView cv, int layer, int biasFactor, int xgrid, int ygrid, int allCells)

Bias cell(s). *cv* is the cellView of the cell to bias, or any cell in the library . *layer* is the layer to bias, and *biasFactor* is the amount to bias the layer in database units. A positive *biasFactor* will grow the shapes, a negative biasFactor will shrink the shapes. X*grid* and *ygrid* are the snap grids to snap resulting shapes to, in database units. If *allCells* is true (1), then all cellViews in the library containing cv will be biassed.

ui.biasCell(cellView cv, int layer, int biasFactor, int xgrid, int ygrid)

Bias a single cell.

ui.scaleCells(cellView cv, double scaleFactor, int grid, bool allCells)

Scale cell(s). *cv* is the cellView of the cell to bias, or any cell in the library . *scaleFactor* is the scale factor to apply to the cell(s). *grid* is the snap grid to snap resulting shapes to, in database units. If *allCells* is true (1), then all cellViews in the library containing cv will be scaled.

ui.scaleCell(cellView cv, double scaleFactor, int grid)

Scale a single cellView by *scaleFactor*. *Grid* is the snap grid to snap shapes to.

int retval = ui.compareCells(char *libName1, char *cellName1, char *viewName1, char *libName2, char *cellName2, char *viewName2, int compareLayer, bool hier, bool countShapes)

Compares two cellViews using an XOR operation using a simple non-tiled approach. This is good for small-ish cells or less than a few thousand transistors/shapes. The comparison is done for *compareLayer*; if this is set to -1 all layers in the cellViews are compared, else just the layer specified. If *hier* is false(0, the default), then the comparison is done at the top level only; if true (1) then it is done hierarchically. The function returns 0 if the two cellViews are identical, -1 if an error occurred e.g. different number of layers in the cells, or different number of shapes (but see countShapes), or the number of differences found. If *countShapes* is false (0, the default) then the number of shapes may differ between the cells, but the XOR result must match.

int retval = ui.compareCells2 (char *libName1, char *cellName1, char *viewName1, char *libName2, char *cellName2, char *viewName2, int compareLayer, bool hier, bool multiThreaded=1, int maxThreads=QThread::idealThreadCount(), bool tileAuto=1, int tileWidth=1, int tileHeight=1, int outputLayer=TECH_DRCMARKER_LAYER)

Compares two cellViews using a tiled XOR operation. The comparison is done for compareLayer; if this is set to -1 all layers in the cellViews are compared. If *hier* is true (1), then the comparison is done hierarchically. The function returns 0 if the two cellViews are identical, -1 if an error occurred, or the number of differences found. If *multiThreaded* is true (the default), then the layout is tiled and run with *maxThreads* threads. If *tileAuto* is true (the default), an intelligent tiling algorithm is used, else tile widths and heights must be specified.

bool retval = ui.runLVS(char *libName, char *cellName, char *viewName, char *netlist, char *globalNets= NULL, char *workDir= ".", bool isHierNetList= false, const char *delimiter= "/", const char *topCellName=NULL, bool checkDeviceProps=false, bool collapseLikeSized= false, bool noCollapseFingered= false,bool noCollapseChains= false, bool warnChains= false, bool caseFoldNets= false,bool noLocalMatching= false, bool noOptLabelling= false, bool matchProperties=false, bool matchPorts=false, bool warnZeroNets= false, bool verbose= false,const char *errorLimit= NULL, const char *netSizeLimit= NULL,const char *progressLimit= NULL, const char *suspectNodeLimit= NULL,db_Float64 tranTolerance=10.0, db_Float64 capTolerance= 10.0 ,const char *equivInFileName= NULL, const char *equivOutFileName= NULL)

Runs LVS, comparing the cellview given by *libName*/*cellName*/*viewName*, which should be an extracted cellView , against the Spice/CDL netlist given by *netlist*. *globalNets* is a space delimited list of global net names. *workDir* is used for the creation of temporary files. If *isHierNetList* is true, then the netlist is treated as hierarchical and will be flattened with delimiter character delimiter and top cell name topCellName. If *checkDeviceProps* is true, device properties e.g. W, L or MOS devices are checked according to the tolerance specified by *tranTolerance* and *capTolerance*.

The remainder of the parameters correspond to Gemini options.

ui.updateLibBrowser()

Updates (refreshes) the library browser.

ui.updateLSW()

Updates (refreshes) the LSW.

bool ok = ui.isLayerVisible(const char *layerName, const char *purpose)

Returns True if the layer specified by *layerName* and *purpose* is visible in the current cellView or False if it is invisible, or there is no current cellView .

bool ok = ui.isLayerSelectable(const char *layerName, const char *purpose)

Returns True if the layer specified by *layerName* and *purpose* is selectablein the current cellView or False if it is invisible, or there is no current cellView .

bool ok = ui.setLayerVisible(const char *layerName, const char *purpose, bool val)

Returns True if the layer specified by *layerName* and *purpose* can be set visible in the current cellView or False if there is no current cellView .

bool ok = ui.setLayerSelectable(const char *layerName, const char *purpose, bool val)

Returns True if the layer specified by *layerName* and *purpose* can be set visible in the current cellView or False if there is no current cellView .

7.3.29.1 Extending Glade by creating menus / bindkeys etc.

cvar.uiptr

A global pointer to the ui class instanciation in Glade. Use this rather than creating your own ui variable using ui(). For example:

```
gui=cvar.uiptr
gui.OpenCellView("default", "nand", "layout")
```

Although you can use e.g. ui().<functionName()>, this will not work for commands like createAction() which only work with the existing instanciated ui object.

menu = ui.createMenu(const char *name)

Creates a menu called *name* in the menu bar.

menu = ui.createMenu(QMenu menu, const char *name)

Creates a submenu called *name* in *menu*.

ui.addSeparator(QMenu menu)

Adds a separator to the *menu*.

action = ui.createAction(const char *name, const char *cmd)

Creates an action called *name* with a command *cmd*. The command should be a valid Python command. An action defines a common command that can be invoked by any or all of a menu item, a bindkey or a toolbar button.

action = ui.createAction(const char *name, const char *cmd, QActionGroup group)

Creates an action called *name* with a command *cmd* that is part of an actionGroup *group*. The command should be a valid Python command. An action defines a common command that can be invoked by any or all of a menu item, a bindkey or a toolbar button.

actionGroup = ui.createActionGroup()

Create an actionGroup.

ui.createMenuItem(menu, action)

Adds the *action* to the *menu*. The action name will be shown on the menu, along with any key binding defined for the action.

setBindKey(action, const char *keysequence)

Sets the bindkey for *action*. *keysequence* can be a key e.g. "k" or a combination e.g. "Ctrl+p", "Shift+p", "Alt+p"

icon = ui.createIcon (const char *fileName)

Creates an icon from an image file (.png format)

ui.setIcon (QAction action, const char *fileName)

Sets the icon for an *action* from the image file (.png format)

ui.setIcon (QAction action, QIcon *icon)

Sets the *icon* for an *action*

toolbar = ui.createToolBar (const char *name)

Creates a tool bar with name *name*.

ui.createToolBarItem (QToolBar *toolBar, QAction *action)

Adds an *action* to a *toolbar*.

ui.addSeparator (QToolBar *toolBar)

Adds a separator to a *toolbar*.

An example of a python script for setting up a user-defined menu is as follows:

```
# define some user function
def myFunction() :
print "Hello World!"
gui = cvar.uiptr
menu = gui.createMenu("MyMenu")
action=gui.createAction("MyAction", "myFunction()")
gui.createMenuItem(menu, action)
gui.setBindKey(action, "!")
```

7.3.30 utils class

7.3.31 via class

The Via class represents a via master, which is a kind of special cellView . Instances of vias are called viaInsts, and are simplified forms on insts. Normally a via is created with a given name; its shapes are added with addViaLayer(), and then the via is added to the library using library ::addVia().

via v = via

via v = via(char *name)

Creates a via object. The second type of constructor creates a via with name *name*.

v.setViaName(char *name)

Sets the via's *name*.

char *name = v.getViaName()

Gets the via's name.

v.addViaLayer(int layer, Rect geom)

Adds a via *layer*. Note that vias can currently only contain rectangular shapes.

int num = v.getNumLayers()

Gets the number of layers in the via. Typically this is 3 (two conductor layers and one via layer).

int first_layer = v.getFirstLayer()

Gets the first (lower) layer of the via.

int last_layer = v.getLastLayer()

Gets the last (upper) layer of the via.

viaLayer vl = v.getViaLayerList()

Returns a viaLayer list which is a structure of the form :

```
struct viaLayer {
   int layer;
   Rect geom;
   viaLayer *next;
} viaLayer;
```

So for example given a viaLayer vl, its rectangle is give by vl.geom

v.setViaLayerList(viaLayer vl)

Sets the via's viaLayer list. Normally the viaLayer list is created using addViaLayer().

v.setViaDefault(bool flag)

Sets the via as a default via if flag is True.

bool v.getViaDefault()

Returns True if the via is a default via.

int other_layer = v.getOtherViaLayer(int layer)

Given one of the via's conducting layers, returns the 'other' conducting layer.

v.setLib(library lib)

Sets the library for this via. Normally this should not be used, as a via, after creation, should be added to a library using lib.addVia(v).

library lib = v.lib()

Gets the library that contains this via.

v.bBox(Rect box)

Updates the via's bounding box. Note this creates a new bounding box which is the union of the existing bounding box and the new box.

Rect box = v.bBox()

Gets the via's bounding box.

v.setResistance(double r)

Sets the via's resistance in ohms.

double val = v.getResistance()

Gets the via's resistance in ohms.

v.setPattern(char *name)

Sets the via's pattern name

char *name = v.getPattern()

Gets the via's pattern name

7.3.32 viaInst class

A viaInst is a reference to a via, in a cellview. viaInsts are like instances but require less memory and have a specific function, i.e. to be instances of vias which again are a type of cellView with a specific function, i.e to hold rectangular shapes of the via. Normally viaInsts are created in a cellView using the dbCreateViaInst() function.

int coord = vi.left()

Get the left edge of the viaInst's bounding box

int coord = vi.bottom()

Get the bottom edge of the viaInst's bounding box

int coord = vi.right()

Get the right edge of the viaInst's bounding box

int coord = vi.top()

Get the top edge of the viaInst's bounding box

bool vi.offGrid(int grid)

Checks if a viaInst's origin is on the grid *grid*, which is in database units.

vi.orient(orient_t orient)

Set the viaInst's orientation. *orient* can be one of: R0, R90, R180, R270, MX, MXR90, MY, MYR90.

db_Orient vi.orient ()

Get the viaInst's orientation.

vi.origin(Point origin)

Sets the viaInst's origin to the Point *origin* .

int vi.origin()

Get the viaInst's origin.

Rect box = vi.bBox()

Get the viaInst's bounding box.

dbtype_t_t vi.objType()

Returns the objects type as VIAINST

char * vi.objName()

Returns the print name i.e. "VIAINST"

int vi.getLowerLayer()

Returns the viaInst's lower layer number.

int vi.getCutLayer()

Returns the viaInst's cut layer.

int vi.getUpperLayer()

Returns the viaInst's upper layer.

vi.transform(transform trans)

Transform the instance by the given transform.

vi.scale(double scalefactor, double grid)

Scale the instance origin coordinates by *scalefactor*, snapping to *grid*.

vi.Move(cellView dest, Point delta, bool opt= 1)

Move the viaInst's origin by *delta*. If *opt* is 1 then the database is re-optimised for the new inst position. If there are a lot of objects to move it makes sense to turn this off and instead use the cellView update() function after moving them all.

dbObj obj = vi.Copy(cellView dest, Point delta)

Copy the viaInst's . *dest* is the destination cellview, *delta* is the offset from the current origin.

dbObj obj = vi.Flatten(cellView dest, transform trans)

Flatten the viaInst's into the cellView *dest*, with the given transform *trans*.

int vi.getViaIndex()

Returns the viaInst's via index.

vi.setViaIndex(int index)

Set the viaInst's via index.

via v = vi.getVia()

Gets the via master for this viaInst

7.3.33 Vector class

A Vector class represents a direction.

v= Vector

Creates a Vector object v. The Vector is initialised to (0 0) by default.

v = Vector(x, y)

Creates a Vector v, initialised with the values *x* and *y*.

setX(int x)

Set the Vector X value.

setY(int y)

Set the Vector Y value.

x = v.x

Get the X component of the Vector v.

y = v.y

Get the Y component of the Vector v.

operator +

Adds the two Vectors.

operator -

Subtracts the two Vectors.

operator *

Returns a Vector multiplied by a scalar.

operator /

Returns a Vector divided by a scalar.

double val = v.dotProduct(Vector other)

Returns the dot product of this Vector with *other*.

double val = v.normal()

Returns the normal of this Vector.

double val = v.distance(Vector other)

Returns the euclidian distance between this Vector and *other*.

7.3.34 vertex class

A vertex is a point on a shape. It is derived from a dbObj so it can be selectable; it also refers to a dbObj . Vertices are used when selecting a vertex of e.g. a rectangle or polygon.

vert = vertex(Point p)

Creates a vertex with coordinate p.

vert = vertex(int x, int y)

Creates a vertex with the specified xy coordinates.

dbtype_t_t vert.objType()

Returns the objects type - VERTEX.

char * vert.objName()

Returns the print name i.e. "VERTEX"

operator ==

Is one vertex equal to another.

operator !=

Is one vertex different from the other.

vert.SetObj(dbObj obj)

Sets the dbObj associated with this vertex.

dbObj vert.GetObj()

Gets the dbObj associated with this vertex.

Rect vert.bBox()

Returns a fake bounding box 10 dbu larger than the vertex itself.

vert.transform(transform trans)

Transforms this vertex according to trans.

vert.Move(cellView dest, Point delta, bool opt = True)

Moves this vertex by *delta*. If *opt* is True then the database is re-optimised for the new vertex position. If there are a lot of objects to move it makes sense to turn this off and instead use the cellView update() function after moving them all.

dbObj newVert = vert.Copy(cellView dest, Point delta)

Copies the vertex. *dest* is the cellView containing the vertex, *delta* is the XY coordinate to move the vertex by during the copy.

vert.transform(transform trans)

Transforms the vertex by the transform *trans*.

int vert.x()

The x coordinate of the vertex.

int vert.y()

The y coordinate of the vertex.

vert.setX(int x)

Set the X coordinate of the vertex.

vert.setY(int y)

Set the Y coordinate of the vertex.

7.3.35 view class

The view class represents a view, which is a representation of a cell. The combination of a cell and a view is a cellView .

view v = view

Creates an view object.

dbObjList<cellView> list = v.cellViews()

Get a list of the cellViews for this view.

list = v.getCellViews()

Gets a Python list of the cellViews for this view.

v.name(char *s)

Sets the view's name.

char *s = v.name()

Gets the view's name.

v.setViewType(db_viewType type)

Sets the view's *type*. The viewType can currently be one of layout, schematic, symbol, abstract.

db_viewType type = v.viewType()

Returns the view's type.

v.addCellView(cellView cv)

Adds cv to the view's cellView list.

cellView cv = v.dbFindCellViewByView(char *cellName)

Finds the cellView for this view with cellName *cellName*. If it does not exist, a null pointer is returned.

dbtype_t type = c.objType()

Returns the object's type (VIEW).

char *name = c.objName()

Returns the object's print name ("VIEW").

7.3.36 VSeg class

A VSeg represents a wiring segment for place&route data. It is a 2 vertex vertical path. A VSeg is normally created by cellView::dbCreateVSeg()

v.setPoints(int x1, int y1, int x2, int y2)

Sets the vertices of the VSeg

v.left()

Gets the leftmost X coordinate of a VSeg.

v.right()

Gets the rightmost X coordinate of a VSeg.

v.bottom()

Gets the lowest Y coordinate of a VSeg.

v.top()

Gets the highest Y coordinate of a VSeg.

int w = v.width()

Gets the VSeg width.

v.setStyle(int s)

Sets the VSeg style, i.e. the type of the path end. The style can be one of: 0 - truncate, 1 - round, 2 - extend, 4 - varextend, 8 - octagonal. Python global variables TRUNCATE, ROUND, EXTEND, VAREXTEND, OCTAGONAL are defined to these values.

int s = v.getStyle()

Gets the VSeg style.

v.setSpecial(bool val)

Sets the VSeg's specialNet status

v.isSpecial()

Returns true is the VSeg is a specialNet.

v.setNet(net n)

Sets the VSeg net

v.getNet()

Returns the VSeg's net.

Rect b = v.bBox()

Get the bounding box of this VSeg.

dbtype_t_t t = v.objType()

Returns the object type of this VSeg as VSEG.

char *name = p.objName()

Returns the object name of this VSeg as "VSEG".

int n = v.nPoints()

Returns the number of points of the VSeg (2).

Point * pts = v.ptlist()

Returns the point list of this VSeg as a C array of Points.

double a = v.area()

Get the area of this VSeg.

int p = v.perimeter()

Get the perimeter of this VSeg.

Point v.origin()

Returns the origin point of a VSeg

v.setOrigin(Point p)

Sets the origin of a VSeg.

int e = v.extent()

Returns the extent, i.e. the length of the VSeg.

v.setExtent(int e)

Sets the extent of the VSeg.

Point p = v.getFirstVertex()

Gets the first vertex of this VSeg

Point p = v.getLastVertex()

Gets the last vertex of this VSeg.

bool v.ptInPoly(Point p)

Returns true if the Point p is contained in the VSeg or on its edges.

v.Move(cellView dest, Point delta, bool opt = True)

Move this VSeg by distance *delta*. If *opt* is True then the database is re-optimised for the new VSeg position. If there are a lot of objects to move it makes sense to turn this off and instead use the cellView update() function after moving them all.

dbObj obj = v.Copy(cellView dest, Point delta, int layer = -1)

Copy this VSeg to cellView *dest*, with offset *delta*. If *layer* is a positive integer the VSeg will be copied to the new layer number.

dbObj obj = v.Flatten(cellView dest, transform trans)

Flatten this VSeg into cellView *dest* with transformation *trans*.

int dist = v.getNearestEdge(Point p, segment &edge, bool centreLine=True, bool outLine=True)

Gets the nearest segment *edge* to the VSeg from the Point *p* and returns the distance. If *centreline* is True, the centre line of the VSeg is considered. If *outLine* is True, the outline edges of the VSeg are considered.

int dist = v.getNearestVertex(Point p, vertex &vert)

Gets the nearest vertex vert to the VSeg from the Point *p* and returns the distance.

char *name = v.getNetName()

Returns the VSeg's net name as a string.

bool ok = v.offGrid(int grid)

Returns true if the VSeg is offgrid.